An Introduction to Least Commitment Planning

Daniel S. Weld

■ Recent developments have clarified the process of generating partially ordered, partially specified sequences of actions whose execution will achieve an agent's goal. This article summarizes a progression of least commitment planners, starting with one that handles the simple STRIPS representation and ending with UCPOP, a planner that manages actions with disjunctive precondition, conditional effects, and universal quantification over dynamic universes. Along the way, I 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 UCPOP.

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 AI 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: (1) planning and (2) situated action. These two techniques have different strengths and weaknesses, as I illustrate later. 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 easily be computed from the current state of the world (that is, 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 the conference engenders many subgoals: booking plane tickets, getting to the airport, changing dollars to francs, making hotel reservations, finding the hotel, and so on. 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 (for example, attempting to make hotel reservations after executing the trans–Atlantic fly action) could lead to failure (that is, a miserable night on the streets of Paris among the city's many hungry canines).

However, if the goal is to stay alive while you play a fast-paced video game, advanced planning might be less important. Instead, it might 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 because the time would be better spent shooting at the myriad enemy. Domain-specific situated-action systems are often implemented as production systems or with hard-wired 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 article, I neglect the situated techniques and concentrate on the converse approach to synthesizing actions: planning. Planners are characterized by two dimensions that 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

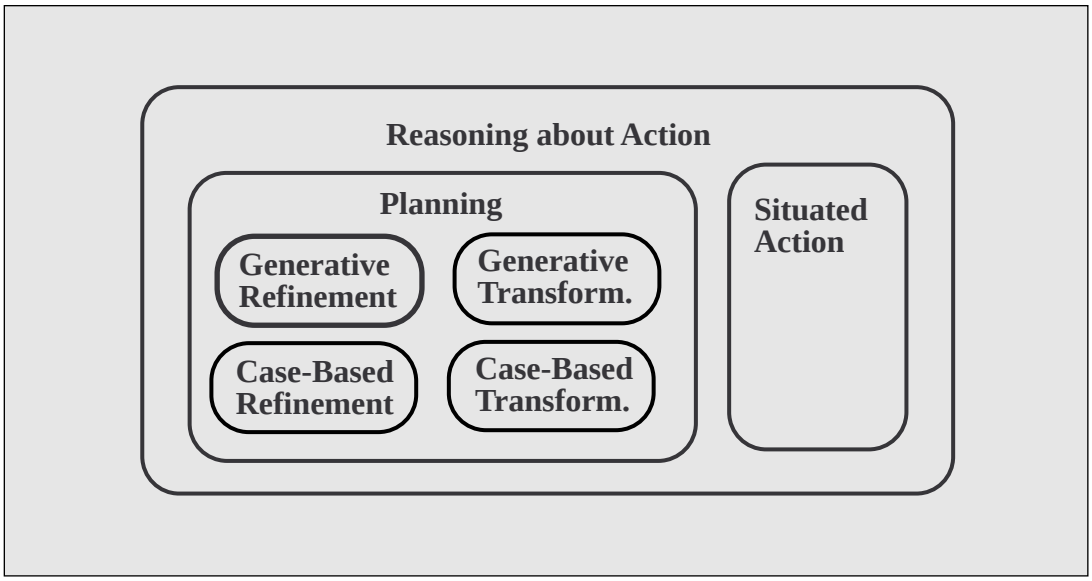

Figure 1. Major Approaches for Reasoning about Action.

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, but *case-based planners* use a library of previously synthesized plans or plan fragments.[1] 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 might 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 that 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.

In the next sections, I define the planning problem more precisely and then describe algorithms for solving the problem. I restrict my attention to generative, refinement planning, but the algorithms can be adapted to transformational and case-based approaches (Hanks and Weld 1992). As we see, planning is naturally formulated as a search problem, but the choice of search space is critical to performance.

The Planning Problem

Formally, a planning algorithm has three input: (1) a description of the world in some formal language, (2) a description of the agent's *goal* (that is, what behavior is desired) in some formal language, and (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 that, when executed in any world satisfying the initial state description, will achieve the goal. Note that this formulation of the planning problem is 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 representation 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 description assumes that all effects are deterministic. It would be 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

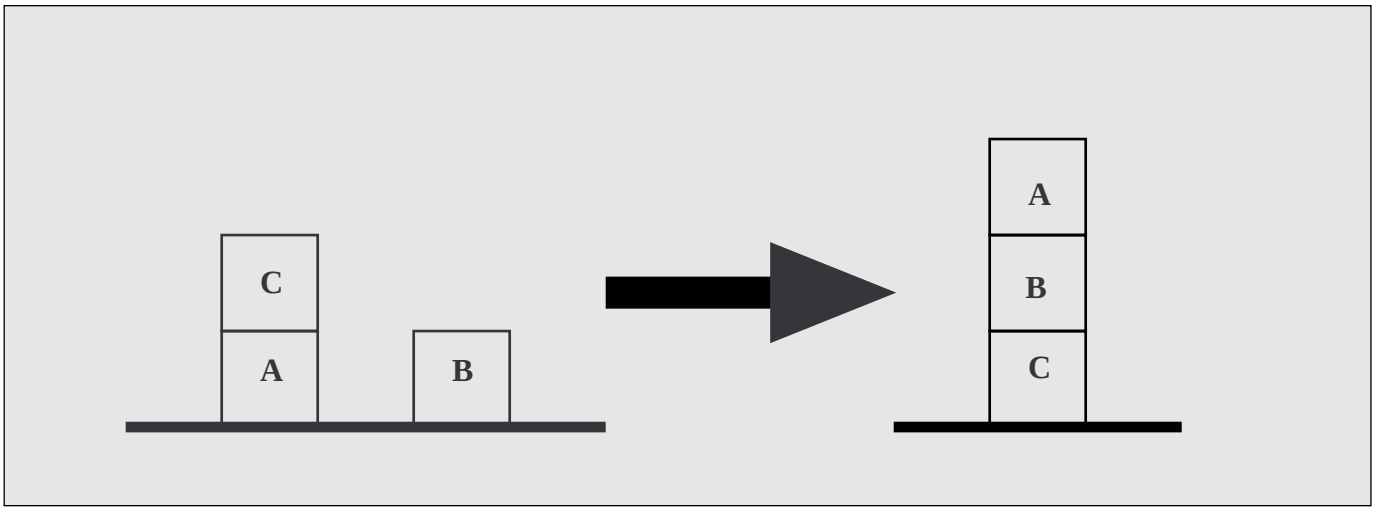

Figure 2. Initial and Goal States for the Sussman Anomaly Problem in the Blocks World.

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 article, I explain how to build planners for several languages, but they all make some simplifying assumptions:

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 might 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 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 I can describe some simple algorithms. Alternatively, skip ahead to the section on advanced topics where I describe extensions to the algorithms that relax these assumptions.

I start our discussion of planning with a simple language: the propositional STRIPS representation.[2] The propositional STRIPS representation describes the initial state of the 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:

(on *A* Table) (on *C A*) (on *B* Table) (clear *B*) (clear *C*) .

Because we require the initial state description to be complete, all atomic formulas not explicitly listed in the description are assumed to be false (this statement is called the *closed-world assumption* [Reiter 1980]). Thus, (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 the goal to remain in the house is rendered inexpressible. Furthermore, the STRIPS representation restricts the type of goal states that can be specified to those matching a conjunction of positive literals. For example, the goal situation shown on the right side of figure 2 could be described as the conjunction of the two literals (on *B C*) and (on *A B*), yielding a simple block-stacking challenge called the Sussman anomaly.[3]

A domain theory, denoted by 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*, however, is a conjunction that can include both positive and negative literals. For example, for the action move-C-from-A-to-Table, we might define the precondition as (and (on *C A*) (clear *C*)) and the effect as (and (on *C* Table) (not (on *C A*)) (clear *A*)).

Actions can be executed only when their precondition is true; in this case, we specified that the robot can 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, and all the negative literals (called the action's *delete list*) are removed.[4] For example, executing move-C-from-A-to-Table in the initial state described earlier 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 formulas are false.

When called with these input—a description of the initial state, a description of the goal, and the domain theory—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 anomaly's initial and goal states (figure 2) and a set of actions such as that described previously, we would like our planner to return a sequence such as

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

As we will see, there are a variety of algorithms that can synthesize these sequences, but some are more efficient than others. We start by looking at planners that are conceptually simple and then look at more sophisticated ways of planning.

Search through World Space

The simplest way to build a planner is to cast the planning problem as 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 this encoding of the blocks world, all actions are reversible; so, I 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 gray. When phrased in this manner, the solution to a planning problem (that is, 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 (Korf 1992; Russell 1992). Because the tradeoffs between these different searching algorithms have been discussed extensively elsewhere, I focus instead on the structure of the search space. A handy way to focus on the search space structure is to specify the planner with a nondeterministic algorithm. This idea might seem strange at first, but I use it extensively in subsequent sections, so it's important to learn it now. In fact, it's simple: When specifying the planning algorithm, one uses a nondeterministic choose primitive. This primitive takes a set of possible options and magically selects the right one. The beauty of this nondeterministic primitive lies in its ease of implementation: It 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 implementor can easily switch between different search strategies in an effort to improve performance.

Progression

Algorithm 1 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.

Algorithm 1: PROGWS(word-state, goal-list, Λ, path)

1. If world-state satisfies each conjunct in goals-list,

2. Then return path,

3. Else let Act = choose from Λ an action whose precondition is satisfied by world-state:

(a) If no such choice was possible,

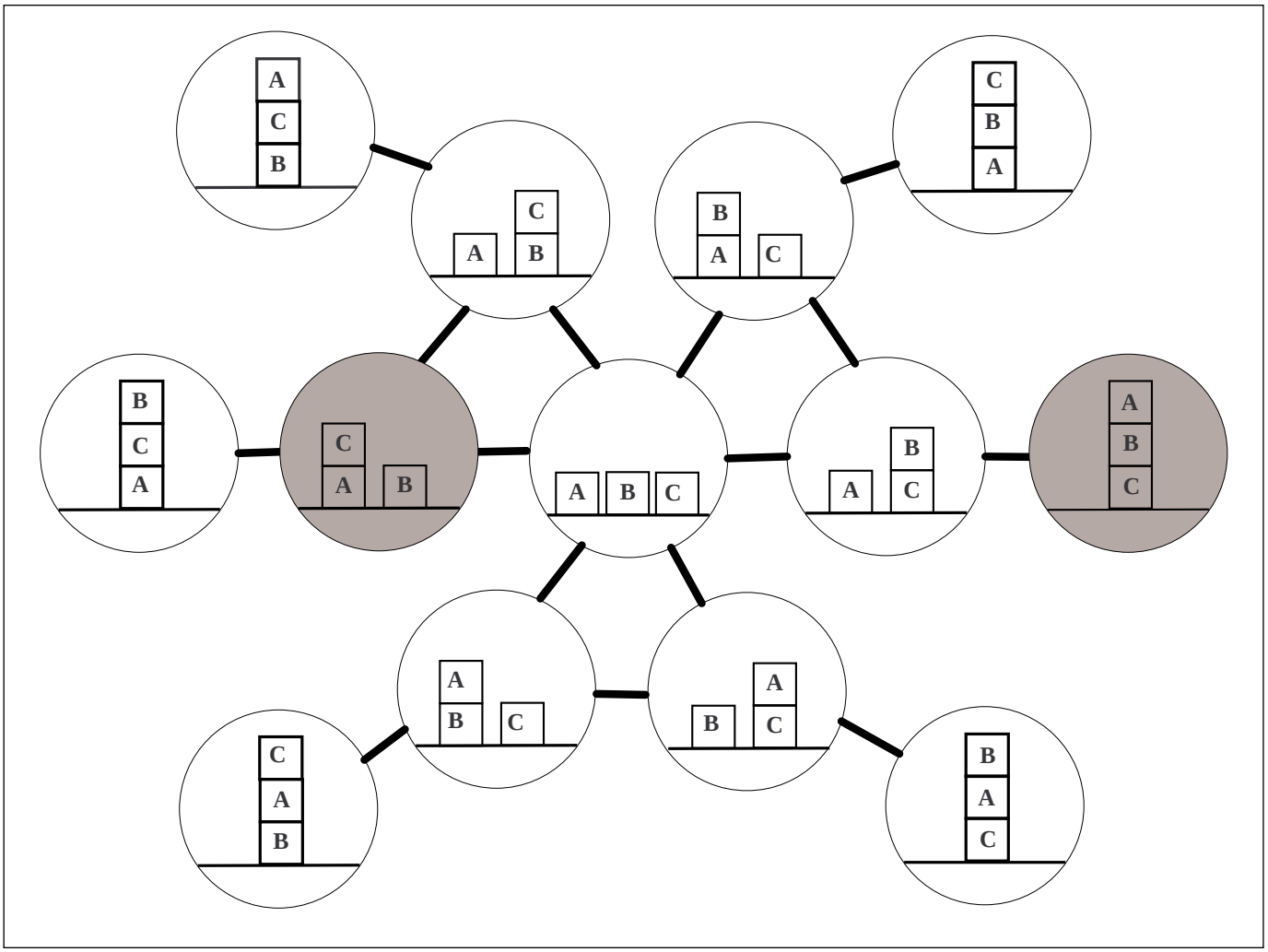

Figure 3. World Space.

(b) Then return failure,

(c) Else let *S* = the result of simulating execution of Act in world-state and return PROGWS(*S*, goal-list, Λ, Concatenate(path, Act)).

The right way to think about a nondeterministic algorithm is with you personally calling the shots every time that the choose primitive gets called. For example, if we try PROGWS on the Sussman anomaly, the first call to the procedure has world-state set to the initial state (the leftmost gray state in figure 3), goal-list set to the implicit conjunction ((on *A B*) (on *B C*)), and path set to the null sequence. Because the initial state doesn't satisfy the goal, execution falls to line 3 of algorithm 1, and choose is called. A moment's thought should convince you that the best choice makes Move-C-from-A-to-Table the first action; so, let's assume that it is this choice that is made by the computer. By giving the program a magical oracle, it can easily find the sequence of three correct choices that lead to a solution. Because 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 electronic mail from the program asking for advice!), then one needs to use search. A simple 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 finds a solution, the breadth-first search version does too (although in the worst case, it might take the searching version exponentially longer).

Regression

Algorithm 1 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 algorithm 2. 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 chooses an action that might possibly help satisfy one of the outstanding goal conjuncts.

Algorithm 2: REGWS(init-state, cur-goals, Λ, path)

1. If init-state satisfies each conjunct in cur-goals,

2. Then return path,

3. Else:

(a) Let Act = choose from Λ, 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, G is undefined, or G ⊃ cur-goals,

(d) Then return failure,

(e) Else return REGWS(init-state, G, Λ, Concatenate(Act, path)).

To illustrate REGWS on a more concrete example, the Sussman anomaly, 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. Because 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 the 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 act action. The result of this regression is another logical sentence that encodes the weakest preconditions that must be true before act is executed to assure that cur-goals will be true after act is executed. This sentence is simply the union of Act's preconditions with all the current goals except those provided by the effects of act:

preconditions(Act) ∪
(cur-goals—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*)).

