An Analysis of Commitment Strategies in Planning: The Details

STEVEN MINTON
JOHN BRESINA
MARK DRUMMOND
ANDREW PHILIPS

AI RESEARCH BRANCH, MAIL STOP 269-2
NASA AMES RESEARCH CENTER
MOFFETT FIELD, CA 94035
(415) 604-6527

(NASA-TM-107881) AN ANALYSIS OF COMMITMENT STRATEGIES IN PLANNING: THE DETAILS (NASA) 31 p

N92-26643

Unclas
G3/63 0091488

NASA Ames Research Center
Artificial Intelligence Research Branch

Technical Report FIA-91-08

December, 1991

FIA-91-08

An Analysis of Commitment Strategies in Planning: The Details
Steve Minton, John Bresina, Mark Drummond, and Andrew Philips — December 1991

In this paper we compare the utility of different commitment strategies in planning. Under a 'least commitment strategy', plans are represented as partial orders and operators are ordered only when interactions are detected. We investigate claims of the inherent advantages of planning with partial orders, as compared to planning with total orders. By focusing our analysis on the issue of operator ordering commitment, we are able to carry out a rigorous comparative analysis of two planners. We show that partial-order planning can be more efficient than total-order planning, but we also show that this is not necessarily so. This paper is an expanded version of a conference paper appearing in AAAI-91. We include proofs that were omitted from the conference paper.

FIA-91-09

The Blind Leading the Blind: Mutual Refinement of Approximate Theories
Smadar Kedar, John L. Bresina, and Lisa Dent — April 1991

We describe it mutual theory refinement, a method for refining world models in a reactive system. The method detects failures, explains their causes, and repairs the approximate models which cause the failures. Our approach focuses on using one approximate model to refine another.

FIA-91-10

Paradigms for Machine Learning
Pat Langley and Jeffrey C. Schlimmer — April 1991

In this paper we describe five paradigms for machine learning – connectionist -neural network- methods, genetic algorithms and classifier systems, empirical methods for inducing rules and decision trees, analytic learning methods, and case-based approaches. We consider some dimensions along which these paradigms vary in their approach to learning, and then review the basic methods used within each framework, together with open research issues. We will argue that the similarities among the paradigms are more important than their differences, and that future work should attempt to bridge the existing boundaries. Finally, we discuss some recent developments in the field of machine learning, and speculate on their impact for both research and applications.

FIA-91-11

NASA TileWorld Manual - System Version 2.0
Andrew Philips and John Bresina — May 1991

This manual documents the commands of the NASA TileWorld simulator, as well as providing information about how to run it and extend it. The simulator, implemented in Common Lisp with Common Windows, encodes a particular range in a spectrum of domains, for controllable research experiments. TileWorld consists of a two-dimensional grid of cells, a set of polygonal tiles, and a single agent which can grasp and move tiles. In addition to agent-executable actions, there is an external event over which the agent has no control; this event corresponds to a 'gust of wind'.

REPORT DOCUMENTATION PAGE

OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** Dates attached	**3. REPORT TYPE AND DATES COVERED**

4. TITLE AND SUBTITLE

Titles/Authors - Attached

5. FUNDING NUMBERS

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Code FIA - Artificial Intelligence Research Branch
Information Sciences Division

8. PERFORMING ORGANIZATION REPORT NUMBER

Attached

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Nasa/Ames Research Center
Moffett Field, CA. 94035-1000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Available for Public Distribution

[illegible] 5/14/92 BRANCH CHIEF

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Abstracts ATTACHED

14. SUBJECT TERMS	**15. NUMBER OF PAGES**
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**

An Analysis of Commitment Strategies in Planning: The Details

Steven Minton John Bresina Mark Drummond
Andrew B. Philips
Sterling Federal Systems
NASA Ames Research Center, Mail Stop 244-17
Moffett Field, CA 94035 U.S.A.

December 30, 1991

Abstract

In this paper we compare the utility of different commitment strategies in planning. Under a "least commitment strategy", plans are represented as partial orders and operators are ordered only when interactions are detected. We investigate claims of the inherent advantages of planning with partial orders, as compared to planning with total orders. By focusing our analysis on the issue of operator ordering commitment, we are able to carry out a rigorous comparative analysis of two planners. We show that partial-order planning *can* be more efficient than total-order planning, but we also show that this is not necessarily so. This paper is an expanded version of a conference paper appearing in AAAI-91. We include proofs that were omitted from the conference paper.

1 Introduction

Since the introduction of non-linear planning over a decade ago (Sacerdoti, 1977), the superiority of non-linear planning over linear planning has been tacitly acknowledged by the planning community. However, there has been little analysis supporting this intuition. In this paper, we focus on one aspect of non-linear planning: the use of *partially ordered* plans rather than *totally ordered* plans. The idea has been that a partially ordered plan allows a planner to avoid premature commitment to an incorrect operator ordering, and thus improve efficiency. We analyze the costs and benefits of using partially ordered and totally ordered plans to implement different commitment strategies for operator ordering.

Why should we be concerned about an issue that is over a decade old? Since modern planners are not very different from early planners in their basic approach, the issue is still with us. In this paper, we address the issue by first considering a simple total-order planner, and from this planner we construct a partial-order planner which can have an exponentially smaller search space. Next, we show that a second, independent source of power is available to a partial-order planner, namely, the ability to make more informed planning decisions. The relationship between our two planners demonstrates the potential utility of a least commitment strategy. We also show that a partial-order planner based on Chapman's (1987) Tweak can be less efficient than our total-order planner, and we examine why this can happen.

2 Background

Planning can be characterized as search through a space of possible plans. A *total-order planner* searches through a space of totally ordered plans; a *partial-order planner* is defined similarly. We introduce these definitions because the terms "linear" and "non-linear" are overloaded. For example, some authors have used the term "non-linearity" when focusing on the issue of *goal ordering*. That is, some "linear" planners, when solving a conjunctive goal, require that all subgoals of one conjunct be achieved before subgoals of the others; hence, planners that can arbitrarily interleave subgoals are often called "non-linear". This version of the linear/non-linear distinction is different than the partial-order/total-order distinction investigated here. The former distinction impacts planner completeness, whereas the total-order/partial-order distinction is orthogonal to this issue (Drummond & Currie, 1989).

We claim that the only significant difference between partial-order and total-order planners is planning efficiency. It might be argued that partial-order planning is preferable because a partially ordered plan can be more flexibly executed. However, this flexibility can also be achieved with a total-order planner and a post-processing step that removes unnecessary orderings from the totally ordered solution plan to yield a partial order. The polynomial time complexity of this post-processing is negligible compared to the search time for plan generation (Veloso *et al.*, 1990). Hence, we believe that execution flexibility is, at best, a weak justification for the supposed superiority of partial-order planning.

In order to analyze the relative efficiency of partial-order and total-order planning, we begin by considering a total-order planner and a partial-order planner that can be directly compared. By elucidating the key differences between these planning algorithms, we reveal some important principles that are of general relevance.

3 Terminology

A plan consists of an ordered set of *steps*, where each step is a unique operator instance. Plans can be *totally ordered*, in which case every step is ordered with respect to every other step; or *partially ordered*, in which case steps can be unordered with respect to each other. We assume that a library of operators is available, where each operator has preconditions, deleted conditions, and added conditions; each deleted condition must be a precondition. Each condition must be a non-negated propositional literal. Later in this paper we show how our results can be extended to more expressive languages.

A *linearization* of a partially ordered plan is a total order over the plan's steps consistent with the existing partial order. In a totally ordered plan, a precondition of a plan step is *true* if it is added by an earlier step and not deleted by any intervening step. In a partially ordered plan, a step's precondition is *possibly true* if there exists a linearization in which it is true, and a step's precondition is *necessarily true* if it is true in *all* linearizations. A step's precondition is *necessarily false* if it is not possibly true.

A *planning problem* is defined by a start state and goal state pair, where a *state* is a set of propositions. For convenience, we represent a problem as a two-step *initial plan*, where the first step adds the start state propositions and the preconditions of the final step are the goal state propositions. The planning process starts with this initial plan and searches through a space of possible plans. A successful search terminates with a *solution* plan, *i.e.*, a plan in which all steps' preconditions are necessarily true. The search space can be characterized as a tree, where each node corresponds to a plan and each arc corresponds to a plan transformation. Each transformation incrementally extends (*i.e.*, refines) a plan by adding additional steps or orderings. Thus, each leaf in the search tree corresponds either to a solution plan or a dead-end, and each intermediate node corresponds to an unfinished plan which can be further extended.