Because act achieved (on *A B*), regression removed the literal from the sentence (replacing it with the precondition of act, namely, (on *A* Table). Because 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 guaranteed true as long as action preconditions are restricted to conjunctions. Hence, it is okay 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 I discuss them in turn:

First, 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, the 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.

Second, 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 to make cur-goals true after execution. What if one of act's effects directly conflicts with cur-goals? The weakest precondition would thus be undefined because no matter what was true before act, execution would ruin things. A good example is when one tries to regress ((on *A B*) (on *B C*)) through Move-A-from-B-to-Table. Because this action negates (on *A B*), the weakest preconditions are undefined.

Third, if G ⊇ cur-goal, then there's really no point in adding act to the path, for the same reasons that were explained in the first part to the dead-end check. In fact, one can show that *G* ⊃ 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* ⊃ cur-goals renders the test of the first

part unnecessary; however, eliminating it would result in reduced efficiency because many more regressions would be required.

Analysis

My presentation of the PROGWS and REGWS planning algorithms brings up several natural questions. The first questions concern the soundness (that is, If a plan is returned, will it really work?) and the completeness (that is, If a plan exists, does a sequence of nondeterministic choices exist that will find it?) of the algorithms. Although I 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 because 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 regression 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 (that is, have effects that match the goal and have legal regressions). Because 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 can be reversed. I should note that there are a variety of other search techniques (means-ends analysis, bidirectional search, and so on) that I haven't discussed.[5] 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.

Search through the Space of Plans

In 1974, Earl Sacerdoti built a planner, called NOAH, with many novel features. The innovation I 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.[6] 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 4 illustrates one such space. Once again, the initial and goal states are highlighted in gray. 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 although world-state planners had to return the path between initial and goal states, in plan space, the goal state *is* the solution.

Total-Order Planning

At this point, I am forced to confess. I claimed that it's useful to think of planning as search through plan space, and I explained that in plan space, nodes denote plans, but I haven't said what plans really are. In fact, this issue is a subtle one that I 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 this 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, that is a totally ordered sequence of actions (that is, a plan). In fact, if we watch the successive values of path at each recursive call, we get the picture shown in figure 4.

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 appending new actions to the beginning of the sequence.

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 that is restricted to appending the actions. However, I 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 subsection.

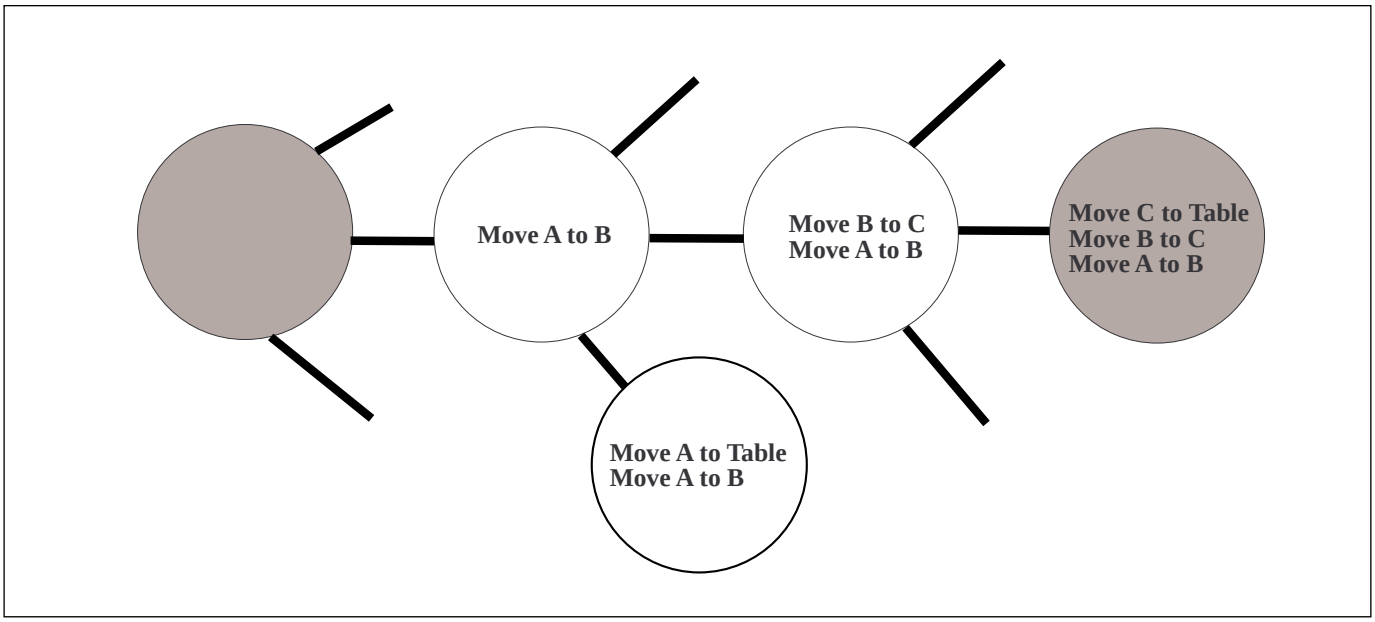

Figure 4. Plan Space.

Partial-Order Planning

Think for a moment about how you might solve a planning problem. For concreteness, I 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 buy a guide to France (to enable choosing hotels and itinerary). However, there's no need to decide (yet) which purchase should be executed first. This idea is 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 planning*—only the essential ordering decisions are recorded.

Plans, Causal Links, and Threats We represent a plan as a three tuple <*A, O, L*> in which *A* is a set of actions, *O* is a set of ordering constraints over *A*, and *L* is a set of causal links (described later). For example, if $A = \{A_1, A_2, A_3\}$, then *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 at least one total order exists that satisfies them. As least commitment planners refine their plans, they must perform constraint satisfaction to ensure the consistency of *O*. Maintaining the consistency of a partially ordered set of actions is just one (simple) example of constraint satisfaction in planning. We see more in subsequent sections.

A key aspect of least commitment is tracking past decisions and the reasons for these 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.[7] To record these dependencies, we use a data structure, called a *causal link*, that was invented by Austin Tate (1977) for use in the NONLIN planner. 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 $A_p \overset{Q}{\rightarrow} A_c$ and store a plan's links in the set *L*.

Causal links are used to detect when a newly introduced action interferes with past decisions. We call such an action a *threat*. More precisely, suppose that <*A*, *O*, *L*> is a plan, and $A_p \overset{Q}{\rightarrow} A_c$ is a causal link in *L*. Let A_t be a different action in *A*; we say that A_t threatens

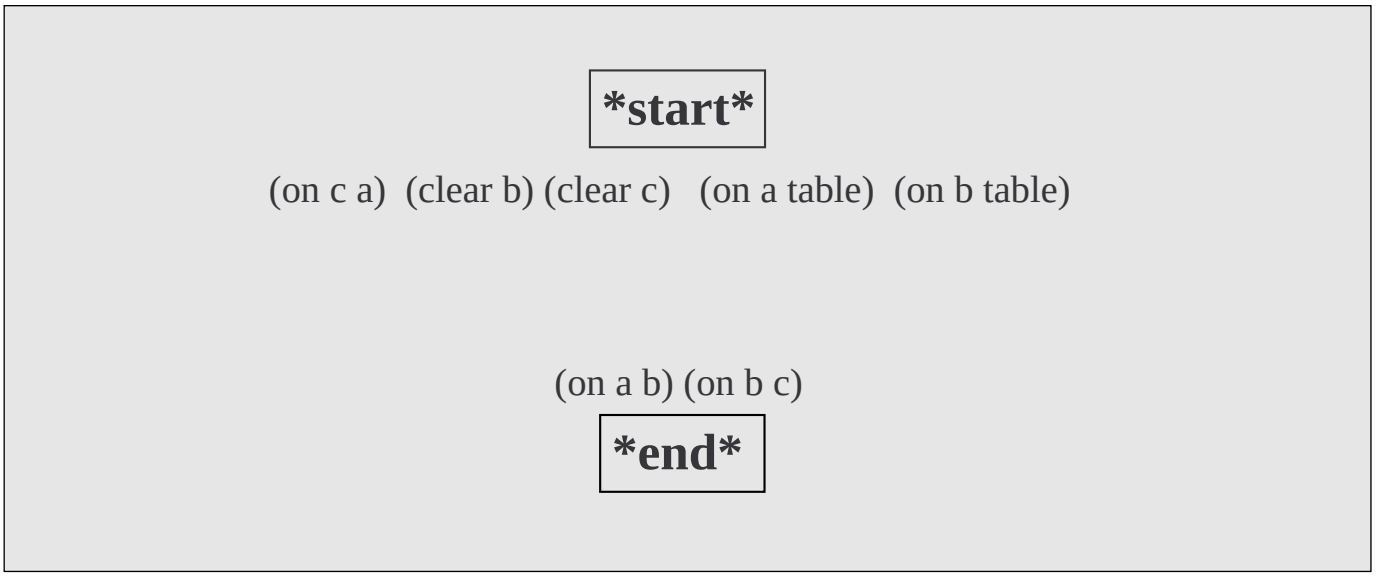

*Figure 5. The Null Plan for the Sussman Anomaly Contains Two Actions: The *start* Action Precedes the *End* Action.*

$A_p \overset{Q}{\rightarrow} A_c$ when $O \cup \{A_p < A_t < A_c\}$ 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 $A_p \overset{Q}{\rightarrow} A_c$, 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, there is a danger that the plan won't work as anticipated. To prevent this situation from happening, the planning algorithm 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*.[8] As we see in subsequent sections, there are other ways to protect against threats as well.

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 incomplete plans. The secret to achieving this uniformity is an encoding trick: The initial state description and goal conjunct can be bundled into a special three tuple called the *null plan*.

The encoding is simple. The null plan of a planning problem has two actions, A = $\{A_0, A_\infty\}$; one ordering constraint, O = $\{A_0, A_\infty\}$; and no causal links, L = {}. All the planning activity stems from these two actions. A_0 is the **start* action*; that is, it has no preconditions, and its effect specifies which propositions are true in the planning problem's initial state and which are false.[9] A_∞ is the **end* action;* that is, 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 5.

The POP Algorithm I now describe a simple regressive algorithm that searches the space of plans.[10] 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, O, of this final plan can still specify only a partial order; in this case, any total order consistent with O is guaranteed to be an action sequence that solves the planning problem. In algorithm 3, 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 $<Q,A_i>$, where Q is a conjunct of the precondition of A_i. (**Note:** many times the identity of A_i is clear from the context, and we pretend that agenda contains propositions, such as Q, instead of $<Q,A_i>$ pairs.)

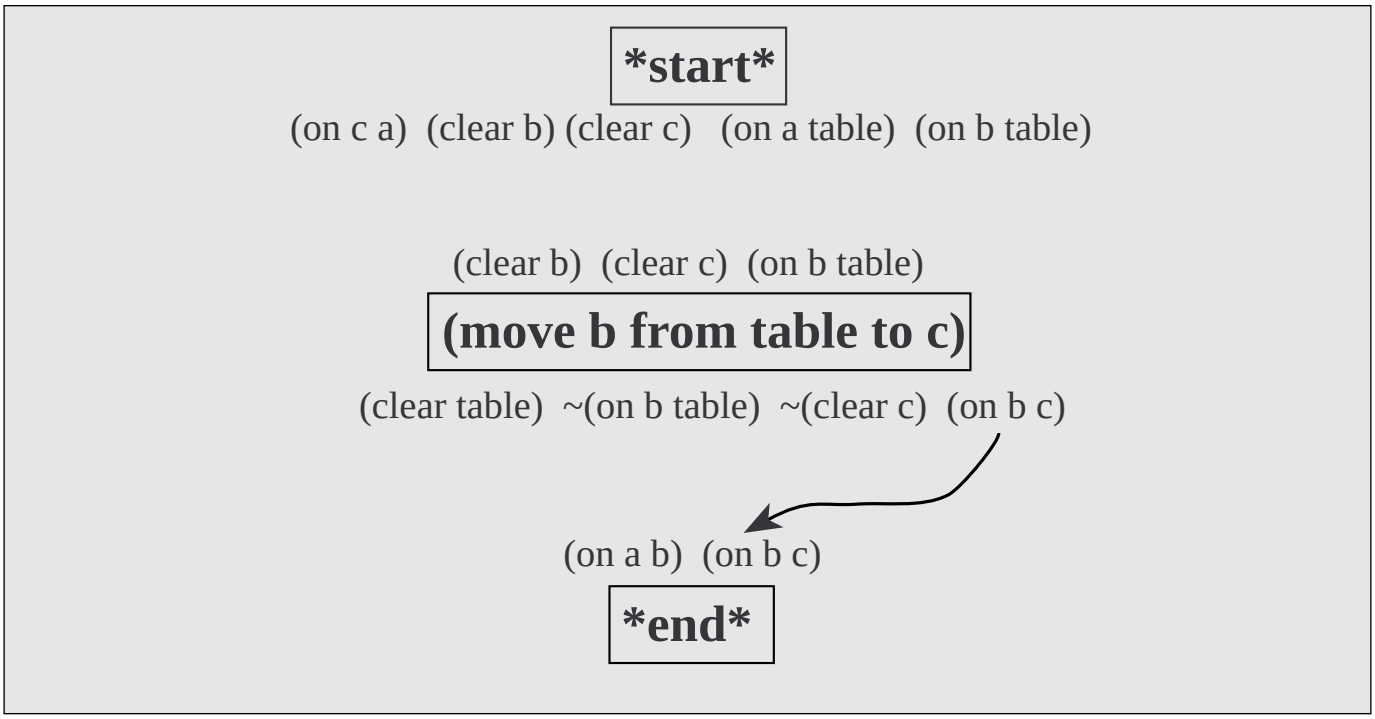

Figure 6. After Adding a Causal Link to Support (on B C), the Plan Is as Shown, and Agenda Contains {(clear B) (clear C) (on B Table) (on A B)} as Open Propositions.

Algorithm 3: POP(<*A*,*O*,*L*>, agenda),

1. **Termination**: If agenda is empty, return <*A*,*O*,*L*>.

2. **Goal selection:** Let <*Q*,A_{need}> be a pair on the *agenda* (by definition, $A_{need} \in A$, 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 Λ or an action already in *A* that can be ordered consistently prior to A_{need}). If no such action exists, then return failure. Let $L' = L \cup \{A_{add} \xrightarrow{Q} A_{need}\}$, and let $O' = O \cup \{A_{add} < A_{need}\}$. If A_{add} is newly instantiated, then $A' = A \cup \{A_{add}\}$ and $O' = O' \cup \{A_0 < A_{add} < A_\infty\}$ (otherwise, let $A' = A$).

4. **Updating of goal set:** Let agenda' = agenda — {<*Q*,A_{need}>}.

If A_{add} is newly instantiated, then for each conjunct, Q_i, of its precondition, add <Q_i,A_{add}> to agenda'.