4 A Tale of Two Planners

In this section we define two simple planning algorithms. The first algorithm, shown in Figure 1, is TO, a total-order planner motivated by Warren (1974), Tate (1974), and Waldinger (1975). TO accepts an unfinished plan, P, and a goal set, G, containing the preconditions of steps in P which are currently false. If the algorithm terminates successfully then it returns a totally ordered solution plan. Note, there are two backtracking points in this procedure: operator selection and ordering selection. The procedure does not need to backtrack over goal choices. (Thus, the planner is presumably more efficient than one that backtracks over goal choices as well as operator and ordering choices). For our purposes, the function

TO(P, G)

1. **Termination:** If G is empty, report success and stop.
2. **Goal selection:** Let c =select-goal(G), and let O_{need} be the plan step for which c is a precondition.
3. **Operator selection:** Let O_{add} be an operator in the library that adds c. If there is no such O_{add}, then terminate and report failure. ***Backtrack point: all such operators must be considered for completeness.***
4. **Ordering selection:** Let O_{del} be the last deleter of c. Insert O_{add} somewhere between O_{del} and O_{need}, call the resulting plan P'. ***Backtrack point: all such positions must be considered for completeness.***
5. **Update goal set:** Let G' be the set of preconditions in P' that are not true.
6. **Recursive invocation:** TO(P', G').

Figure 1: The TO Planning Algorithm

`goal-select` can be any function that selects a member of G.

As used in step 4, the *last deleter* of a precondition c for a step O_{need} is a step O_{del} before O_{need} which deletes c, such that there is no other deleter of c between O_{del} and O_{need}. The first plan step is considered the last deleter if it does not add c and no other step before O_{need} deletes c.

Our purpose here is to characterize the search space of the TO planning algorithm, and the pseudo-code we give does this by defining a depth-first procedure for enumerating possible plans. All the algorithms described in this paper can also be implemented as breadth-first procedures in the obvious way, and in that case, all are provably complete as shown in Appendix B.

The second planner is UA, a partial-order planner, shown in Figure 2. UA is similar to TO in that it uses the same procedures for goal selection and operator selection, and unlike TO in that its solution plans are partially ordered. Step 4 of UA orders plan steps based on "interactions". Two steps in a plan are said to *interact* if they are unordered with respect to each other and there exists a precondition c of one step that is added or deleted by the other.[1] The significant difference between UA and TO lies in step 4: TO orders the new step with respect to *all* others, whereas UA adds *only* those orderings that are required to eliminate interactions. It is in this sense that UA is *less committed* than TO.

Since UA orders all steps which interact, the plans that are generated have a special property: each precondition in a plan is either *necessarily* true or *necessarily* false. We call such plans *unambiguous.* This property yields a tight correspondence between the two planners' search spaces. Suppose UA is given the unambiguous plan P_{ua} and that TO is given P_{to}, one of its linearizations. P_{ua} and P_{to} have the same set of goals since, by definition, each goal in P_{ua} is necessarily false and if a precondition is necessarily false, it is false in every linearization.

Consider the relationship between the way that UA extends P_{ua} and TO extends P_{to}.

[1]Note, a step that deletes c interacts with one that adds or deletes c according to this definition because a step's deleted conditions are required to be a subset of its preconditions.

```
UA($P, G$)
 1. Termination: If $G$ is empty, report success and stop.
 2. Goal selection: Let $c$ =select-goal($G$), and let $O_{need}$ be the plan step for which $c$ is a precondition.
 3. Operator selection: Let $O_{add}$ be an operator in the library that adds $c$. If there is no such $O_{add}$, then
    terminate and report failure. Backtrack point: all such operators must be considered for completeness.
 4. Ordering selection: Let $O_{del}$ be the last deleter of $c$. Order $O_{add}$ after $O_{del}$ and before $O_{need}$.
    Repeat until there are no interactions:
        ◦ Select a step $O_{int}$ that interacts with $O_{add}$.
        ◦ Order $O_{int}$ either before or after $O_{add}$.
        Backtrack point: both orderings must be considered for completeness.
    Let $P'$ be the resulting plan.
 5. Update goal set: Let $G'$ be the set of preconditions in $P'$ that are necessarily false.
 6. Recursive invocation: UA($P', G'$).
```

Figure 2: The UA Planning Algorithm

Since the two plans have the same set of goals, and since both planners use the same goal selection method, both algorithms pick the same goal; therefore, O_{need} is the same for both. Similarly, both algorithms consider the same library operators to achieve this goal. Since P_{to} is a linearization of P_{ua}, and O_{need} is the same in both plans, both algorithms find the same last deleter as well.[2] When TO adds a step to a plan, it orders the new step with respect to all existing steps. When UA adds a step to a plan, it orders the new step *only* with respect to interacting steps. UA considers all possible combinations of orderings which eliminate interactions, so for any plan produced by TO, UA produces a corresponding plan that is less-ordered or equivalent. The following sections exploit this tight correspondence between the search spaces of UA and TO. In the next section we compare the entire search spaces of UA and TO, and later we compare the number of plans actually generated under different search strategies.

5 Search Space Comparison

Recall that the search space for both TO and UA can be characterized as a tree of plans. We denote the search space of TO by $tree_{to}$, and similarly the search space of UA by $tree_{ua}$. Thus, the number of plans in a search tree is equal to the number of times the planning procedure (UA or TO) would be invoked in a complete exploration of the search space. Formally, every plan in $tree_{ua}$ and $tree_{to}$ is unique, since each step in a plan is given a unique label. Thus, although two plans in the same tree might both be instantiations of a particular operator sequence, such as $O1 \prec O2 \prec O3$, the plans are distinct because their steps have different labels.

[2]There is a unique last deleter in an unambiguous plan since two steps which delete the same condition interact, and thus, must be ordered.

We show that for any given problem, $tree_{to}$ is at least as large as $tree_{ua}$, that is, the number of plans in $tree_{to}$ is greater than or equal to the number of plans in $tree_{ua}$. This is done by proving the existence of a function $\mathcal{L}$ which maps plans in $tree_{ua}$ to sets of plans in $tree_{to}$ that satisfies the following two conditions.

1. **Totality Property**: For every plan U in $tree_{ua}$, there exists a non-empty set $\{T_1, \ldots, T_m\}$ of plans in $tree_{to}$ such that $\mathcal{L}(u) = \{T_1, \ldots, T_m\}$.

2. **Disjointness Property**: $\mathcal{L}$ maps distinct plans in $tree_{ua}$ to disjoint sets of plans in $tree_{to}$; that is, if $U_1, U_2 \in tree_{ua}$ and $U_1 \neq U_2$, then $\mathcal{L}(U_1) \cap \mathcal{L}(U_2) = \{\}$.

Let's examine why the existence of an $\mathcal{L}$ with these two properties is sufficient to prove that the size of UA's search tree is no greater than that of TO. Figure 3 provides a guide for the following discussion. Intuitively, we can use $\mathcal{L}$ to count plans in the two search trees. For each plan counted in $tree_{ua}$, we use $\mathcal{L}$ to count a non-empty set of plans in $tree_{to}$. The first property of $\mathcal{L}$ means that every time we count a plan in $tree_{ua}$, we count at least one plan in $tree_{to}$; this implies that $\mid tree_{ua} \mid \;\leq\; \sum_{U \in tree_{ua}} \mid \mathcal{L}(U) \mid$. Of course, we must further show that each plan counted in $tree_{to}$ is counted only once; this is guaranteed by the second property of $\mathcal{L}$, which implies that $\sum_{U \in tree_{ua}} \mid \mathcal{L}(U) \mid \;\leq\; \mid tree_{to} \mid$. Thus, the conjunction of the two properties implies that $\mid tree_{ua} \mid \;\leq\; \mid tree_{to} \mid$.

We can define a function $\mathcal{L}$ that has these two properties as follows. Let U be a plan in $tree_{ua}$, let T be a plan in $tree_{to}$, and let *parent* be a function from a plan to its parent plan in the tree. Then $T \in \mathcal{L}(U)$ if and only if T is a linearization of U and either both U and T are root nodes of their respective search trees, or $parent(T) \in \mathcal{L}(parent(U))$. Intuitively, $\mathcal{L}$ maps a plan U in $tree_{ua}$ to all linearizations which share common derivation ancestry.[3] This is illustrated in Figure 3, where for each plan in $tree_{ua}$ a dashed line is drawn to the corresponding set of plans in $tree_{to}$.

We can show that $\mathcal{L}$ satisfies both of the properties by induction on the depth of the search trees. Detailed proofs are in the appendix. To prove the first property, we show that for every plan contained in $tree_{ua}$, all linearizations of that plan are contained in $tree_{to}$. This can be proved by examining the tight correspondence between the search trees of UA and TO. To prove the second property, we show that $\mathcal{L}$ maps plans U_1 and U_2 at the same depth in $tree_{ua}$ to disjoint sets of plans in $tree_{to}$: if U_1 and U_2 do not have the same parent, then the property holds; if they have the same parent, then the plans U_1 and U_2 must be different (by the definition of UA), in which case their corresponding sets of linearizations are disjoint.

How much smaller is $tree_{ua}$ than $tree_{to}$? The mapping described above provides an answer. For each plan U in $tree_{ua}$ there are $\mid \mathcal{L}(U) \mid$ distinct plans in TO, where $\mid \mathcal{L}(U) \mid$ is the number of linearizations of U. The exact number depends on how unordered U is.

[3]The reader may question why $\mathcal{L}$ maps U to all its linearizations which share common derivation ancestry, as opposed to simply mapping U to all its linearizations. The reason is that the derivational history allows $\mathcal{L}$ to distinguish plans that have the same operators and orderings. For example, suppose two instantiations of the same operator sequence $O1 \prec O2 \prec O3$ exist within a $tree_{to}$ but they correspond to different plans in $tree_{ua}$. $\mathcal{L}$ can use their different derivations to determine the appropriate correspondence.

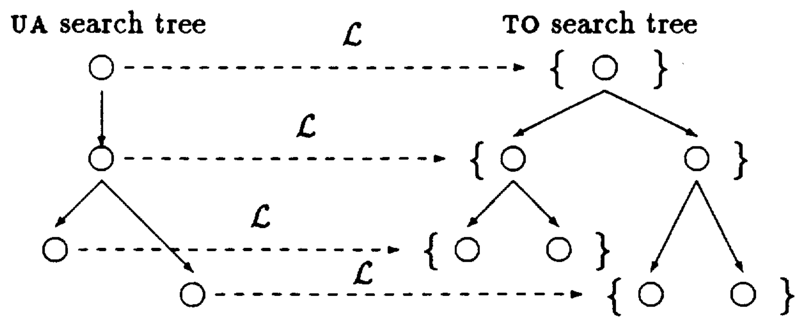

Figure 3: How $\mathcal{L}$ maps from $tree_{ua}$ to $tree_{to}$

A totally *unordered* plan has a factorial number of linearizations and a totally ordered plan has only a single linearization. Thus, the only time that the size of $tree_{ua}$ equals the size of $tree_{to}$ is when every plan in $tree_{ua}$ is totally ordered; otherwise, $tree_{ua}$ is strictly smaller than $tree_{to}$, and possibly exponentially smaller.

6 Time Cost Comparison

While the size of UA's search tree is possibly exponentially smaller than that of TO, it does not follow that UA is necessarily more efficient. Efficiency is determined by two factors: the time cost per plan in the search tree (discussed in this section) and the size of the subtree actually explored to find a solution (discussed below).

In this section we show that while UA can indeed take more time per plan, the extra time is relatively small and grows only polynomially with the size of the plan. In our analysis, the size of the plan is simply the number of steps in the plan.[4] In comparing the relative efficiency of UA and TO, we first consider the number of times that each algorithm step is executed per plan in the search tree and then consider the time complexity of each step.

As noted in the preceding sections, each node in the search tree corresponds to a plan, and each each invocation of the planning procedure for both UA and TO corresponds to an attempt to extend that plan. Thus, for both UA and TO, steps 1 and 2 are each executed once per plan, and the number of executions of step 3 per plan is bounded by a constant (the number of operators in the library). Analyzing the number of times step 4 is executed might seem more complicated, since it may be executed many times at an internal plan (i.e., internal node) in the search tree and is not executed at all at a leaf. However, notice that a new plan is generated each time step 4 is executed. Consequently, step 4 is executed once per plan generated (i.e., once for each node other than the root node). Step 5 is also executed

[4]We disregard operator size and the number of conditions in any given "state", since we assume these are bounded by a constant for a given domain. An analysis that includes these factors does not affect our conclusion.

once per plan generated since it always follows step 4. Thus, both algorithms execute each step $O(1)$ times per plan as summarized in Table 1.

In examining the costs for each step, we first note that for both algorithms, steps 1, 2, and 3 can be accomplished in $O(1)$ time. The cost of step 4, the ordering step, is different for TO and UA. In TO, step 4 is accomplished by inserting the new operator, O_{add}, somewhere between O_{del} and O_{need}. If the possible insertion points are considered starting at O_{need} and working towards O_{del}, then step 4 takes constant time, since each insertion constitutes one execution of the step. On the other hand, step 4 in UA involves carrying out interaction detection and elimination. As shown in Appendix A this step can be accomplished in $O(e)$ time, where e is the number of edges in the graph required to represent the partially ordered plan. If n is the number of steps in the plan, then in the worst case, there may be $O(n^2)$ edges in the graph, and in the best case, $O(n)$ edges. To carry out step 5 may require examining the entire plan, and thus, for UA, takes $O(e)$ time and for TO, $O(n)$ time. To summarize, UA pays the penalty of having a more complex ordering procedure (step 4), as well as the penalty for having a more expressive plan language (a partial order as compared to total order) which is reflected in the extra cost of step 5. Overall, UA requires $O(e)$ time per plan, whereas TO only requires $O(n)$ time per plan.

Step	Executions Per Plan	TO Cost	UA Cost
1	1	$O(1)$	$O(1)$
2	1	$O(1)$	$O(1)$
3	$O(1)$	$O(1)$	$O(1)$
4	1	$O(1)$	$O(e)$
5	1	$O(n)$	$O(e)$

Table 1: Cost Per Plan Comparisons

7 Overall Efficiency Comparison

The previous sections compared TO and UA in terms of relative search space size and relative time cost per plan. The extra processing time required by UA for each plan would appear to be justified since its search space may contain exponentially fewer plans. To complete our analysis, we must consider the number of plans actually explored by each algorithm under a given search strategy. (Recall that a plan is explored by an algorithm if the algorithm is called with that plan as its argument.)

Consider a *breadth-first* search technique that explores the entire search tree up to the depth of the smallest solution plan. By the search tree correspondence established earlier, both algorithms find the first solution at the same depth. Thus, TO explores all linearizations of the plans explored by UA. We can formalize the overall efficiency comparison as follows. For a plan U in $tree_{ua}$, we denote the number of steps in U by n_u, and the number of edges by

e_u. Then for each plan U that UA generates, UA incurs time cost $O(e_u)$; whereas, TO incurs time cost $O(n_u) \cdot \mid \mathcal{L}(U) \mid$, where $\mid \mathcal{L}(U) \mid$ is the number of linearizations of U. Therefore, the ratio of the total time costs of TO and UA is as follows, where $bf(tree_{ua})$ denotes the subtree considered by UA under breadth-first search.

$$\frac{\text{cost}(\text{TO}_{bf})}{\text{cost}(\text{UA}_{bf})} = \frac{\sum_{U \in bf(tree_{ua})} O(n_u) \cdot \mid \mathcal{L}(U) \mid}{\sum_{U \in bf(tree_{ua})} O(e_u)}$$

The cost comparison is not so clear-cut for depth-first search, since TO does not necessarily explore all linearizations of the plans explored by UA. A plan in a search tree is *completable* if it is on a path to a solution, otherwise, it is *uncompletable*. If a plan in $tree_{ua}$ is uncompletable, all of the corresponding plans in $tree_{to}$ are also uncompletable. If a UA plan is completable, then some subset of the corresponding TO plans are completable. If, under a depth-first strategy, UA and TO generate corresponding plans in the same order, then *(i)* for every uncompletable plan U that UA explores, TO explores all plans in $\mathcal{L}(U)$ and *(ii)* for every UA plan U that succeeds, TO generates at least one plan in $\mathcal{L}(U)$. However, in actuality, UA and TO need not generate corresponding plans in the same order. In this case, while the search *spaces* correspond, there is no guarantee that the planners will explore corresponding subtrees.