5. **Causal link protection:** For every action A_t that might threaten a causal link $A_p \xrightarrow{R} A_c$, add a consistent ordering constraint, either

(a) **Demotion:** Add $A_t < A_p$ to *O'*.

(b) **Promotion:** Add $A_c < A_t$ to *O'*.

If neither constraint is consistent, then return failure.

6. **Recursive invocation:** POP(<*A'*,*O'*,*L'*>, agenda', Λ).

It's important to understand how this algorithm works in detail, so I now illustrate its behavior on the Sussman anomaly. When making the initial call, I provide two arguments: the null plan, shown in figure 5, and agenda = {<(on *A B*), A_∞>, <(on *B C*), A_∞>}. Because agenda isn't empty, control passes to line 2 of the POP algorithm. There are two choices for the immediate goal—*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 choice was not written with the nondeterministic choose primitive—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 quickly, but the other choice might lead to enormous

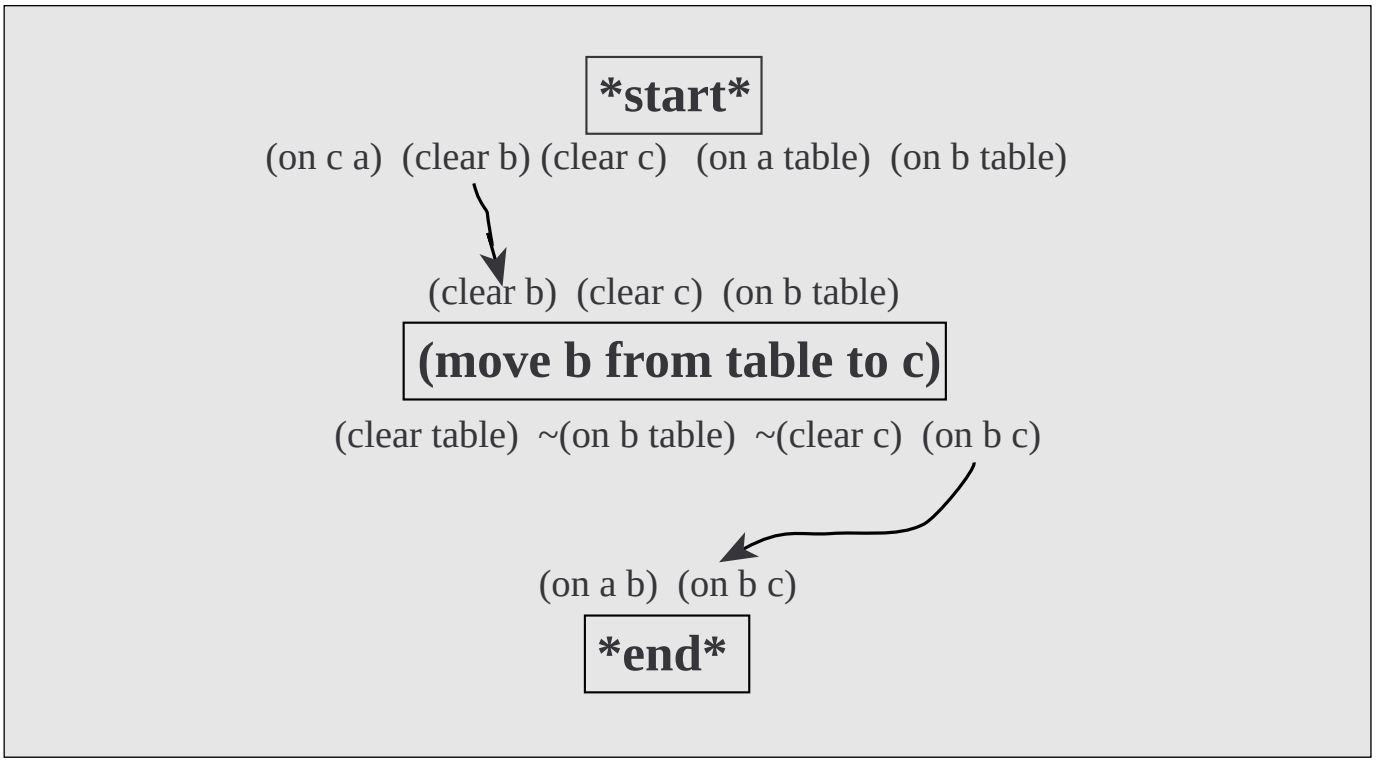

Figure 7. After Adding a Causal Link to Support (clear B), the Plan Has Two Causal Links, and Agenda Is Set to {(clear C) (on B Table) (on A B)}.

search. In practice, the choice can be very important for efficiency, and it is often useful to interleave reasoning about different subgoals. However, 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 nondeterministic choice this time!) an action, A_{add}, which has (on *B C*) as an effect. Suppose that the magic oracle suggests making A_{add} a new instance of a move-B-from-Table-to-C action. A new causal link, A_{add} (on $\stackrel{BC}{\rightarrow}$) A_∞, is added to *L'*, and the agenda is updated. Because there are no threats to the sole link, line 6 makes a recursive call with the arguments depicted in figure 6.

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 I 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 *L,* as illustrated in figure 7, and to shrink agenda slightly.

Suppose that on the third invocation of POP, the planner selects the top-level goal (on *A B*) from 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 *L*, the new action is constrained to precede A_∞, and 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 *O* contains no constraints on their relative ordering. Furthermore, note that move-A-from-Table-to-B negates (clear *B*), but this action means that it threatens the link from A_0 labeled (clear *B*) (figure 8).

To protect against this threat, POP must nondeterministically choose an ordering constraint. In general, there are two possibilities:

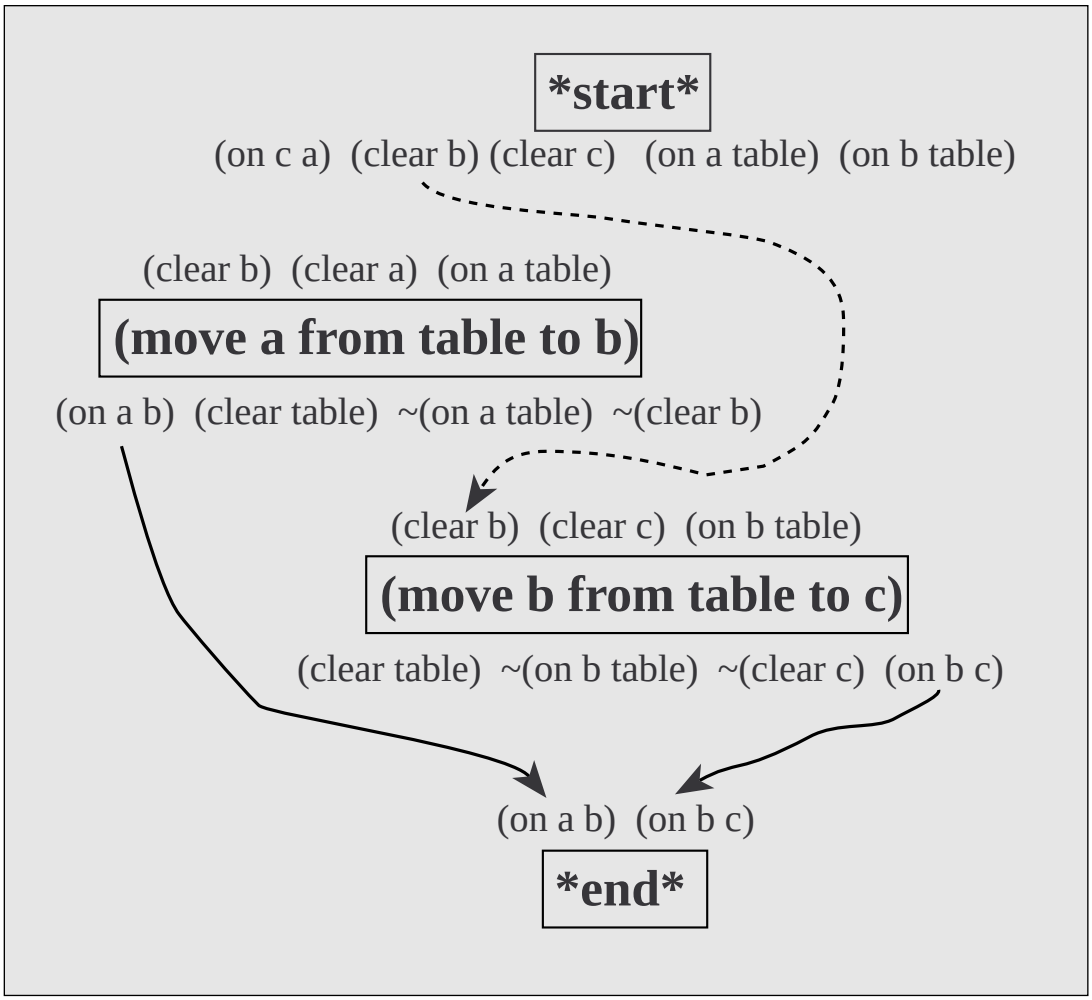

Figure 8. Because the move-A Action Could Possibly Precede the move-B Action, It Threatens the Link Labeled (clear B), as Indicated by the Dashed Line.

Constrain the move-A action after the move-B action, or constrain move-A to precede the *start* action A_0. However, because line 3 of POP assures that every action follows A_0, this last choice would make O inconsistent. Thus, POP orders the threat after the link's consumer, as shown in figure 9.

Because the agenda still contains five entries, much work is left to be done. However, all subsequent decisions follow the same lines of reasoning that I showed earlier, so I omit them here. Eventually, POP returns the plan shown in figure 10. Careful inspection of this figure confirms that no link is threatened. Indeed, the three actions in 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.

Implementation Details To implement POP, one must choose data structures to represent the partial order over actions (O). The operations that the data structure needs to support are adding new constraints to O, testing if O is consistent, determining if A_i can consistently be 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 O and test O for consistency, but this set won't necessarily lead to the greatest efficiency because many more queries are typically performed (that is, in threat detection, as discussed in the following paragraph) than there are true updates. Caching the results of queries (that is, 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

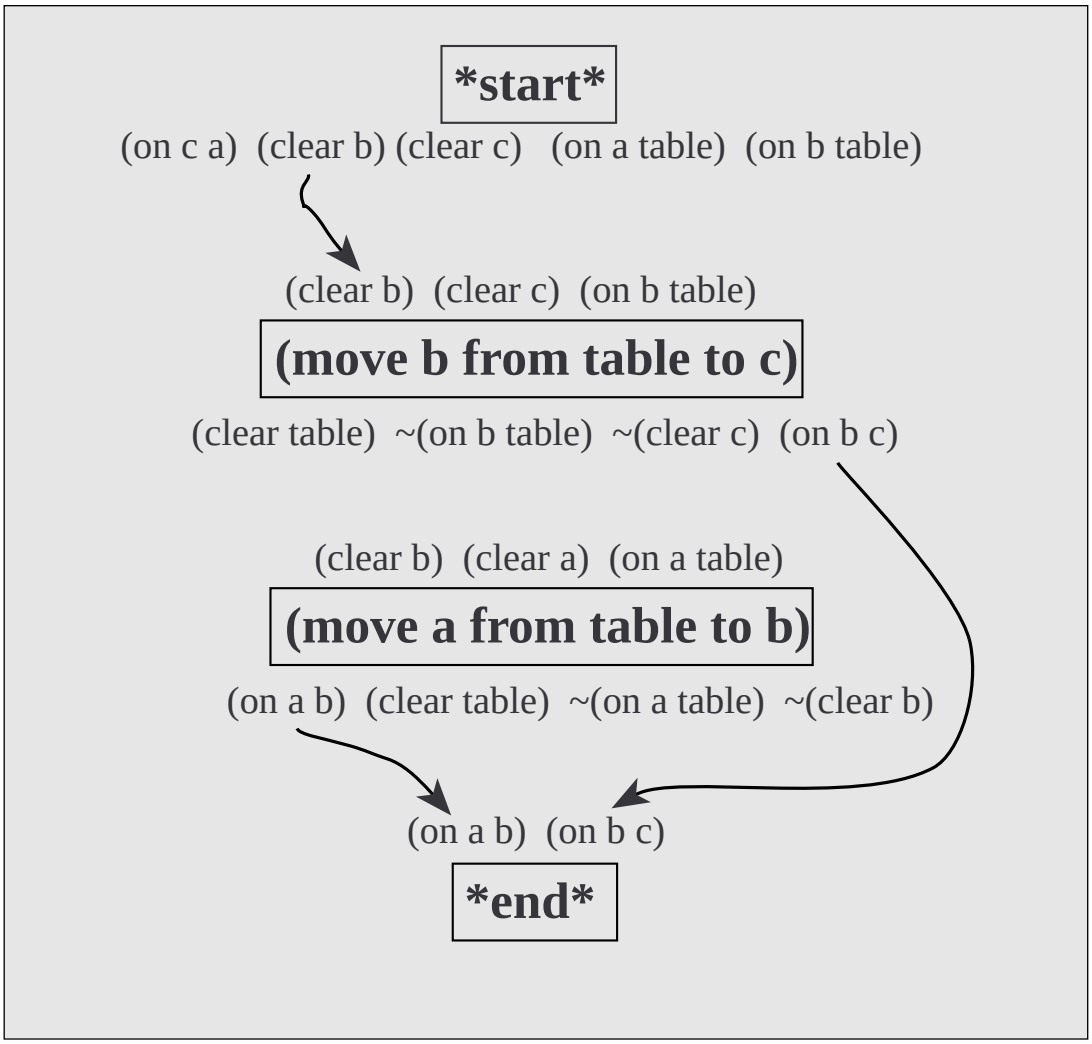

Figure 9. After Promoting the Threatening Action, the Plan's Actions Are Totally Ordered.

closure and $O(a^2)$ space to store it, but queries can be answered quickly (see the Floyd-Warshall algorithm and discussion [Cormen, Leiserson, and Rivest 1991]). Considerable research has focused on this time-space tradeoff and on variants that assume a different interface to the temporal manager (see Williamson and Hanks [1993] for a discussion and pointers).

Another implementation detail concerns the efficiency of testing for threatened causal links. There can be $O(a^2)$ links and, hence, $O(a^3)$ threats. I have found that the most efficient way to handle threats is incrementally: Whenever a new causal link is added to L, all actions in A are tested to see if they threaten it. This action takes $O(a)$ time. Whenever a new action instance is added to A, all links in L are tested to see if they are threatened. This action takes $O(a^2)$ time.

Analysis

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

First, how many times is nondeterministic choose called before a solution is obtained? This parameter determines the exponent n.

Second, how many possibilities need to be considered (by a searching algorithm) at each call to choose? This parameter determines the average branching factor b.

Third, how long does it take to process a given node in the search space (that is, how much processing goes on before the recursive call)? I have written this parameter as the constant c, although it is usually a function of the size of the node being considered.

The cost for each node in the search space c is different for REGWS and POP, but this differ-

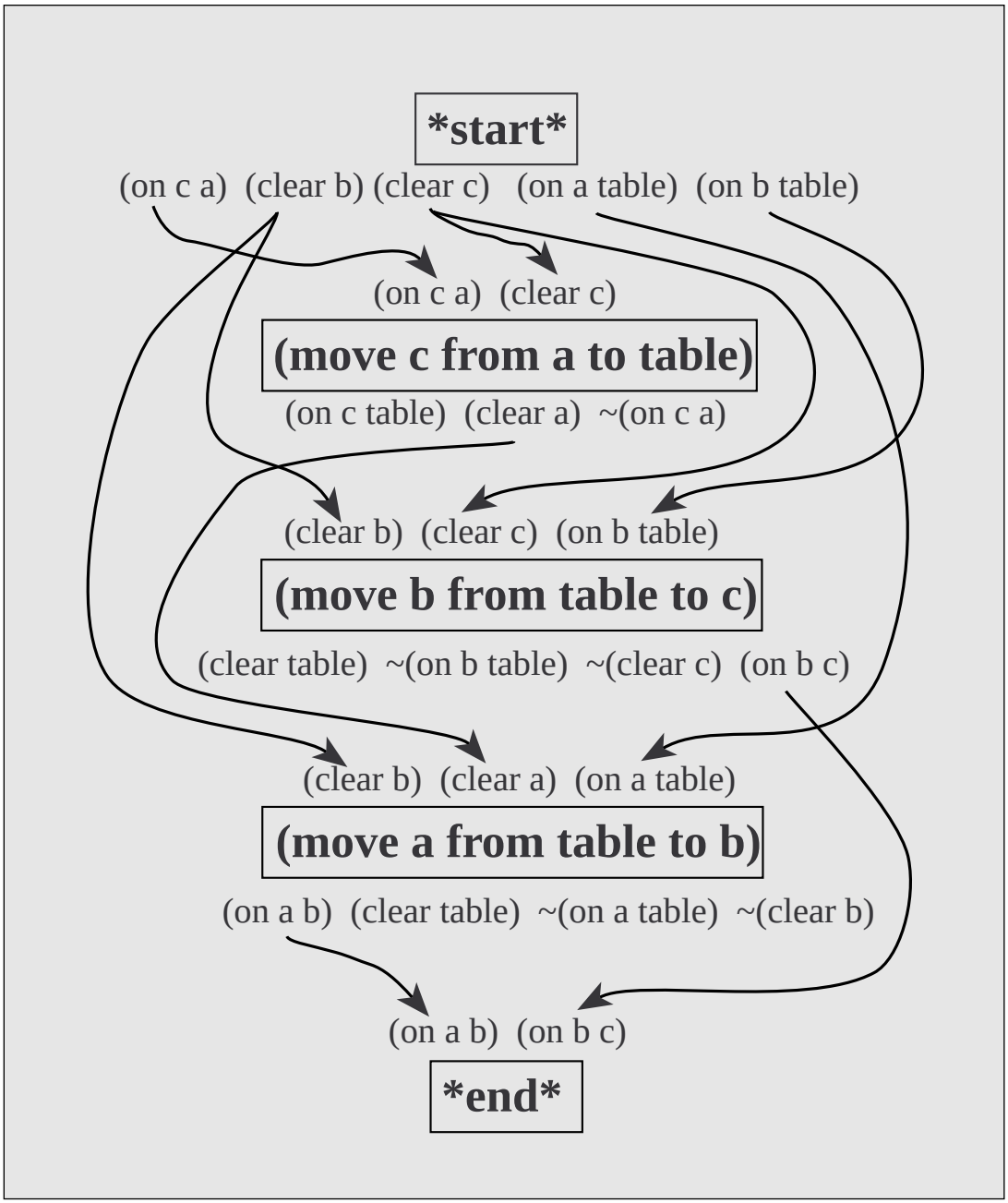

Figure 10. Eventually, This Plan Is Returned as a Solution.

ence doesn't matter much in practice. For the world-state planner, the operations for each node can all be implemented in time proportional to the number of current goals, |curgoals|, and the number and complexity of actions. POP, however, 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 much because the exponential b^n dominates these costs.

The number of nondeterministic calls, n, can vary somewhat. In particular, REGWS makes one call for each action introduced (that is, $n = a$), but POP makes one call for each 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, which is almost always the case. For example, the *start* action often supports many conjuncts. However, the ratio between POP's n and REGWS's n can never be greater than the maximum number of pre-

condition conjuncts for each action, which is typically a small constant (three for the blocks world). In any case, the value of n certainly doesn't suggest that POP will run faster.

However, 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 of the algorithm. Recall that this choice does not require backtracking.

The situation with REGWS is 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 additional branching factor 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, 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 large. 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 (1994, 1993), Minton, Bresina, and Drummond (1991), and 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 possibilities, but see Kambhampati (1993a, 1993b) for a survey of approaches and a fascinating taxonomy of design trade-offs.

Action Schemata with Variables

Because 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 I describe the next step, I want 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 including the set of ordering constraints, 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 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. In addition, 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 these choices are 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 this commitment until later (when some choices might easily be ruled out).

We can accomplish this delay 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 that defines the class of actions that move an arbitrary block, ?b, from an arbitrary prior location, ?x, to any destination, ?y? I 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 answer is to allow codesignation and noncodesignation constraints, which we write as ?x = ?y and ?$x \neq$?y, respectively.[11] To make these ideas concrete, see the definition of figure 11, which defines a move operator that is reasonably general.[12]

Planning with Partially Instantiated Actions

It should be fairly clear that the operator in figure 11 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 were replaced, but other copies were mistakenly left unchanged.

However, representation is one thing, and planning is another. How must the POP plan-

```
(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 11. Variables, Codesignation (=), and Noncodesignation (≠) Constraints Allow Specification of More General Action Schemata.

ning algorithm be modified to handle partially instantiated actions resulting from general operators such as the one in figure 11? I explain the changes in this subsection.

The data-structure–representing plans must include a slot for the set of variable binding (codesignation) constraints. Thus, a plan is now <*A*, *O*, *L*, *B*>, and a problem's null plan has $B = \{\}$.[13]

We also need some way to perform unification. Let *MGU*(*Q*, *R*, *B*) be a function that returns the most general unifier of literals *Q* and *R* with respect to the codesignation constraints in *B*. ⊥ 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 form allows us to treat codesignation constraints in *B* as a conjunction of general unifiers (although, in general, *B* might contain noncodesignation constraints as well, even though *MGU*() cannot generate them).

$$MGU((\text{on } ?x\ B), (\text{on } A\ B)) = \{(?x, A)\}$$
$$MGU((\text{on } ?x\ B), (\text{on } A\ B), \{(?x, C)\}) = \perp$$

For shorthand, we sometimes write *MGU*(*Q*,*R*) rather than explicitly specify an empty set of bindings. Furthermore, we assume that redundantly negated literals are treated in the obvious way. That is, ¬¬*P* unifies with *P*.

When Δ is a logical sentence, the notation $\Delta\backslash B$ (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; that is, $MGU(Q, E, B) \neq \perp$. For example, given the null plan for the Sussman anomaly and the supposition 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 *B* to force the producer's effect to supply the condition required by the consuming action. Continuing the Sussman anomaly example, the constraints {?*b* = *B*, ?*x* = *C*} must be added to *B*.

Also in line 3, when a new action instance is created from an operator in Λ, the planner must ensure that all variables referred to in the action 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 be referred to as ?*b*1, ?*x*1, and ?*y*1.

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

Line 5 (causal link protection) of POP considers every action $A_t \in A$ that might threaten a causal link $A_p \xrightarrow{R} A_C \in L$ and either promotes

it or demotes it. Now that actions have variables in them, the meaning of the phrase "might threaten" is subject to interpretation. If A_t has (not (clear ?y1)) as an effect, could it threaten a link labeled (clear B)? Well, unless B contains a constraint of the form ?y1 ≠ 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, that is, until they unify with no substitution returned. Only at this point does the planner need to decide between adding $A_t < A_p$ or $A_c < A_t$ to O.[14]

One final change is necessary. Line 1 of POP returns the plan if 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 test is necessary to ensure that all threatened links are actually recognized (see the previous paragraph). 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.

Implementation Details

To implement the generalized POP algorithm just described, one must choose data structures for representing the binding constraints, B. The necessary operations include the addition of constraints, the 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 equality constraints, whereas we require noncodesignation constraints as well.

This subsection describes one way to implement these functions. Casual readers might want to skip this discussion and jump directly to Conditional Effects and Disjunction.

One implementation represents B as a list of *varset* structures, defined as follows: 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 B, one first searches through 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, and the const field is copied from whichever of the two found varsets had it set (if any). Next, the union of the two cd-sets is 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 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 ≠ ?y to B, one first searches through B to find the varsets for ?x and ?y. If either symbol is in the cd-set of the other varset, then it fails; otherwise, make a copy of the varsets, augmenting the ncd-sets, and push the two new copies onto B.

These routines might seem inefficient (note that they do not remove old varsets from B, and they make numerous copies; however, they perform well in practice because they enable the planner to explore many plans in parallel (that is, using an arbitrary search technique) with reasonable space efficiency (because the 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.[15]

Conditional Effects and Disjunction

One annoying aspect of the move operator of figure 11 is the restriction that the destination location can't be the table. This restriction means that to describe the 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. The restriction is irritating for both software-engineering and efficiency reasons, but I concentrate on the latter. Note that the existence of two separate movement operators means that the planner has to commit (at algorithm 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.

```
(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 12. Conditional Effects Allow the Move Operator to Be Used When the Source or Destination Location Is the Table (compare with figure 11).

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, I alluded to the fact that we could relax this annoying restriction if the action language allowed conditional effects. Indeed, conditional effects are useful and represent an important step in the journey toward increasingly expressive action-representation languages that I described at the beginning of this article. The basic idea is simple: We allow a special *when* clause in the syntax of action effects. When takes two arguments, an antecedent and a consequent. Both the antecedent and the consequent parts can be filled by a single literal or a conjunction of literals, but their interpretation is different. The *antecedent* refers to the world before the action is executed, and 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 antecedent is true immediately before execution (that is, much like the action's precondition determines if execution itself is legal). Figure 12 illustrates how conditional effects allow a more general definition of move.

Planning with Conditional Effects

Historically, planning with actions that have conditional effects was thought to be an inherently expensive and problematic affair. Thus, it might come as a surprise that conditional effects demand only two small modifications to the planning algorithm presented earlier.

First, recall that line 3 (action selection) of the POP 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 can be used to support the causal link. In this case, line 4 (update goal set) must add the conditional effect's antecedent to agenda. Without conditional effects, POP line 5 (causal link protection) makes the nondeterministic choice between adding $A_t < A_p$ to O' (that is, demotion) or adding $A_c < A_t$ to O' (that is, 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 antecedent to the agenda. For an example of confrontation, see later in the article.

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

Disjunctive Preconditions

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

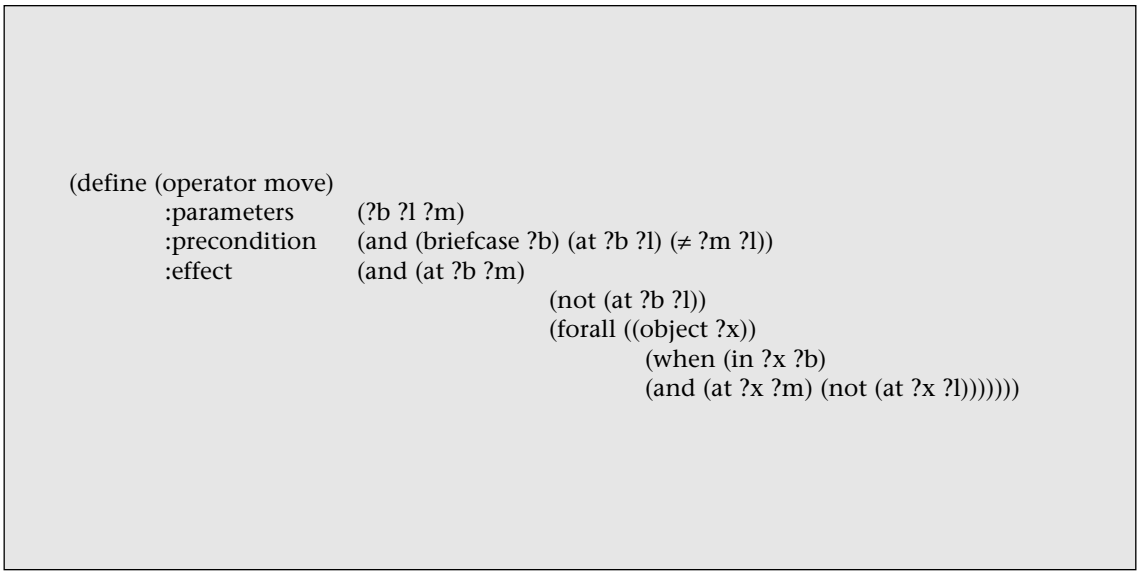

Figure 13. Moving a Briefcase Causes All Objects Inside the Briefcase to Move as Well.

Describing this move requires universally quantified conditional effects. The forall quantifies over all ?*x* that have type object.

selects either Q_1 or Q_2. Whichever disjunct is selected is added back into the agenda.

Note that we are only allowing preconditions, not effects, to be disjunctive. Even though the previous section described conditional effects, (when *P Q*) should not be confused with effects that allow logical implication, that is, (=> *P* Q). In particular, (when *P Q*) is not the same as (or (not *P*) *Q*). The antecedent 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.

Although it is easy to extend the planner to handle disjunctive preconditions, disjunctive effects are much harder. Disjunctive effects only make sense when describing an action that has unpredictable effects. For example, the action of flipping a coin might be described with a disjunctive effect (or (heads ?*x*) (tails ?*x*)). Planning with actions whose effects are only partially known is tricky and warrants more room than I have in this simple introduction. See Kushmerick, Hanks, and Weld (1994); Etzioni et al. (1992); Krebsbach, Olawsky, and Gini (1992); Peot and Smith (1992); Olawsky and Gini (1990); Kaelbling (1988); Schoppers (1987); and Warren (1976).

Universal Quantification

Now we are ready to take the next major step toward more expressive actions. Allowing universal quantification in preconditions allows one to easily describe real-world actions such as the UNIX rmdir command that deletes a directory only if all files inside it have already been deleted. Universally quantified effects allow one to describe actions such as chmod* that 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 clearness 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 the specification of objects such as briefcases, where moving the briefcase causes all objects inside to move as well (figure 13).

Assumptions

To implement a planner, UCPOP, that handles universally quantified preconditions and effects, I need to make a few simplifying assumptions.[16] First, I assume that the world being modeled has a finite, static universe of objects. Furthermore, 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.[17]

Our assumption that the universe is static

means that action effects must not assert type information. If an action were allowed to assert (not (briefcase *B*)), then this assertion would amount to the destruction of an object. Similarly, execution of an effect that said (block *G*001) would create a new block. For now, we don't allow either of these types of effects.