In fact, the expected performance of the two planners depends on exactly how solutions are distributed within their search spaces. To see this, assume for argument's sake that the expected number linearizations for a UA leaf plan is independent of whether the plan fails or succeeds. Then we would expect the ratio of solution nodes to failed leaf nodes to be the same in $tree_{ua}$ and $tree_{to}$. Then, if we also assume that the solution nodes are randomly distributed within each search space, it is easily seen that both planners can be expected to search the same number of nodes! This is illustrated in Figure 4. In practice, however, whereas the first assumption seems reasonable, the second of these assumptions is unrealistic. Typically, plans that fail tend to be grouped together in the search space. This occurs because a "wrong decision" near the top of the search tree can lead to an entire subtree of failed plans, as shown in Figure 5. Intuitively, if plans in $tree_{to}$ that map to the same plan in $tree_{ua}$ tend to be grouped together, then UA will have an advantage over TO, due to the relatively smaller size of its search space.

This intuition is supported by empirical experimentation with depth-limited versions of UA and TO. In a blocksworld domain where *all* steps interact, UA tends to Explore the same number of plans as TO under depth-first search. On another version of the blocksworld, where the probability of two randomly selected steps interacting is approximately 0.5, UA tends to explore many fewer plans. For example, on a representative problem, with a solution depth (and depth-bound) of eight, TO explored 8.0 times as many plans as UA. This ratio tends to increase with solution depth; for a problem with solution depth of nine, TO explored 15.4 times as many plans. Although UA required more time per plan, in terms of total search time UA ran 4.6 times faster than TO on the first problem and 9.0 times faster than TO on the second problem. The results under breadth-first search were also as expected: when all steps interact, UA and TO search exactly the same number of plans, and when relatively few

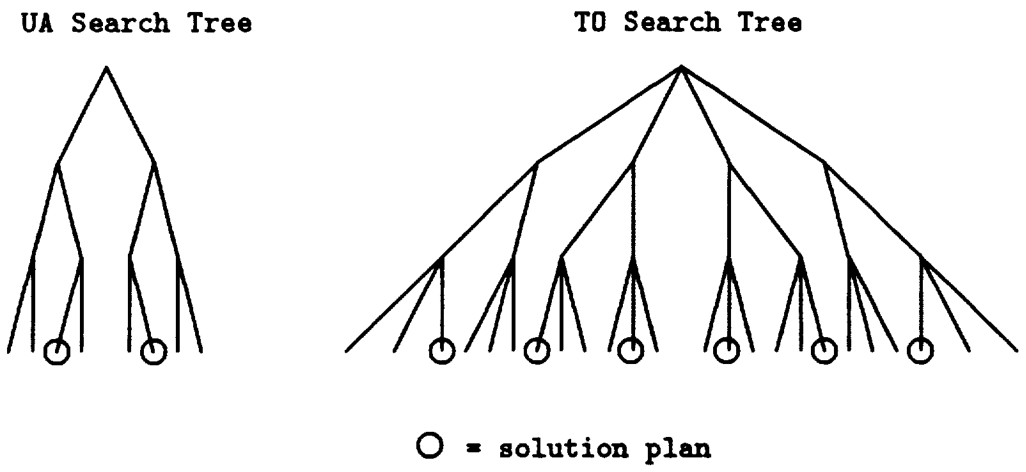

Figure 4: UA and TO search trees with evenly distributed solutions. The ratio of solutions to leaf nodes is .25

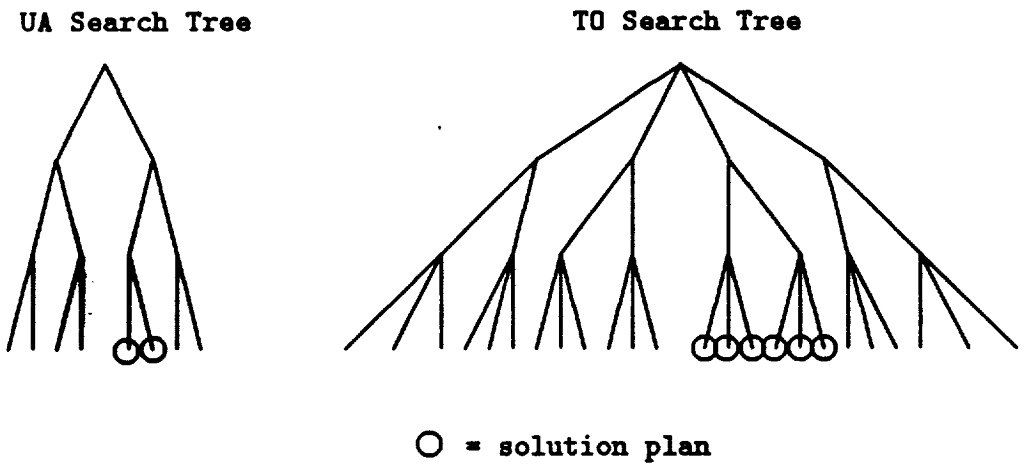

Figure 5: UA and TO search trees with solutions clumped together. The ratio of solutions to leaf nodes is 1:4.

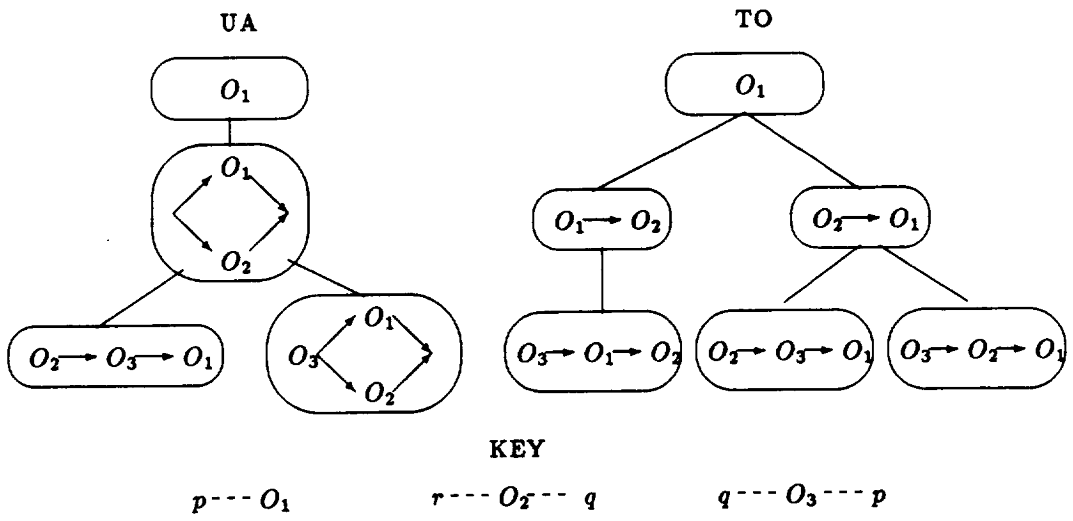

Figure 6: Comparison of UA and TO on an example.

steps interact, UA explores many fewer plans than TO. For example, in our low-interaction version of the blocksworld, on a problem where the first solution is found at depth seven, TO explored 4.8 times as many plans as UA, and UA ran 2.8 times faster. We caution that this is a small-scale study, intended only to illustrate our theoretical results.

The performance of TO can be improved with the addition of dependency-directed backtracking. We note that TO can be augmented with dependency-directed backtracking so that it behaves similarly to UA in certain respects. Specifically, when TO backtracks to a plan, a dependency analysis may indicate whether or not the failure below was independent of the ordering decision that was made in extending that plan. Of course, this dependency analysis increases the cost per plan.

8 Heuristic Advantages

It is often claimed that partial-order planners are more efficient due to their ability to make more informed ordering decisions. So far, we have shown that a partial-order planner can be more efficient simply because its search tree is smaller, independent of its ability to make more informed decisions. We now show that a partial-order planner does in fact have a "heuristic advantage" as well.