The Universal Base

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

$$\Upsilon(\Delta) = \Delta \text{ if contains no quantifiers}$$
$$\Upsilon(\forall_{t1}\, x\, \Delta(x)) = \Upsilon(\Delta_1) \wedge \ldots \wedge \Upsilon(\Delta_n) \ ,$$

where the Δ_i corresponds to each possible interpretation of $\Delta(x)$ under the universe of discourse, $\{C_1,\ldots,C_n\}$, that is, the possible objects of type $t1$ (Genesereth and Nilsson 1987). 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 the ground sentence 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:

$$\Upsilon(\exists_{t1} y \Delta(y)) = t1(y) \wedge \Upsilon(\Delta(y))$$
$$\Upsilon(\forall_{t1} x\, \exists_{t2} y\, \Delta(x,y)) = t2(y_1) \wedge \Upsilon(\Delta_1) \wedge \ldots \wedge t2(y_n) \wedge \Upsilon(\Delta_n) \ .$$

Once again the Δ_i corresponds 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 (that is, the y_i).[18] All existential quantifiers are eliminated as well, but the remaining free variables (which act as Skolem constants) are implicitly quantified existentially. Because 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. Because each type's universe is assumed to be finite, the universal base is guaranteed to be finite as well. Two more examples illustrate the handling of existential quantification. First, if Δ is

(exists ((briefcase ?*b*))
(forall ((book ?*y*)) (in ?*y* ?*b*))) ,

then the universal base is[19]

(and (briefcase ?*b*) (in moby ?*b*)
(in crime ?*b*) (in dict ?*b*)) .

Second, 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 (?*y*1 and ?*y*2):

(and (book ?*y*1) (in ?*y*1 B1) (book ?*y*2)
(in ?*y*2 B2)) .

Because there are only two briefcases, the Skolem constants ?*y*1 and ?*y*2 exhaust the range of the Skolem function whose domain is the universe of briefcases. Because of the finite, static universe assumption, we can always do this expansion when creating the universal base.

The UCPOP Algorithm

The UCPOP planning algorithm is based on POP (algorithm 3); it is modified to allow action schemata with variables, conditional effects, disjunctive preconditions, and universal quantification. In previous sections, I 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 modification. Before I do, however, it helps to define a few utility functions:

First, if a goal or precondition is a universally quantified sentence, then UCPOP computes the universal base and plans to achieve it instead.

Second, if an effect involves universal quantification, UCPOP does not immediately compute the universal base. Instead, the universal base is generated incrementally, and the effect is used to support causal links.

Third, we need to change the definition of threaten to account for universally quantified effects. Let $A_p \overset{Q}{\rightarrow} A_c$ be a causal link. If a step A_t exists that satisfies the following conditions, then it is a threat to $A_p \overset{Q}{\rightarrow} A_c$. Thus, (1) $A_p < A_t < A_c$ is consistent with *O; (2)* A_t has an effect conjunct R (or has a conditional effect whose consequent has a conjunct); (3) $MGU(Q, \neg R, B)$ does not equal $\perp$; and (4) for all pairs $(u,v) \in MGU(Q,\neg R,B)$, 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 I mentioned that we wanted to consider an action to be a threat only if B necessarily forced the codesignation. With ordinary least commitment variables, this happens when $MGU()$ returns the empty set. However, 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!). Although this substitution would eliminate the need for special treatment of universal variables (in the definition of threats given earlier), it would be inefficient. Because universally quantified effects are expanded into their universal base incrementally, the definition of threat must be altered. A summary of algorithm 4 follows.

Algorithm 4: UCPOP(<A,O,L,B>, agenda, Λ)

1. **Termination:** If agenda is empty, return <A,O,L,B>.

2. **Goal reduction:** Remove a goal <Q,A_c> from agenda.

(a) If Q is a quantified sentence, then post the universal base <$\Upsilon(Q)$,A_c> to agenda. Go to 2.

(b) If Q is a conjunction of Q_i, then post each <Q_i,A_c> to agenda. Go to 2.

(c) If Q is a disjunction of Q_i, then nondeterministically choose one disjunct, Q_k, and post <Q_k,A_c> to agenda. Go to 2.

(d) If Q is a literal, and a link $A_p \xrightarrow{\neg Q} A_c$ exists in L, fail (an impossible plan).

3. **Operator selection:** Nondeterministically choose any existing (from A) or new (instantiated from Λ) action, A_p, with effect conjunct R such that $A_p < A_c$ is consistent with O, and R (note R is a consequent conjunct if the effect is conditional) unifies with given B. If no such choice exists, then fail. Otherwise, let

(a) $L' = L \cup \{A_p \xrightarrow{Q} A_c\}$.

(b) $B' = B \cup \{(u,v) | (u,v) \in MGU(Q,R,B) \wedge u,v$ not universally quantified variables of the effect$\}$.

(c) $O' = O \cup \{A_p < A_c\}$.

4. **Enabling of new actions and effects:** Let $A' = A$ and agenda' = agenda. If $A_p \notin A$, then add A_p to A', add <preconds(A_p)$\backslash MGU(Q,R,B)$,A_p> to agenda', add $\{A_o < A_p < A_\infty\}$ to O, and add non–cd-constraints (A_p) to B'. If the effect is conditional, and it has not already been used to establish a link in L, then add its antecedent to agenda after substituting with $MGU(Q,R,B)$.

5. **Causal link protection:** For each causal link $l = A_i \xrightarrow{P} A_j$ in L and for each action A_t that threatens l nondeterministically, choose one of the following (or if no choice exists, fail):

(a) **Promotion**: If consistent, let $O' = O' \cup \{A_j < A_t\}$.

(b) **Demotion:** If consistent, let $O' = O' \cup \{A_t < A_i\}$.

(c) **Confrontation:** If A_t's threatening effect is conditional with antecedent S and consequent R, then add <$\neg S \backslash MGU(P,\neg R)$,$A_t$> to agenda'.

6. **Recursive invocation:** If B is inconsistent, then fail; else call UCPOP(<A',O',L',B'>, agenda', Λ).

Although I do 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.

Confrontation Example

To see a concrete example of UCPOP in action, recall the move operator defined in figure 13 that 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 14.

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

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

Figure 14. This Action Removes an Item from the Briefcase.

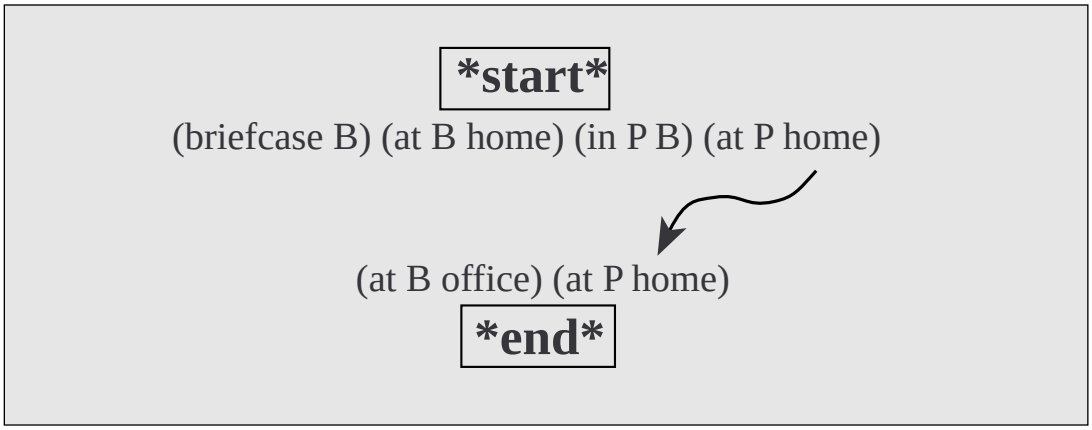

Figure 15. One Subgoal Is Easy to Support.

Suppose that there is just one briefcase, *B,* that is at home with a paycheck, *P*, inside it, as codified by the following 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 <(and (at *B* office) (at *P* home)), A_∞>,[20] and Λ = the move and take-out operators. Because agenda is nonempty, the goal is removed from the agenda, recognized as a conjunction, and reduced to two literals that are both put back on the agenda (UCPOP line 2b). UCPOP line 2 is now executed again, and the goal (at *P* home) is removed from agenda; because it is a literal, control proceeds to line 3 (operator selection). There are two ways to support this goal: (1) create a new instance of a move action or (2) link to the initial conditions (that is, the existing step A_0). Suppose that UCPOP makes the correct nondeterministic choice and links to the initial state. Because the one link isn't threatened, UCPOP calls itself recursively with the plan shown in figure 15.

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 because no existing steps have effects that match 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 have as a subgoal getting *B* inside the other briefcase. Because 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. Because 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 recur-

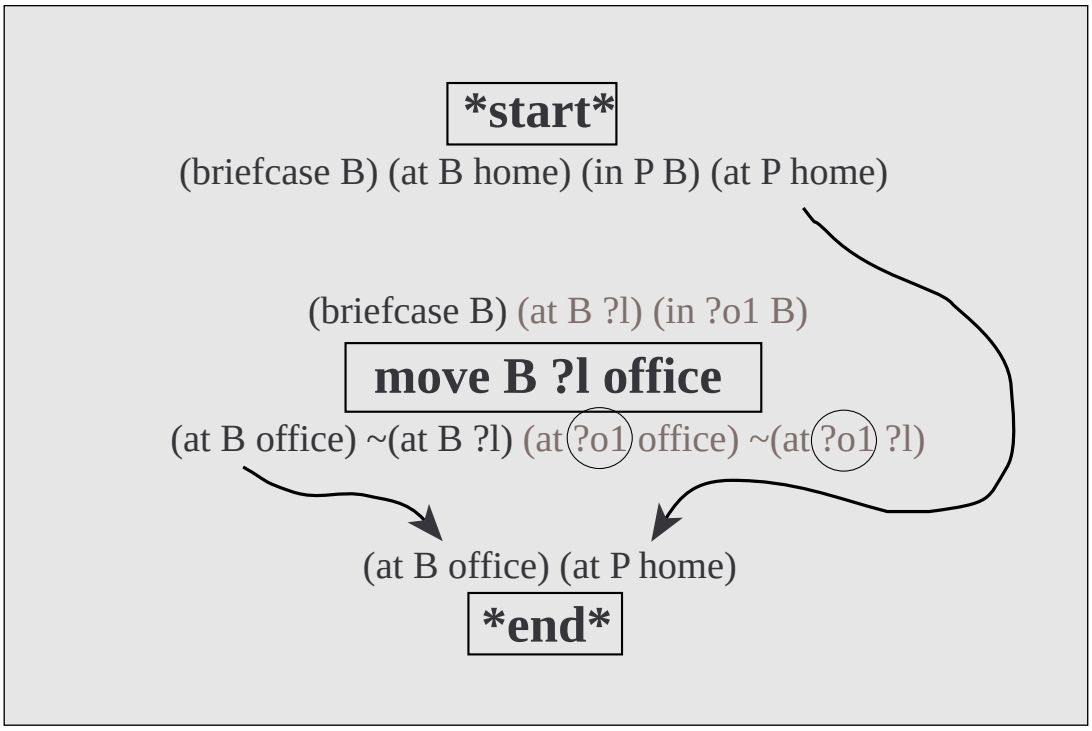

Figure 16. A New Move Step Supports the Second Goal without Threatening the First Link.

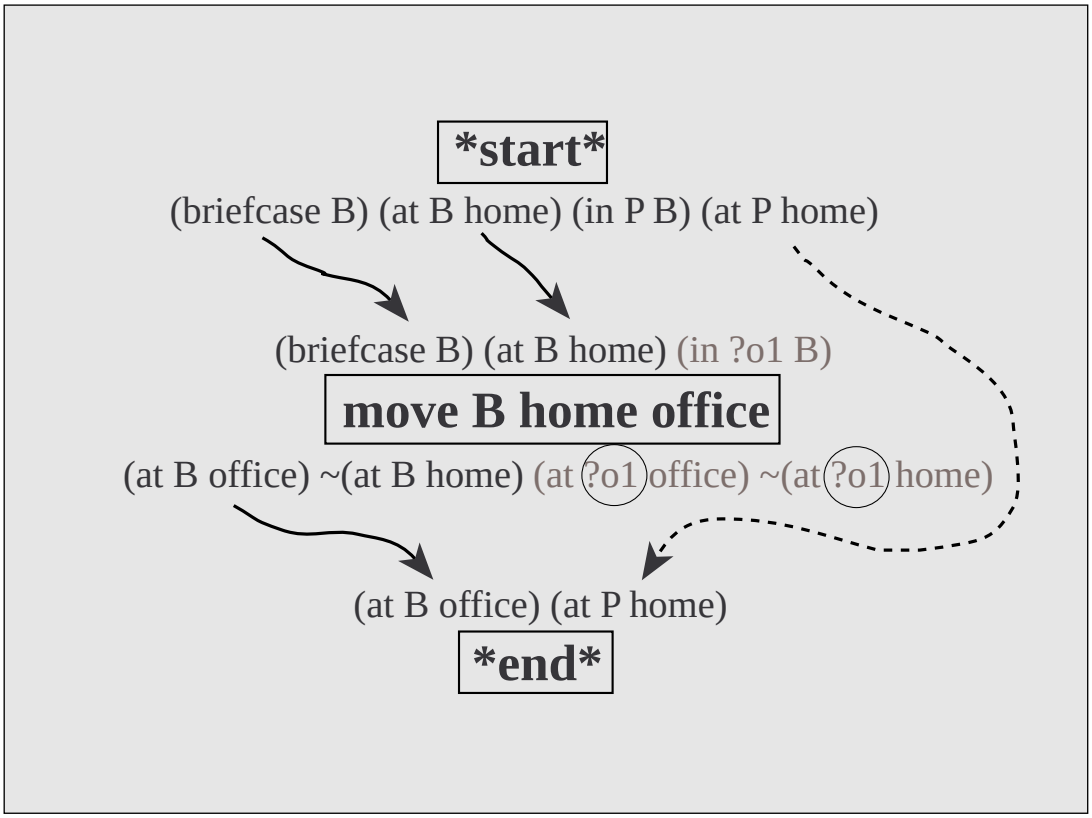

Figure 17. Now That Codesignation Constraints Bind ?l to Home, the Move Step Threatens an Existing Link.

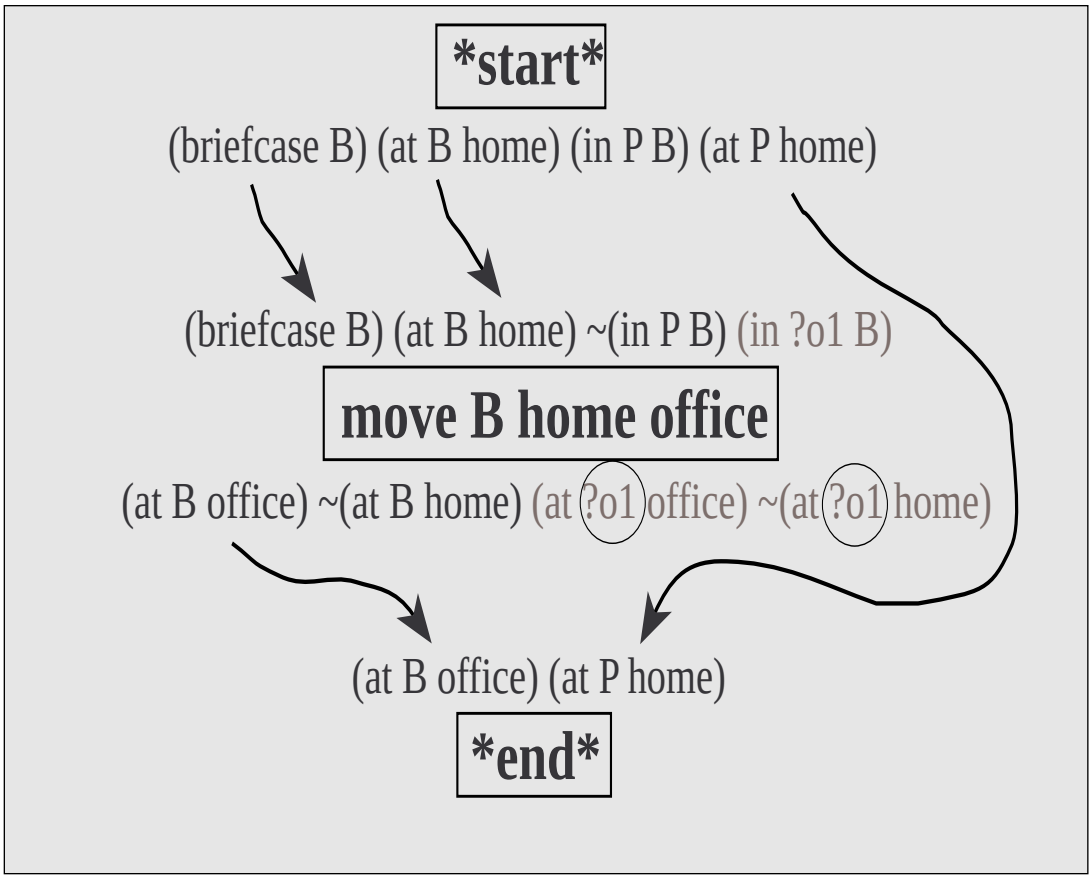

Figure 18. After Confronting the Threat.

sively with the plan shown in figure 16. Note that some of the effects of move are shown in gray rather than black, signifying that they are conditional. Furthermore, note that the variable ?*o*1 is surrounded by a circle to denote that it is universally quantified.

The conjunctive goal—(and (briefcase *B*) (at *B* ?*l*))—gets reduced into its component literals that get chosen in turn. In both cases, it is possible to link them to the initial state in the same way that was illustrated earlier. 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 *B* causes move to threaten the link labeled (at *P* home), as signified by the dashed line in figure 17.

Previously, the link wasn't threatened by move because *MSU*((at ?*o*1 ?*l*)), (at *P* home),*B*) unified with a complex unifier—{(?*l*, home),(?*o*1, *P*)}. Although the second binding pair contains a universally quantified variable ?*o*1, the first pair does not; so, the definition of threat is unsatisfied. However, after the *B* is extended to constrain ?*l* to the value home, the most general unifier consists solely of (?*o*1, *P*). Now the threat is real, as figure 17 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_o or after the *end* action A_∞. Fortunately, the threatening effect is conditional; so, confrontation is a viable technique. 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 ?*o*1 are crucially important when posting this new subgoal. UCPOP does not need to 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 18.

Satisfying the new subgoal requires instan-

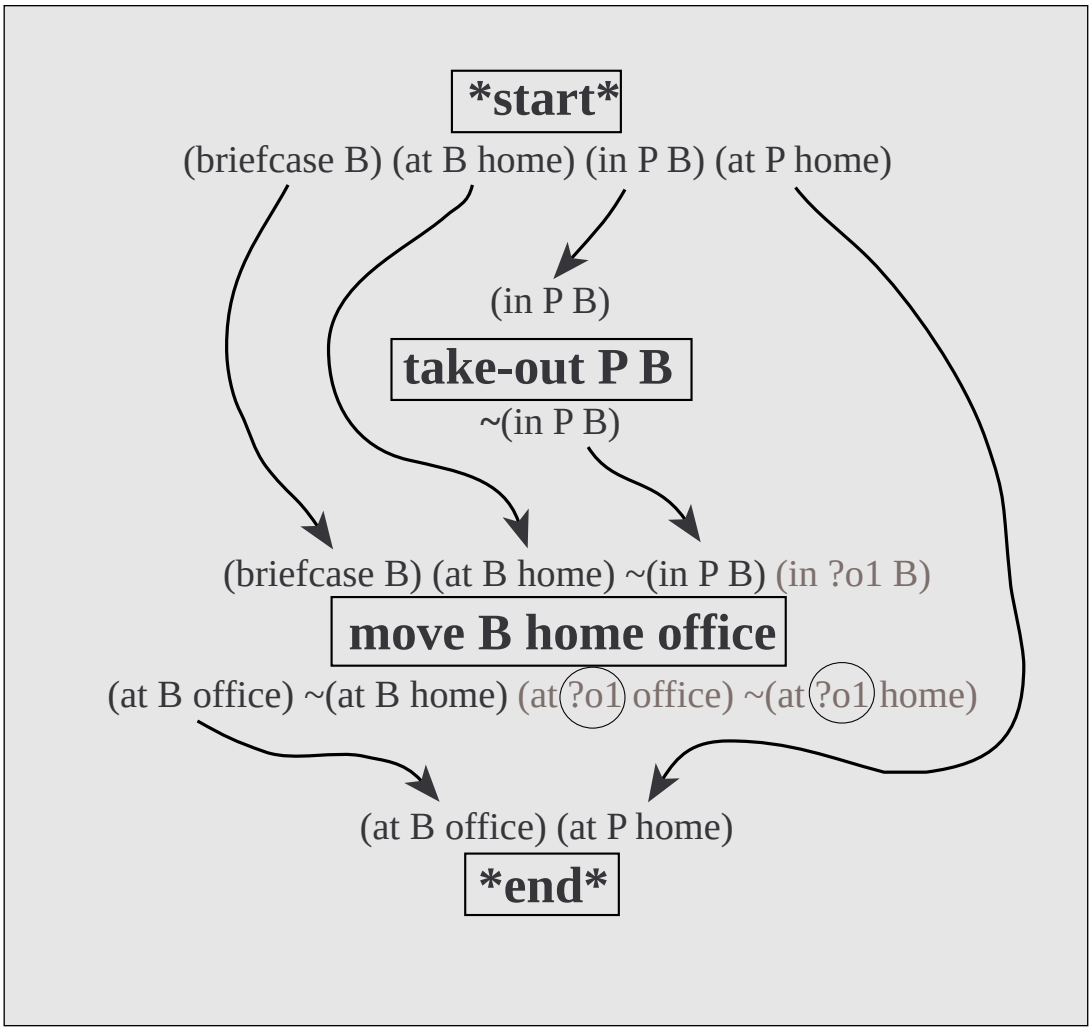

Figure 19. Final Plan.

tiating and adding a take-out step to A, which adds another subgoal to the agenda. However, the goal of (in *P B*) is easily satisfied by the initial conditions; so, UCPOP quickly returns the plan shown in figure 19 as its solution to this planning problem.

Quantification Example

Although the example in the previous subsection 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. We also did not link to a universally quantified effect to demonstrate the incremental expansion of the universal base. To demonstrate these features, let's consider another primitive operator and another problem. The new action schema allows one to add items to the briefcase (figure 20).

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 21.