In the UA planning algorithm, step 4 arbitrarily orders interacting plan steps. Similarly, step 4 of TO arbitrarily chooses an insertion point for the new step. It is easy to see, however, that some orderings should be tried before others in a heuristic search. This is illustrated by Figure 6, which compares UA and TO on a particular problem. The key in the figure describes the relevant conditions of the library operators, where preconditions are indicated to the left of an operator and added conditions are indicated to the right (there are no deletes). For brevity, the start step and final step of the plans are not shown. Consider the plan in $tree_{ua}$ with unordered steps O_1 and O_2. When UA introduces O_3 to achieve precondition p of O_1, step 4 of UA will order O_3 with respect to O_2, since these steps interact. However, it makes more sense to order O_2 before O_3, since O_2 achieves precondition q of O_3. This illustrates a simple planning heuristic: "prefer the orderings that yield the fewest false preconditions". This strategy is not guaranteed to produce the optimal search or the optimal plan, but tends to be effective and is commonly used.

Notice, however, that TO cannot exploit this heuristic as effectively as UA because it must prematurely commit to an ordering on O_1 and O_2. Due to this inability to postpone an ordering decision, TO must choose arbitrarily between the plans $O_1 \prec O_2$ and $O_2 \prec O_1$, before the impact of this decision can be evaluated.

In general, UA is more informed than TO by any heuristic h that satisfies the following property: for any UA plan U and corresponding TO plan T, $h(U) \geq h(T)$; that is, a partially ordered plan must be rated at least as high as any of its linearizations. (Note that for unambiguous plans the heuristic function in our example satisfies this property.) When we say that UA is *more informed* than TO, we mean that under h, some child of U is rated at least as high as every child of T. This is true since every child of T is a linearization of some child of U, and therefore no child of T can be rated higher than a child of U. Furthermore, there may be a child of U such that none of its linearizations is a child of T, and therefore this child of U can be rated higher than every child of T. Assuming that h is a good heuristic, this means that UA can make a better choice than TO.

9 A Less Committed Planner

We have shown that UA, a partial-order planner, has certain computational advantages over a total-order planner, TO, due to its ability to delay commitments. However, there are planners that are even less committed than UA. In fact, there is a continuum of commitment strategies that we might consider. At the extreme liberal end of the spectrum is the strategy of maintaining a *totally unordered* set of steps during search, until there exists a linearization that is a solution plan.

Compared to many well-known planners, UA is conservative since it requires each plan to be unambiguous. This is not required by NOAH (Sacerdoti, 1977), NonLin (Tate, 1977), and Tweak (Chapman, 1987), for example. How do these less-committed planners compare to UA and TO? One might expect a less-committed planner to have the same advantages over UA that UA has over TO. However, this is not necessarily true. For example, we show in this section that Tweak's search tree is larger than TO's in some circumstances.[5] See Figure 7 for a propositional planner, MT, based on Chapman's (1987) Modal Truth Criterion, the formal

[5]We use Tweak for this comparison because, like UA and TO, it is a formal construct rather than a realistic planner, and therefore more easily analyzed.

```
MT($P, G$)
  1. Termination: If $G$ is empty, report success and stop.
  2. Goal selection: Let $c$ be a goal in $G$, and let $O_{need}$ be the plan step for which $c$ is a precondition.
  3. Operator selection: Let $O_{add}$ be either a plan step possibly before $O_{need}$ that adds $c$ or an operator
     in the library that adds $c$. If there is no such $O_{add}$, then terminate and report failure. Backtrack
     point: all such operators must be considered for completeness.
  4. Ordering selection: Order $O_{add}$ before $O_{need}$. Repeat until there are no steps possibly between
     $O_{add}$ and $O_{need}$ which delete $c$:
        Let $O_{del}$ be such a step; choose one of the following
        ways to make $c$ true for $O_{need}$
          o Order $O_{del}$ before $O_{add}$.
          o Order $O_{del}$ after $O_{need}$.
          o Choose a step $O_{knight}$ that adds $c$ that is possibly
            between $O_{del}$ and $O_{need}$; order it after $O_{del}$ and
            before $O_{need}$.
            Backtrack point: all alternatives must be considered for completeness.
     Let $P'$ be the resulting plan.
  5. Update goal set: Let $G'$ be the set of preconditions in $P'$ that are not necessarily true.
  6. Recursive invocation: MT($P', G'$).
```

Figure 7: A Propositional Planner based on the MTC

statement that characterizes Tweak's search space.

The proof that UA's search tree is no larger than TO's search tree rested on the two properties of $\mathcal{L}$ elaborated in section 5. By investigating the relationship between MT and TO, we found that the second of these properties does not hold for MT, and its failure illustrates how MT can explore more plans than TO (and consequently UA) on certain problems. The second property of section 5 guarantees that UA does not generate "overlapping" plans. The example in Figure 8 shows that MT fails to satisfy this property because it can generate plans that share common linearizations, leading to considerable redundancy in the search. The figure shows three steps, O_1, O_2, and O_3, where each O_i has precondition p_i, and added conditions g_i, p_1, p_2, and p_3. The final step has preconditions g_1, g_2, and g_3, but the start and final steps are not shown in the figure. In the plan at the top of the figure, constructed by MT, goals g_1, g_2, and g_3 have been achieved, but p_1, p_2, and p_3 remain to be achieved. Subsequently, in solving the precondition p_1, MT generates plans which share the linearization $O_3 \prec O_2 \prec O_1$ (among others). In comparison, both TO and UA only generate the plan $O_3 \prec O_2 \prec O_1$ once. In fact, it is simple to show that, under breadth-first search, MT explores many more plans than TO on this example (and also more than UA, by transitivity) due to the redundancy in its search space.

This example shows that although one planner may be less committed than another, it is not necessarily more efficient. In general, a partially ordered plan can represent a large set of linearizations, but of course, there can be many more partial orders over a set of steps

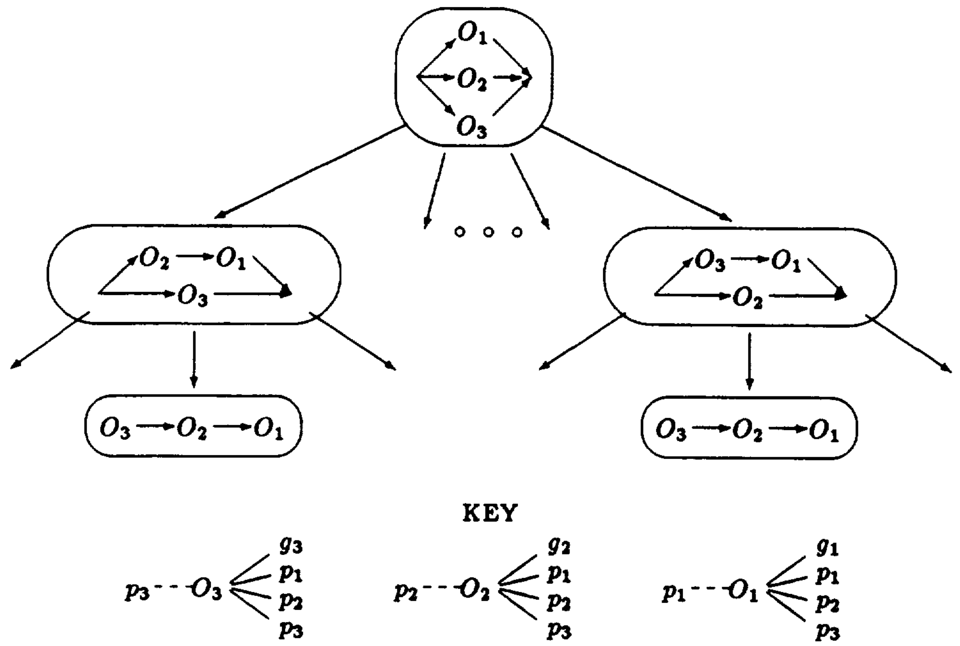

Figure 8: "Overlapping" plans.

than there are linearizations. A general lesson from this is that a search space should be defined so as to minimize redundancy whenever possible. In particular, considering partially ordered plans with linearization overlap should be avoided. This conclusion was recently and independently discovered by McAllester and Rosenblitt (1991) as well.