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 agen-

```
(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 20. What Good Is a Briefcase If We Can't Put Things into It?

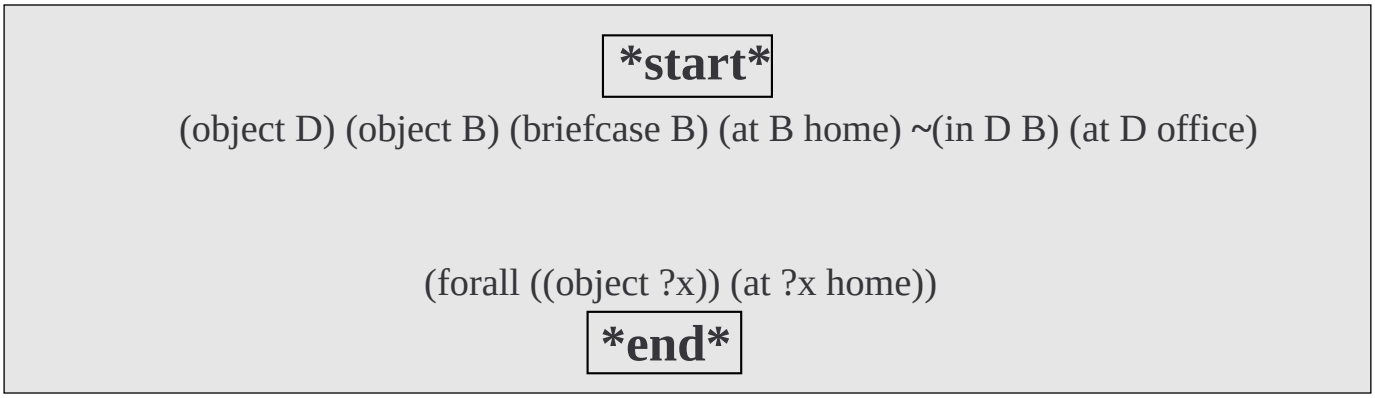

Figure 21. Dummy Plan Representing Problem.

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 that are not explicitly listed as true are presumed false.

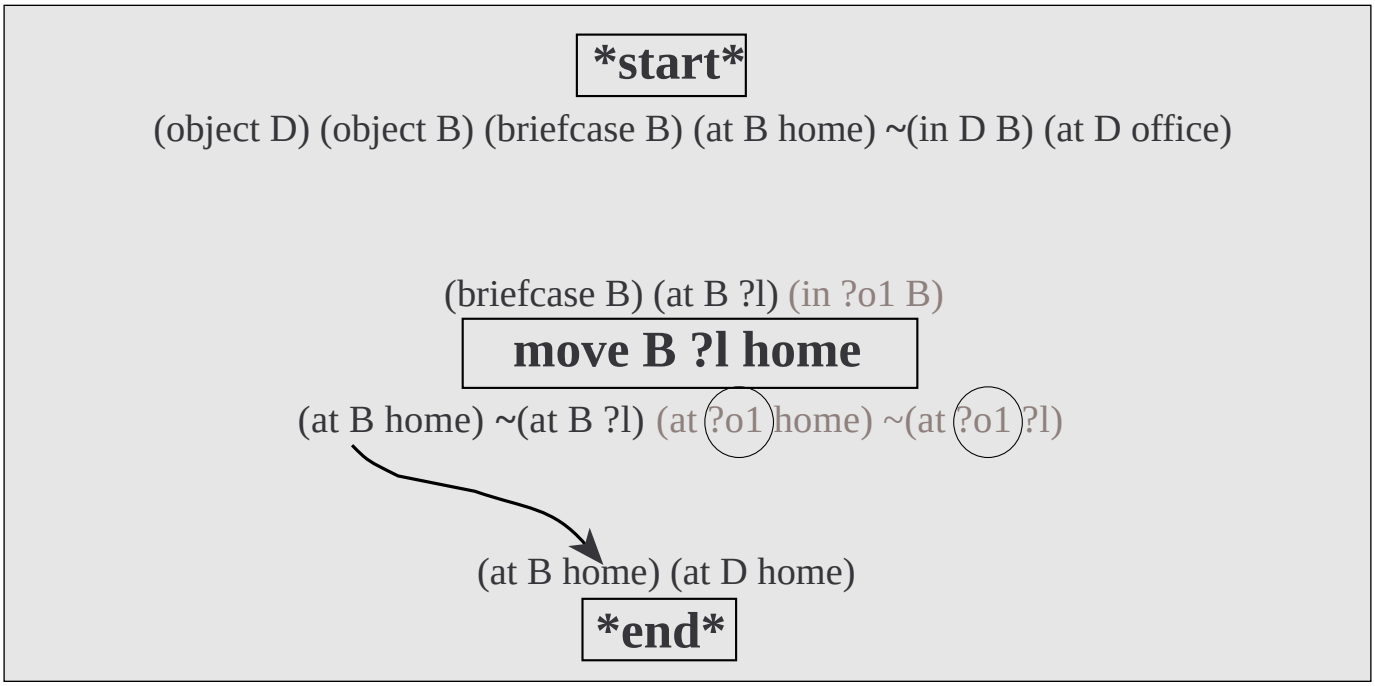

Figure 22. After Supporting First Conjunct.

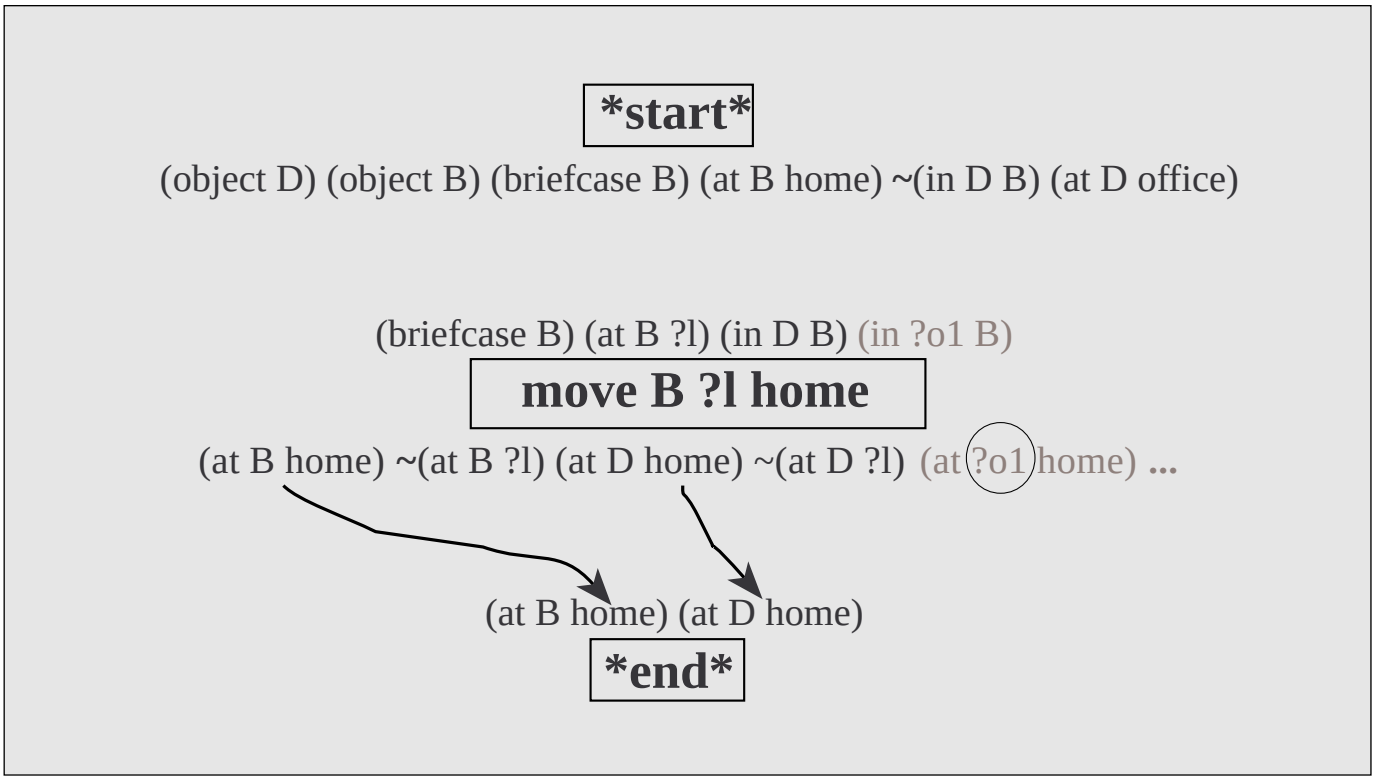

Figure 23. After Incrementally Expanding Part of the Universal Base of the Universally Quantified Effect.

da 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 *A*, and its preconditions are added to the agenda. When the recursive call occurs (UCPOP line 6), the updated plan is shown, as in figure 22.

On the next iteration, suppose that *Q* = (at *D* home). Because *Q* is a literal, control goes to line 3. Suppose that UCPOP wisely (that is, 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 ?*o*1 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 antecedent of the conditional effect to agenda. The resulting plan is shown in figure 23. Note that although I have drawn (at *D* home) and (at *D* ?*l*) as instantiated effects of move, they aren't actually explicitly added to the data structures. There's no point as long as the universally quantified version is there; so, perhaps the catch phrase "incrementally generating the universal base for effects" is misleading. You might prefer to think of it as not being generated at all.

Now that we've covered most of the interesting stuff, I'll fast forward to the end (figure 24).

Quantification over Dynamic Universes

To this point, 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, I explain how to handle *dynamic universes*, that is, domains whose action effects can create new objects or delete existing ones.[21] I address two independent questions in turn: (1) how object creation and destruction should be

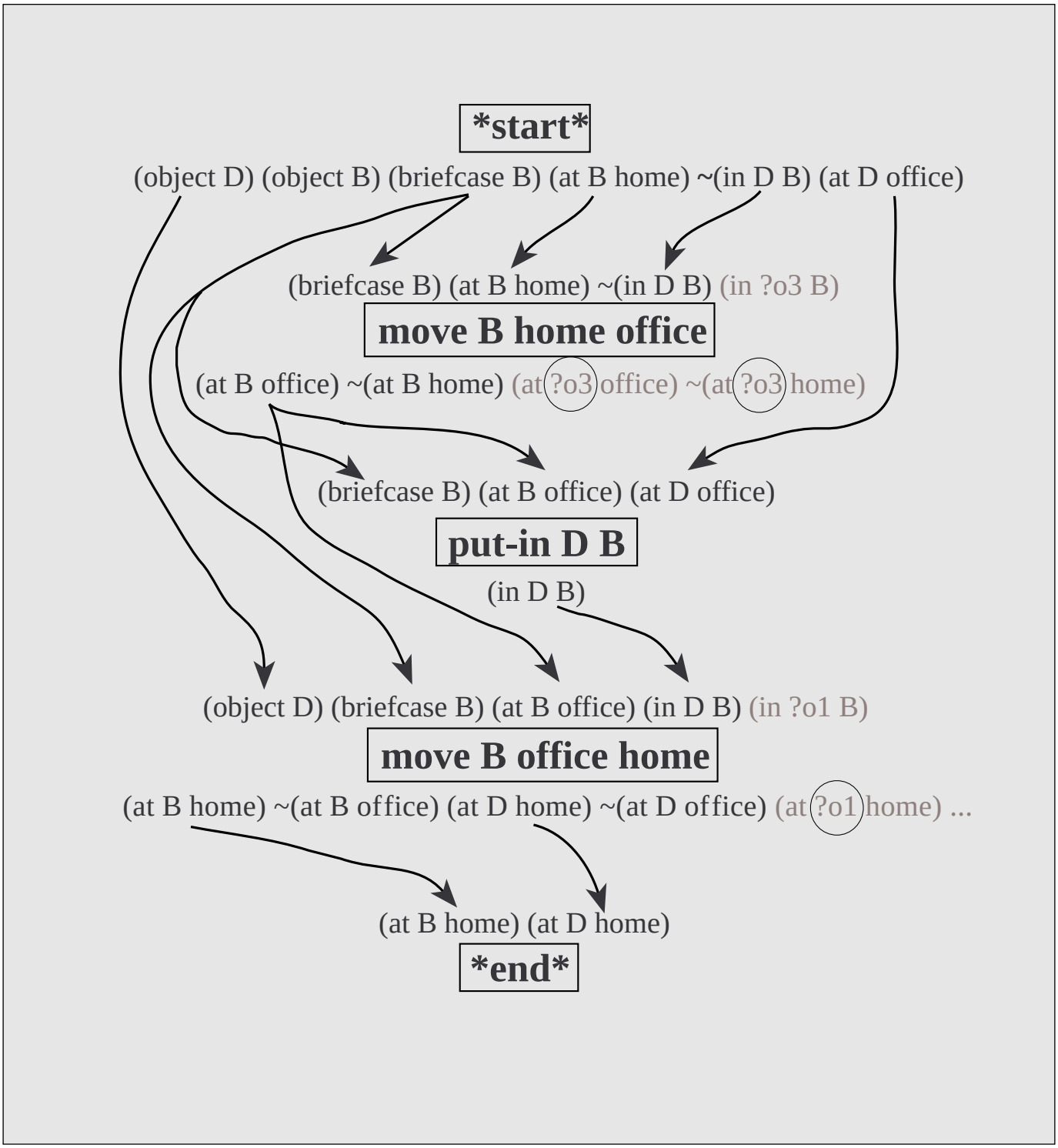

Figure 24. Final Plan.

represented (syntactically) in the action language and (2) how the planner should 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 the universe of Δ is (forall ((book ?*y*)) (in ?*y B*)). Recall that if the universe of books is static, it then 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 because there will be no way to achieve the (not (book ... subgoals.[22]

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 previous example, UCPOP determines that moby and dict are the only possible books in this manner: by matching against the initial state. 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. Because 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 the forall goal is reconsidered, and the universal base is incrementally updated.[23] This update gets tricky if the goal expression involves nested universal and existential quantifiers because the incremental

David Chapman's "Planning for Conjunctive Goals"

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, 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 MTC can be used for both plan verification and plan generation; to demonstrate the latter, he implemented a sound and complete planner called TWEAK.

Chapman also proved that evaluating MTC is NP hard when actions contain conditional effects. Because TWEAK evaluated MTC repeatedly in its innermost loop, Chapman (and other researchers) speculated that least commitment planning would not scale up to expressive action languages, for example, 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 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 algorithm pushes the complexity of evaluating MTC into the size of the search space. For an in-depth discussion of this subject and other aspects of Chapman's results, see Kambhampati and Nau (1993); for more information on least commitment planning with conditional effects, see Collins and Pryor (1992), Penberthy and Weld (1992), and Pednault (1991).

expansion must create the appropriate number of new Skolem constants.

Implementation

Common Lisp source code for the planner is available for nonprofit use through anonymous FTP (on june.cs.washington.edu as the compressed file ftp/pub/ai/ucpop.tar.Z [use binary mode for transfer]). The code is simple enough for classroom use but efficient (that is, it takes about 2 to 20 microseconds to explore and refine a partial plan on a SPARCIPX). In addition to the features described in this section, UCPOP 2.0 provides the following enhancements: (1) declarative specification of control rules that guide the nondeterministic search, (2) a graphic plan-space browser written in CLIM for portability, (3) domain axioms, (4) predicates that call Lisp code when used in action preconditions (useful when the domain theory involves arithmetic, and so on), (5) a set of domain theories (including those used in this article and many more) for experimentation, and (6) a users' manual (Barrett et al. 1993).

Advanced Topics

The discussion in this article was restricted to goals of attainment. Although I explained how to handle goal descriptions involving disjunction and universal quantification (not just conjunction as in STRIPS), I 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 want to specify that a household robot should never set the house on fire and that a software robot (that is, a softbot [Etzioni, Lesh, and Segal, 1993b; Etzioni and Segal 1992]) shouldn't delete valuable files. One class of behavioral constraints, called *maintenance goals*, can be implemented easily on top of by an extension of the causal link threat-detection mechanism; see Weld and Etzioni (1994) and Etzioni et al. (1992). 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 P-space complete if actions can have more than two conjuncts in their preconditions (Bylander 1991). In some cases, planning is undecidable (Erol, Nau, and Subrahmanian 1992). As a result, we can't expect any of the planners described in this article to perform quickly on all problems all the time. In fact, to achieve reasonable performance much of the time, it is usually necessary to add domain-dependent control knowledge, and in addition, it is often necessary to sacrifice completeness. Because the nondeterministic choose function is implemented with search, adding domain knowledge amounts to using an aggressive heuristic search algorithm rather than breadth-first or iterative deepening depth-first search. A simple way to encode domain information is to provide a ranking function (that is, 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*, that is, 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 rulelike form is much easier because individual rules refer to local decisions, and there is no need to weight the pieces as required when computing a single metric rank. These ideas were first explored in the SOAR system (Laird, Newell, and Rosenbloom 1987) and refined in the PRODIGY planner (Minton et al. 1989b); they were also 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, 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 (Currie and Tate 1991; Yang 1990; Charniak and McDermott, 1984; Tate 1977).

Another form of search control exploits the notion of resources. SIPE (Wilkins, 1990, 1988a) 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. The restriction to one link for each precondition can be seen as a violation of least commitment because it demands that the planner respond to threats even in cases where 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, 1992b) 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 (1989) describes the ADL language, which is slightly more expressive than that handled by UCPOP; he discusses the theory behind regression planning for this language in Pednault (1988), but no one has implemented a planner for the full language.[24] Many other extensions have been implemented, however, including incomplete information, execution, and sensing operations (Golden, Etzioni, and Weld 1994; Etzioni et al. 1992; Peot and Smith 1992); probabilistic planning (Draper, Hanks, and Weld 1994; Kushmerick, Hanks, and Weld 1993); 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 (Wellman 1993; Haddawy and Hanks 1992), 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 I can't describe it here. See Allen, Hendler, and Tate (1990) for the tip of the iceberg.

Acknowledgments

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, Ramesh Patil, Kari Pulli, Ying Sun, Austin Tate, and Mike Williamson for helpful comments. This research was funded in part by the Office of Naval Research, grant 90-J-1904, and the National Science Foundation, grant IRI-8957302.

Notes

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

2. The acronym STRIPS stands for Stanford Research Institute problem solver, which is a famous and influential planner built in the 1970s to control an unstable mobile robot known affectionately as SHAKEY (Fikes and Nilsson 1971).

3. The etymology of the name is a bit puzzling because the problem was discovered at the Massachusetts Institute of Technology in 1973 by Allen Brown, who noticed that the HACKER problem solver had problems dealing with it. Because 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 (1975) INTERPLAN system used more sophisticated reasoning about goal interactions to find an optimal solution, and Sacerdoti's (1975) NOAH planner introduced a more flexible representation to sidestep the problem. Because the planners described in this article 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: (1) to achieve (on *A B*) and (2) 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*. Then we accidentally wind up with *A* on *B* when *B* is still on the table. Of course, one can't get *B* on without taking *B* off; so, trying to solve the first subgoal first appears to be a mistake. However, if we try to achieve (on *B C*) first, then we have a similar problem: *B* is on *C*, but *A* is still buried at the bottom of the stack. No matter which order is tried, the subgoals interfere with each other. Humans seem to use divide and conquer, so why can't computers? In fact, they can, as I show in the section on plan-space search.

4. It's illegal for an action's effect to include both an atomic formula and its negation because it would lead to an undefined result.

5. Means-end 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 et al., 1989b; Minton 1988). Unfortunately, GPS-like planners are incomplete (for example, they cannot solve the Sussman anomaly), which complicates analysis and comparison to the algorithms in this article. Future work is needed to investigate the benefits, if any, of the GPS approach.

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

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

8. The rationale behind the names stems from the fact that demotion moves the threat lower in the temporal ordering, but promotion moves it higher.

9. Actually, we adopt the convention that every proposition that is not explicitly specified to be true in the initial state is assumed to be false. This convention is called the closed-world assumption (CWA).

10. The POP planner is similar to McAllester's SNLIP algorithm (McAllester and Rosenblitt 1991), which is an improved formalization of Chapman's (1987) TWEAK planner. The difference between SNLP and POP concerns the definition of threat. SNLP treats A_t as a threat to a link $A_p \overset{Q}{\rightarrow} A_c$ when A_t has Q as an effect as well as when it has $\neg Q$ as an effect. Although this might seem counterintuitive (what does it matter if Q is asserted twice?), the SNLP definition leads to a property, called *systematicity,* that 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.

11. A more elaborate approach would incorporate ideas from programming language–type systems.

12. Figure 11's definition of move is restricted so that the block can't be moved to the table. This restriction 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, but blocks can have only one block on top of them. Thus, moving a block onto another must negate (clear ?y), but this negation mustn't happen if ?y = Table. Although 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 later).

13. *B* stands for binding.

14. 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) and Kambhampati (1993b), this approach has the advantage of both simplicity and efficiency.

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

16. These assumptions can be relaxed, but this discussion is beyond the scope of this article.

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

18. 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.

19. Because ?*b* is free, it is implicitly existentially quantified. Of course, because there are only two briefcases, (briefcase ?*b*) is equivalent to saying that the books have to be in *B*1 or *B*2. Hence, in this case, Δ and $\Upsilon(\Delta)$ both specify an (implicit) disjunction. As a result, although this goal will be legal for UCPOP, we cannot allow it as an action effect for the reasons described in Disjunctive Preconditions.

20. Recall that we often refer to the agenda as if it contained 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.

21. Infinite universes of discourse and situations in which the agent has only incomplete information can also be handled, but these are considerably more difficult. See Golden, Etzioni, and Weld (1994) for more information.

22. Although my 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. I claim that this issue of plan quality and harmful side effects is best treated separately because it crops up in many situations other than universally quantified goals. See Weld and Etzioni (1994), Lansky (1993), Pollack (1992), and Wilkins (1988b) for a discussion of this topic.

23. This technique was first used in the GORDIUS planner (Simmons 1992).

24. McDermott's (1991) PEDESTAL planner is a total order planner that roughly handles the same subset of ADL as UCPOP.

References

Allen, J.; Hendler, J.; and Tate, A., eds. 1990. *Readings in Planning*. San Mateo, Calif.: Morgan Kaufmann.

Ambros-Ingerson, J., and Steel, S. 1988. Integrating Planning, Execution, and Monitoring. In Proceedings of the Seventh National Conference on Artificial Intelligence, 735–740. Menlo Park, Calif.: American Association for Artificial Intelligence.

Barrett, A., and Weld, D. 1994. Partial Order Planning: Evaluating Possible Efficiency Gains. *Artificial Intelligence* 67(1): 71–112.