10 More Expressive Languages

Up to this point, we have only considered a very restricted planning language in which the operators must unconditionally add and delete propositions. However, many problems demand operators with variables, conditional effects, or conditional preconditions. Fortunately, our basic results extend to more expressive operator languages. In many important cases, UA and TO can be extended so that the search space correspondence still holds. In such cases, the relative advantages of UA over TO will be preserved as long as the time cost of detecting possible interactions remains relatively small.

Let us first consider the simple extension to our language where library operators have variables. We have implemented simple versions of TO and UA for this language. The description of TO is shown in Figure 9 (and UA follows in the obvious way). The new algorithm is identical to the original, except for the addition of a step which instantiates the operator. Thus, this algorithm requires that all possible bindings be computed by this step. This is accomplished, as in other planners (Minton *et al.*, 1989) by requiring operators to have a set of ***static*** or *type* preconditions. For example, below we show a definition of the blocksworld STACK operator in which the static preconditions Is-Block(x) and Is-Block(y) allow the planner to find the complete set of possible bindings for x and y.

```
STACK(x y)
PRECONDITIONS:
   Is-Block(x)
   Is-Block(y)
   Not-equal(x y)
   Holding(x)
   Clear(y)
EFFECTS:
   ADD On(x y)
   DELETE Holding(x)
   DELETE Clear(y)
```

As it turns out, the analysis that we have done so far holds without modification for this language. In particular, the search space correspondence holds because the TO and UA algorithms are essentially unchanged. The time cost analysis holds because the complexity of identifying goals and detecting interactions is unchanged; since operators are immediately instantiated, the plans that are generated do not contain variables.

How about other languages? The relationship between UA and TO's search spaces depends on the fact that UA generates unambiguous plans. In general, however, the work required to demonstrate step interaction tends to increase with the expressiveness of the operator language used (Dean & Boddy, 1988; Hertzberg & Horz, 1989). Thus we might expect that the expense of the "disambiguation" process used by UA will increase with expressiveness of the language. Presumably, the relative savings in search space that UA enjoys will then be eventually outweighed by the extra time required to extend a plan. In other words, the potential advantages of UA over TO hold only if we can find a relatively inexpensive way for UA to maintain unambiguous plans. Fortunately, we believe the cost of detecting interactions can often be kept low by relying on conservative definitions of step interaction.

As an example, let us consider a simple propositional language with conditional effects, such as "If p, then Add q". An operator can thus add or delete propositions depending on the state in which it is executed. We will refer to conditions such as "p" in our example as *dependency conditions.*[6] Chapman (1987) showed that with this type of language, it is NP-hard to decide whether a precondition is true or false in a partially-ordered plan. We can employ (the original) TO to deal with this language without any modification. Interestingly, if we slightly modify the definition of step interaction, we can also employ (the original) UA without modification. In particular, we can redefine step interaction for UA as follows:

[6]Note that, for simplicity, we still require that any condition that is deleted also be a precondition or a dependency condition.

Two steps in a plan are said to *interact* if they are unordered with respect to each other and there exists a precondition or dependency condition of one step that can be added or deleted by the other.

This definition allows interacting steps to be detected via an inexpensive syntax check, and thus the cost of step interaction and disambiguation is kept low. In fact, checking whether two operators interact requires only constant time, just as with our original language. Furthermore, because the plans produced by UA are unambiguous, UA can determine whether a precondition is true or false in polynomial time. In fact, it suffices to take any linear ordering of the plan, which can be accomplished in $O(e)$ time using a topological sort, and then use the same procedure as TO for determining the truth or falsity of a precondition.[7] Consequently, with this language, TO and UA have exactly the same time complexity as with our original language, and our original analysis holds without modification.

As this example has illustrated, the potential advantages of UA over TO can extend to more expressive languages provided that the cost of maintaining unambiguous partially-ordered plans is not much more than that of maintaining totally-ordered plans. We believe that this can generally be accomplished through the use of conservative strategies for interaction detection and resolution. By conservative, we mean that a strategy is guaranteed to insert orderings where required, but may also occasionally introduce orderings unnecessarily. For example, according to the above definition, two steps interact if they both conditionally add **p**. However, in any given plan, the conditions may be such that neither step actually adds **p**. Thus, the cost of interaction detection is low, since the planner does not need to check the conditions in the plan, however, the use of unnecessary orderings can lead to a larger search space than is required.

The larger lesson here is that the cost of plan extension is not solely dependent on the expressiveness of the operator language, it also depends on how the planner deals with that expressiveness. So, although plan extension is NP-hard for languages with conditional effects, this does not necessarily effect UA, as we have shown. By relying on a conservative disambiguation methods, we can preserve UA's advantages over TO.

11 Concluding Remarks

By focusing our analysis on the single issue of operator ordering commitment, we were able to carry out a rigorous comparative analysis of two planners. In contrast, most previous work has focused on the definition of a single planner, and comparative analyses have been rare.[8] We have shown that the search space of one partial-order planner, UA, is *never* larger than the search space of one total-order planner, TO. Indeed for certain problems, UA's search space is exponentially smaller than TO's. Since UA pays only a small polynomial time

[7]One method for determining whether a precondition is true or false is to simply simulate the plan, which can be accomplished in $O(n)$ time.

[8]Soderland and Weld (1991) have very recently, and independently, carried out a comparative analysis of two planners, corroborating some of the results reported in Section 5.

TO(P, G)

1. **Termination:** If G is empty, report success and stop.
2. **Goal selection:** Let c be a goal in G, and let O_{need} be the plan step for which c is a precondition.
3. **Operator selection:** Let O_{add} be an operator in the library that adds c. If there is no such O_{add}, then terminate and report failure. *Backtrack point: all such operators must be considered for completeness.*
4. **Bindings selection:** Let $v_1, v_2, \ldots$ be the variables in O_{add}. Consider a set of bindings for $v_1, v_2, \ldots$ and instantiate the operator. *Backtrack point: all such sets of bindings must be considered for completeness.*
5. **Ordering selection:** Let O_{del} be the last deleter of c. Insert O_{add} somewhere between O_{del} and O_{need}, call the resulting plan P'. *Backtrack point: all such positions must be considered for completeness.*
6. **Update goal set:** Let G' be the set of preconditions in P' that are not true.
7. **Recursive invocation:** TO(P', G').

Figure 9: The TO planning algorithm, modified for operators with variables

increment per plan over TO, it is generally more efficient. We have also demonstrated that UA can be more informed than TO under a certain class of heuristic evaluation functions. Lastly, we have shown that partial-order planners do not necessarily have smaller search spaces; in particular, we demonstrated that a Tweak-like planner can have a larger search space than TO on some problems.

How general are these results? While our analysis has considered only two specific planners, the tradeoffs that we have examined are of general relevance. We believe these tradeoffs are manifested in other styles of planner, including temporal-projection planners (Drummond, 1989) and STRIPS-like planners such as Prodigy (Minton *et al.*, 1989). We conjecture that one can define a partial-order version of Prodigy, for instance, which corresponds to the original in the same way that UA corresponds to TO. The key difficulty in analyzing possible correspondences between such planners is establishing a mapping between the planners' search trees.

The general lesson from this work is that partial-order planning *can* be more efficient than total-order planning, but is not necessarily more efficient. When designing a partial-order planner, one must understand the effect of plan representation on the planner's search space, the cost incurred per node, and sources of possible redundancy in the search space.

12 Acknowledgements

The TORPOR group thanks Mark Boddy for helping start this project and for contributing to the design of the TO algorithm. We also thank John Allen, Smadar Kedar, Rich Keller, Bernadette Kowalski, Wayne Iba, Amy Lansky, Rich Levinson, and Keith Swanson for their helpful comments on previous drafts.

A Implementation of Planning Algorithms

In this section we describe the implementation of TO and UA in more detail. Figure 10 shows a procedure implementing the TO algorithm. The procedure is straightforward, selecting a goal and then looping through the choices for an operator and position. Each of the steps in the algorithm can be implemented in constant time, except for the call to the subprocedure Update-Goal-Set.