Barrett A., and Weld, D. 1993. Characterizing Subgoal Interactions for Planning. In Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, 1388–1393. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Barrett, A.; Golden, K.; Penberthy, J. S.; and Weld, D. 1993. UCPOP User's Manual, Version 2.0, Technical Report, 93-09-06, Dept. of Computer Science and Engineering, Univ. of Washington.

Bylander, T. 1991. Complexity Results for Planning.

In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, 274–279. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Chapman, D. 1987. Planning for Conjunctive Goals. *Artificial Intelligence* 32(3): 333–377.

Charniak, E., and McDermott, E. 1984. *Introduction to Artificial Intelligence*. Reading, Mass.: Addison-Wesley.

Collins, G., and Pryor, L. 1992. Achieving the Functionality of Filter Conditions in a Partial Order Planner. In Proceedings of the Tenth National Conference on Artificial Intelligence, 375–380. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Cormen, T.; Leiserson, C.; and Rivest, R. 1991. *Introduction to Algorithms*. Cambridge, Mass.: MIT Press.

Currie, K., and Tate, A. 1991. O-PLAN: The Open Planning Architecture. *Artificial Intelligence* 52(1): 49–86.

Draper, D.; Hanks, S.; and Weld, D. 1994. Probabilistic Planning with Information Gathering and Contingent Execution. In Proceedings of the Second International Conference on AI Planning Systems, 31–36. Menlo Park, Calif.: American Association for Artificial Intelligence.

Drummond, M. 1989. Situated Control Rules. In *Proceedings of the First International Conference on Knowledge Representation and Reasoning,* 103–113. San Mateo, Calif.: Morgan Kaufmann.

Erol, K.; Nau, D.; and Subrahmanian, V. 1992. When Is Planning Decidable? In *Proceedings of the First International Conference on AI Planning Systems,* 222–227. San Mateo, Calif.: Morgan Kaufmann.

Etzioni, O. 1933a. Acquiring Search-Control Knowledge via Static Analysis. *Artificial Intelligence* 62(2): 255–302.

Etzioni, O. 1993b. A Structural Theory of Explanation-Based Learning. *Artificial Intelligence* 60(1): 93–140.

Etzioni, O. 1993c. Intelligence without Robots: A Reply to Brooks. *AI Magazine* 14(4): 7–13.

Etzioni, O., and Segal, R. 1992. Softbots as Testbeds for Machine Learning. Presented at the AAAI Spring Symposium on Knowledge Assimilation, March 25–27, Stanford, Calif.

Etzioni, O.; Lesh, N.; and Segal, R. 1993. Building Softbots for UNIX, Preliminary Technical Report, 93-09-01, Dept. of Computer Science and Engineering, Univ. of Washington.

Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An Approach to Planning with Incomplete Information. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning,* 115–125. San Mateo, Calif.: Morgan Kaufmann.

Fikes, R., and Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2(3–4): 189–208.

Genesereth, M., and Nilsson, N. 1987. *Logical Foundations of Artificial Intelligence*. San Mateo, Calif.: Morgan Kaufmann.

Golden, K.; Etzioni, O.; and Weld, D. 1994. Omnipotence without Omniscience: Sensor Management in Planning. In Proceedings of the Twelfth National Conference on Artificial Intelligence, 1048–1054. Menlo Park, Calif.: American Association for Artificial Intelligence.

Haddawy, P., and Hanks, S. 1992. Representations for Decision-Theoretic Planning: Utility Functions for Deadline Goals. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning,* 71–82. San Mateo, Calif.: Morgan Kaufmann.

Hammond, K. 1990. Explaining and Repairing Plans That Fail. *Artificial Intelligence* 45:173–228.

Hanks, S. 1990. Practical Temporal Projection. In Proceedings of the Eighth National Conference on Artificial Intelligence, 158–163. Menlo Park, Calif.: American Association for Artificial Intelligence.

Hanks, S., and McDermott, D. 1994. Modeling a Dynamic and Uncertain World I: Symbolic and Probabilistic Reasoning about Change. *Artificial Intelligence* 66(1): 1–56.

Hanks, S., and Weld, D. 1992. Systematic Adaptation for Case-Based Planning. In *Proceedings of the First International Conference on AI Planning Systems,* 96–105. San Mateo, Calif.: Morgan Kaufmann.

Kaelbling, L. P. 1988. Goals as Parallel Program Specifications. In Proceedings of the Seventh National Conference on Artificial Intelligence, pages???. Menlo Park, Calif.: American Association for Artificial Intelligence.

Kambhampati, S. 1993a. On the Utility of Systematicity: Understanding the Trade-Offs between Redundancy and Commitment in Partial-Order Planning. In Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, 1380–1385. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Kambhampati, S. 1993b. Planning as Refinement Search: A Unified Framework for Comparative Analysis of Search Space Size and Performance, Technical Report, TR-93-004, Dept. of Computer Science and Engineering, Arizona State Univ.

Kambhampati, S. 1992a. Characterizing Multi-Contributor Causal Structures for Planning. In *Proceedings of the First International Conference on AI Planning Systems,* 116–125. San Mateo, Calif.: Morgan Kaufmann.

Kambhampati, S. 1992b. Multi-Contributor Causal Structures for Planning: A Formalization and Evaluation, Technical Report, CS-TR-92-019, Dept. of Computer Science and Engineering, Arizona State Univ.

Kambhampati, S., and Hendler, J. 1992. A Validation Structure Based Theory of Plan Modification and Reuse. *Artificial Intelligence* 55:193–258.

Kambhampati, S., and Nau, D. S. 1993. On the Nature and Role of Modal Truth Criteria in Planning, Technical Report, ISR-TR-93-30, Inst. for Systems Research, Univ. of Maryland.

Knoblock, C. 1990. Learning Abstraction Hierar-

chies for Problem Solving. In Proceedings of the Eighth National Conference on Artificial Intelligence, 923–928. Menlo Park, Calif.: American Association for Artificial Intelligence.

Korf, R. 1992. Linear-Space Best-First Search: Summary of Results. In Proceedings of the Tenth National Conference on Artificial Intelligence, 533–538. Menlo Park, Calif.: American Association for Artificial Intelligence.

Korf, R. 1988. Search: A Survey of Recent Results. In *Exploring Artificial Intelligence*, ed. H. Shrobe, 197–237. San Mateo, Calif.: Morgan Kaufmann.

Krebsbach, K.; Olawsky, D.; and Gini, M. 1992. An Empirical Study of Sensing and Defaulting in Planning. In *Proceedings of the First International Conference on AI Planning Systems,* 136–144. San Mateo, Calif.: Morgan Kaufmann.

Kushmerick, N.; Hanks, S.; and Weld, D. 1994. An Algorithm for Probabilistic Least-Commitment Planning. In Proceedings of the Twelfth National Conference on Artificial Intelligence, 1073–1078. Menlo Park, Calif.: American Association for Artificial Intelligence.

Laird, J.; Newell, A., and Rosenbloom, P. 1987. SOAR: An Architecture for General Intelligence. *Artificial Intelligence* 33(1): 1–64.

Lansky, A., ed. 1993. Foundations of Automatic Planning: The Classical Approach and Beyond, Technical Report SS-93-03, AAAI Press, Menlo Park, Calif.

Lansky, A. 1988. Localized Event-Based Reasoning for Multiagent Domains. *Computational Intelligence* 4(4): 319–340.

McAllester, D., and Rosenblitt, D. 1991. Systematic Nonlinear Planning. In Proceedings of the Ninth National Conference on Artificial Intelligence, 634–639. Menlo Park, Calif.: American Association for Artificial Intelligence.

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

Minton, S. 1988. Quantitative Results Concerning the Utility of Explanation-Based Learning. In Proceedings of the Seventh National Conference on Artificial Intelligence, 564–569. Menlo Park, Calif.: American Association for Artificial Intelligence.

Minton, S.; Bresina, J.; and Drummond, M. 1991. Commitment Strategies in Planning: A Comparative Analysis. In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, 259–265. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Minton, S.; Drummond, M.; Bresina, J.; and Phillips, A. 1992. Total Order vs. Partial Order Planning: Factors Influencing Performance. In *Proceedings of the Third International Conference on the Principles of Knowledge Representation and Reasoning,* 83–92. San Mateo, Calif.: Morgan Kaufmann.

Minton, S.; Carbonell, J. G. ; Knoblock, C. A.; Kuokka, D. R.; Etzioni, O.; and Gil, Y. 1989. Explanation-Based Learning: A Problem-Solving Perspective. *Artificial Intelligence* 40:63–118.

Minton, S.; Knoblock, C.; Koukka, D.; Gil, Y.; Joseph, R.; and Carbonell, J. 1989. PRODIGY 2.0: The Manual and Tutorial, CMU-CS-89-146, Dept. of Computer Science, Carnegie-Mellon Univ.

Newell, A., and Simon, H. 1963. GPS: A Program That Simulates Human Thought. In *Computers and Thought*, eds. E. Feigenbaum and J. Feldman. New York: McGraw-Hill.

Nilsson, N. 1980. *Principles of Artificial Intelligence*. San Mateo, Calif.: Morgan Kaufmann.

Olawsky, D., and Gini, M. 1990. Deferred Planning and Sensor Use. In *Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, pp. 166–174. San Mateo, Calif.: Morgan Kaufmann.

Pednault, E. 1991. Generalizing Nonlinear Planning to Handle Complex Goals and Actions with Context-Dependent Effects. In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, 240–245. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Pednault, E. 1989. ADL: Exploring the Middle Ground between STRIPS and the Situation Calculus. In *Proceedings of the First Knowledge Representation Conference,* 324–332. San Mateo, Calif.: Morgan Kaufmann.

Pednault, E. 1988. Synthesizing Plans That Contain Actions with Context-Dependent Effects. *Computational Intelligence* 4(4): 356–372.

Penberthy, J. S., and Weld, D. 1994. Temporal Planning with Continuous Change. In Proceedings of the Twelfth National Conference on Artificial Intelligence, 1010–1015. Menlo Park, Calif.: American Association for Artificial Intelligence.

Penberthy, J. S., and Weld, D. 1992. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *Proceedings of the Third International Conference on the Principles of Knowledge Representation,* 103–114. San Mateo, Calif.: Morgan Kaufmann.

Peot, M., and Smith, D. 1993. Threat-Removal Strategies for Partial-Order Planning. In Proceedings of the Eleventh National Conference on Artificial Intelligence, 492–499. Menlo Park, Calif.: American Association for Artificial Intelligence.

Peot, M., and Smith, D. 1992. Conditional Nonlinear Planning. In *Proceedings of the First Conference on AI Planning Systems,* 189–197. San Mateo, Calif.: Morgan Kaufmann.

Pollack, M. 1992. The Uses of Plans. *Artificial Intelligence* 57(1): 43–68.

Reiter, R. 1980. A Logic for Default Reasoning. *Artificial Intelligence* 13:81–132.

Russell, S. 1992. Efficient Memory-Bounded Search Algorithms. In *Proceedings of the Tenth European Conference on Artificial Intelligence,* 1–5. New York: Wiley.

Sacerdoti, E. 1975. The Nonlinear Nature of Plans. In Proceedings of the Fourth International Joint Conference on Artificial Intelligence, 206–214. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Schoppers, M. 1987. Universal Plans for Reactive Robots in Unpredictable Environments. In Proceedings of the Tenth International Conference on Arti-

ficial Intelligence, 1039–1046. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Shoham, Y. 1993. Agent-Oriented Programming. *Artificial Intelligence* 60(1): 51–92.

Simmons, R. 1992. The Roles of Associational and Causal Reasoning in Problem Solving. *Artificial Intelligence* 53(2–3): 159–208.

Simmons, R. 1988a. A Theory of Debugging Plans and Interpretations. In Proceedings of the Seventh National Conference on Artificial Intelligence, 94–99. Menlo Park, Calif.: American Association for Artificial Intelligence.

Simmons, R. 1988b. Combining Associational and Causal Reasoning to Solve Interpretation and Planning Problems, Technical Report, AI-TR-1048, AI Lab, Massachusetts Institute of Technology.

Smith, D., and Peot, M. 1993. Postponing Threats in Partial-Order Planning. In Proceedings of the Eleventh National Conference on Artificial Intelligence, 500–506. Menlo Park, Calif.: American Association for Artificial Intelligence.

Tate, A. 1977. Generating Project Networks. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 888–893. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Tate, A. 1975. Interacting Goals and Their Use. In Proceedings of the Fourth International Joint Conference on Artificial Intelligence, 215–218. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Veloso, M. 1992. *Learning by Analogical Reasoning in General Problem Solving*. Ph.D. thesis, Technical Report, CMU-CS-92-174, Carnegie Mellon Univ.

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

Waldinger, R. 1977. Achieving Several Goals Simultaneously. In *Machine Intelligence 8*. Chichester, U.K.: Ellis Horwood.

Warren, D. 1976. Generating Conditional Plans and Programs. In *Proceedings of AISB Summer Conference*, 44–354.

Weld, D., and Etzioni, O. 1994. The First Law of Robotics (A Call to Arms). In Proceedings of the Twelfth National Conference on Artificial Intelligence, 1042–1047. Menlo Park, Calif.: American Association for Artificial Intelligence.

Wellman, M. 1993. Challenges for Decision-Theoretic Planning. In Foundations of Automatic Planning: The Classical Approach and Beyond, Technical Report SS-93-03, AAAI Press, Menlo Park, Calif.

Wilkins, D. 1990. Can AI Planners Solve Practical Problems? *Computational Intelligence* 6(4): 232–246.

Wilkins, D. 1988a. Causal Reasoning in Planning. *Computational Intelligence* 4(4): 373–380.

Wilkins, D. E. 1988b. *Practical Planning*. San Mateo, Calif.: Morgan Kaufmann.

Williamson, M., and Hanks, S. 1994. Optimal Planning with a Goal-Directed Utility Model. In Proceedings of the Second International Conference on AI Planning Systems, 176–181. Menlo Park, Calif.: American Association for Artificial Intelligence.

Williamson, M., and Hanks, S. 1993. Exploiting Domain Structure to Achieve Efficient Temporal Reasoning. In Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, 152–157. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Yang, Q. 1990. Formalizing Planning Knowledge for Hierarchical Planning. *Computational Intelligence* 6(1): 12–24.

Daniel Weld received bachelors degrees in both computer science and biochemistry at Yale University in 1982. He received a Ph.D. from the Massachusetts Institute of Technology Artificial Intelligence Lab in 1988 and immediately joined the Department of Computer Science and Engineering at the University of Washington, where he is now associate professor. Weld received a Presidential Young Investigator's Award in 1989 and an Office of Naval Research Young Investigator's Award in 1990. He is associate editor of the *Journal of AI Research,* was guest editor of *Computational Intelligence,* and was elected to the executive council of the American Association for Artificial Intelligence. Weld's research interests include planning, software agents, and engineering problem solving.