The Update-Goal-Set procedure is shown in Figure 11. The procedure takes a step that has been newly inserted into a plan, and it updates the set of unachieved goals. There are three basic phases in the algorithm. First, the algorithm sweeps backward through the plan, starting at the position of the newly introduced step and moving backward until the initial step is reached. The algorithm compares the preconditions of the new step to the add and delete lists for each prior step, marking those preconditions that match with an added condition as *achieved*, and those that match a deleted condition as *unachieved*. Since each condition can be marked in constant time and the size of each operator is assumed to be bounded by a constant, the complexity of this phase is equal to the number of predecessor steps, which is $O(n)$.

Next, the algorithm sweeps forward through the plan, starting after the newly introduced step and moving forward until the final step is reached. The algorithm compares the preconditions each step along the way to a list of *active* propositions added and deleted by the new step, marking the preconditions *achieved* or *unachieved* as appropriate. Initially, all of the added and deleted propositions are *active*. If a proposition added (deleted) by the new step is ever deleted (added) by a subsequent step, the added (deleted) proposition is marked as inactive. In this way the algorithm guarantees that the effects of a step are correctly propagated. As with the first phase, the time complexity of this phase is $O(n)$.

Finally, the preconditions that are marked as *unachieved* are collected by traversing the plan, which also takes $O(n)$ time. Thus, the complexity of this procedure is $O(n)$.

A procedure implementing the UA algorithm is shown in Figure 12. Similarly to TO, the procedure selects a goal and then loops through the choices for an operator and position. Each of the steps in the algorithm can be implemented in constant time, except for the calls to the subprocedures Update_Goal_Set_for_UA_Plan, Disambiguate, and Disambiguate_Backtrack.

Update_Goal_Set_for_UA_Plan is not shown, but it is quite similar to Update_Goal_Set, which is called by TO. The only difference is that instead of sweeping backward and forward through a total-order, the procedure sweeps backward and forward through a partial-order. This can be accomplished in $O(e)$ time. Alternatively, the same effect can be achieved by taking the partial-order, converting it to a total-order using an $O(e)$ topological sort, and then calling the original Update_Goal_Set procedure.

The subprocedures Disambiguate and Disambiguate_Backtrack are shown in Figures 13 and 14, respectively. The first procedure determines the steps that interact with the new step and orders them after the new step. If this ordering does not succeed, Disambiguate_Backtrack is called to undo the orderings one by one. When an ordering is undone, the alternative ordering is then tried, and Disambiguate is called again. In this way, all combinations of orderings are eventually tried.

Together, the two procedures try all consistent sets of orderings between the new step and the interacting steps. For each combination of orderings, the two procedures must mark all the steps that are transitively before and after the new step. This can be accomplished in an efficient manner, so that each step is only marked once. As each ordering is accomplished, Disambiguate keeps track of which steps are still in parallel to the new step, by marking all plan steps that are now before or after the new step. Each edge in the partial order only need be examined once. Whenever a previously parallel step is ordered before (or after) the new step, it is marked as *before* (or *after*) and then any steps before (or after) the newly marked step are recursively marked. The recursion stops whenever a step is already marked. Thus, an edge is traversed only if it is before (or after) a step that has just been marked. Since a step is marked (and then unmarked) at most once per each combination of orderings, the complexity of the disambiguation process is $O(e)$ per child node that is generated, as discussed in section 6.

```
;;; pre($s$) = the precondition predicates of step $s$
;;; del($s$) = the delete predicates of step $s$
;;; unachieved($c$) = the predicate $c$ is marked as unachieved
;;; $P =< S, O >$
;;; $S = \{s | s$ is a step in plan $P\}$
;;; $O = \{< s_1, s_2 > | s_1 \in S$, $s_2 \in S$ and $s_1$ ordered before $s_2\}$
;;; $G = \{< c, s > | c \in$ pre($s$) and unachieved($c$) $\}$

Procedure TO ($P$,$G$)
    If $G$ is empty, return SUCCESS
    Choose a $c$ and $s_{need}$ such that $< c, s_{need} > \in G$
    Let $OPS_{relevant}$ be the set of operators that achieve $c$
    while Not_Empty($OPS_{relevant}$) do
        $s_{add} \leftarrow$ Make-Unique-Step(Pop($OPS_{relevant}$)
        $poststep \leftarrow s_{need}$
        repeat
            $prestep \leftarrow$ Predecessor($poststep$)
            Let plan $P'$ be the result of inserting step $s_{add}$ between $prestep$ and $poststep$
            $G' \leftarrow$ Update_Goal_Set($s_{add}$, $G$)
            If TO($P'$, $G'$) = SUCCESS, then return SUCCESS
            $poststep \leftarrow prestep$
        until $c \in$ del($prestep$) or Predecessor($prestep$) = NIL
    end-while
    Return FAILURE
end-of TO
```

Figure 10: Implementation of the TO Planning Algorithm

```
procedure Update-Goal-Set (new-step, unachieved)

    possible-goals ← Precondition_List (new-step)
    step ← new-step
    /*  All Preconditions in possible-goals are unmarked at this point.  */
    while (step ← Predecessor (step)) do
        For each unmarked Precondition in possible-goals which matches an Add for this
            step, mark the Precondition as Achieved.
        For each unmarked Precondition in possible-goals which matches a Delete for
            this step, mark the Precondition as Unachieved.
    end-while
    If any Precondition in possible-goals is still unmarked, mark it as Unachieved.

    adds ← Add_List (new-step)
    deletes ← Delete_List (new-step)
    step ← new-step
    /*  All Adds in adds and Deletes in deletes are marked Active at this point.  */
    while (step ← Successor (step)) do
        For each Unachieved Precondition in this step which matches with an Active Add in
            adds, mark that Precondition as Achieved.
        For each Achieved Precondition in this step which matches with an Active Delete in
            deletes, mark that Precondition as Unachieved.
        For each Active Add in adds which matches an Add or Delete for this step, mark
            that Add as Inactive.
        For each Active Delete in deletes which matches an Add or Delete for this step,
            mark that Delete as Inactive.
    end-while

    Collect all Unachieved Preconditions in the plan and place them in unachieved.
end-of Update-Goal-Set
```

Figure 11: Update-Goal-Set Procedure

```
;;; pre($s$) = the precondition predicates of step $s$
;;; del($s$) = the delete predicates of step $s$
;;; unachieved($c$) = the predicate $c$ is marked as unachieved
;;; $P = < S, O >$
;;; $S = \{s | s \text{ is a step in plan } P\}$
;;; $O = \{< s_1, s_2 > | s_1 \in S, s_2 \in S \text{ and } s_1 \text{ ordered before } s_2\}$
;;; $G = \{< c, s > | c \in \text{pre}(s) \text{ and unachieved}(c)\}$

Procedure UA ($P$,$G$)
    If $G$ is empty, return SUCCESS
    Choose $c$ and $s_{need}$ such that $< c, s_{need} > \in G$.
    Let $s_{lastdel}$ be the last deleter of $c$ in the plan $P$.
    $OPS_{relevant} \leftarrow$ Get_Relevant_Operators($c$)
    while Not_Empty($OPS_{relevant}$) do
        $s_{add} \leftarrow$ Make-Unique-Step(Pop($OPS_{relevant}$))
        Order $s_{add}$ before $s_{need}$ in $P$.
        Order $s_{add}$ after $s_{lastdel}$ in $P$.
        ordering_stack $\leftarrow$ NULL
        $< P$, ordering_stack$> \leftarrow$ Disambiguate($P$, $s_{add}$, ordering_stack)
        $G \leftarrow$ Update_Goal_Set_for_UA_Plan($s_{add}$, $G$)
        If UA($P$,$G$) = SUCCESS, then return SUCCESS
        while Not_Empty(ordering_stack) do
            $< P$, ordering_stack$> \leftarrow$ Disambiguate_Backtrack($P$, $s_{add}$, ordering_stack)
            $G \leftarrow$ Update_Goal_Set($s_{add}$, $G$)
            If UA($P$,$G$) = SUCCESS, then return SUCCESS
        end-while
    end-while
    Return FAILURE
end-of UA
```

Figure 12: Implementation of the UA Planning Algorithm

```
Procedure Disambiguate($P$, $s_{add}$, ordering_stack)
    Mark all steps in $P$ that are before $s_{add}$ as BEFORE.
    Mark all steps in $P$ that are after $s_{add}$ as AFTER.
    Let parallel-steps be all steps in $P$ which are neither before nor after $s_{add}$.
    while Not_Empty(parallel-steps) do
        step $\leftarrow$ Pop(parallel-steps)
        if step is not marked BEFORE or AFTER, and if Interact(step,$s_{add}$),
          then Order step after $s_{add}$ in $P$
               ordering_stack $\leftarrow$ Push(< step,AFTER>,ordering_stack)
               Mark all unmarked steps after step as AFTER
        end-if
    end-while
    Return < $P$,ordering_stack>
end-of Disambiguate
```

Figure 13: Disambiguate Procedure used by UA

```
Procedure Disambiguate_Backtrack($P$, $s_{add}$, ordering_stack)
    while Not_Empty(ordering_stack) do
        < step,ordering> $\leftarrow$ Pop(ordering_stack)
        if ordering = AFTER
          then Undo ordering of step after $s_{add}$ in $P$
               Order step before $s_{add}$ in $P$
               ordering_stack $\leftarrow$ Push(< step,BEFORE>,ordering_stack)
               Return Disambiguate($P$, $s_{add}$, ordering_stack)
          else Undo ordering of step before $s_{add}$ in $P$
        end-if
    end-while
    Return < $P$,NULL>
end-of Disambiguate_Backtrack
```

Figure 14: Disambiguate_Backtrack Procedure used by UA

B Proofs

B.1 Definitions

- A *plan* is a pair $< \theta, \prec >$, where θ is a set of steps, and $\prec$ is the "before" relation on θ, *i.e.* $\prec$ is a *strict partial order* on θ. Notationally, $O_1 \prec O_2$ if and only if $(O_1, O_2) \in \prec$.

- A problem is *solvable* if there exists a plan that solves the problem. A planner is *complete* iff the planner will produce a solution plan for every solvable problem.

- Two plans, $P_1 =< \theta_1, \prec_1>$ and $P_2 =< \theta_2, \prec_2>$ are said to be *equivalent*, denoted $P_1 \simeq P_2$, if there exists a bijective function f from θ_1 to θ_2 such that:
  - for all $s \in \theta_1$, s and $f(s)$ are instances of the same operator, and
  - for all $O', O'' \in \theta_1$, $O' \prec O''$ if and only if $f(O') \prec f(O'')$.

- A plan P_2 is a *1-step extension* of a plan P_1 with respect to TO (or UA) if P_2 is equivilent to some plan produced from P_1 in one invocation of TO (or UA).

- P_1 is a *subplan of* $P_2 =< \theta_2, \prec_2>$ denoted $P_1 \subseteq P_2$, if $P_1 \simeq< \theta_1, \prec_1>$ where
  - $\theta_1 \subseteq \theta_2$ and
  - $\prec_1 \subseteq \prec_2$.

- P_1 is a *strict* subplan of P_2, denoted $P_1 \subset P_2$, if $P_1 \subseteq P_2$ and P_1 has fewer steps than P_2.

- P_1 is a *linearization of* $P_2 =< \theta_2, \prec_2>$ if P_1 is totally ordered and $P_1 \simeq< \theta_2, \prec_1>$ where $\prec_2 \subseteq \prec_1$.

- A solution plan P is a *compact solution* to a problem if no strict subplan of P solves the problem.

- For a given problem, we define the search tree $tree_{to}$ as the complete tree of plans that are generated by the TO algorithm on that problem. $tree_{ua}$ is the corresponding search tree generated by UA on the same problem.

- Given a search tree, let *parent* be a function from a plan to its parent plan in the tree. P_1 is the is the parent of P_2, denoted $P_1 = parent(P_2)$, only if P_2 is a 1-step extension of P_1.

- Given $U \in tree_{ua}$ and $T \in tree_{to}$, $T \in \mathcal{L}(U)$ if and only if plan T is a linearization of plan U and either both U and T are root nodes of their respective search trees, or $parent(T) \in \mathcal{L}(parent(U))$.

B.2 Extension and Existence Lemmas

TO-Extension Lemma: Consider totally ordered plans $T_0 =< \theta_0, \prec_0>$ and $T_1 =< \theta_0 \cup \{O_{add}\}, \prec_1>$, such that $\prec_0 \subset \prec_1$. Let G be the set of false preconditions in T_0. Then T_1 is a one-step extension of T_0 by TO if:

- $c =$ select-goal(G), where c is the precondition of step O_{need} in T_0, and
- O_{add} adds c, and
- O_{add} is ordered before O_{need} in T_1, and
- O_{add} is ordered after the last deleter of c in T_1.

Proof Sketch: This lemma follows from the definition of TO. Given plan T_0, with false precondition c, once TO selects c as the goal, TO will consider all operators that achieve c, and for each operator TO considers all positions before c and after the last deleter of c.

UA-Extension Lemma: Consider a plan $U_0 =< \theta_0, \prec_0>$ produced by UA and plan $U_1 =< \theta_1, \prec_1>$, such that $\theta_1 = \theta_0 \cup \{O_{add}\}$ and $\prec_1 \supset \prec_0$. Let G be the set of false preconditions of the steps in U_0. Then U_1 is a one-step extension of U_0 with respect to UA if:

- $c =$ select-goal(G), where c is the precondition of step O_{need} in U_0, and
- O_{add} adds c, and
- $\prec_1$ is the minimal set of consistent orderings such that
  - $\prec_0 \subseteq \prec_1$, and
  - $(O_{add}, O_{need}) \in \prec_1$, and
  - $(O_{del}, O_{add}) \in \prec_1$, where O_{del} is the last deleter of c in U_1, and
  - $(O', O'') \in \prec_1$ if O' and O'' would interact if they were in parallel.

Proof Sketch: This lemma follows from the definition of UA. Given plan U_0, with false precondition c, UA considers all operators that achieve c, and for each such operator UA then inserts it in the plan such that it is before c and after the last deleter. UA then considers all consistent combinations of orderings between the new operator and the operators with which it interacts. No other orderings are added to the plan.

Existence Lemma: Let P_1 be a one-step extension of P_0 with respect to TO (or UA). If plan P_0 is a member of $tree_{to}$ (or $tree_{ua}$), then some child of P_0 is equivalent to P_1.
Proof Sketch: Since P_1 is a one-step extension of P_0, there must be a series of choices by TO (or UA) in extending P_0. A corresponding series of choices can be made in expanding P_0, and thus the resulting plan will be equivalent to P_1.

B.3 Proof of Search Space Correspondence $\mathcal{L}$

Mapping Lemma: Let $U_0 =< \theta_0, \prec_{u0}>$ be an unambiguous plan and let $U_1 =< \theta_1, \prec_{u1}>$ be a 1-step extension of U_0 with respect to UA. If $T_1 =< \theta_1, \prec_{t1}>$ is a linearization of U_1 then there exists a plan T_0 such that T_0 is a linearization of U_0 and T_1 is a 1-step extension of T_0 with respect to TO.

Proof: By the definition of UA, $\theta_1 = \theta_0 \cup \{O_{add}\}$, where O_{add} added some c that is a precondition of some plan step O_{need} that is necessarily false in U_0. Hence, $T_0 =< \theta_0, \prec_{t0}>$ is a linearization of U_0, where $\prec_{t0}= \{(O_i, O_j) \mid (O_i, O_j) \in \prec_{t1} \text{ and } O_i, O_j \neq O_{add}\}$; that is, T_0 is the result of removing O_{add} from T_1. Using the TO-Extension lemma, we can show that T_1 is a 1-step extension of T_0. First, since U_0 and T_0 must have the same set of goals and UA selected goal c in expanding U_0, TO will select c in extending T_0. Second, O_{add} adds c. Third, O_{add} is before O_{need} in T_1, since O_{add} is before O_{need} in U_1 (by definition of UA) and T_1 is a linearization of U_1. Finally, O_{add} is after the last deleter of c, O_{del}, in T_1, since O_{add} is after O_{del} in U_1 (by definition of UA) and T_1 is a linearization of U_1. *Q.E.D.*

Totality Property For every plan U in $tree_{ua}$, there exists a non-empty set $\{T_1, \ldots, T_m\}$ of plans in $tree_{to}$ such that $\mathcal{L}(U) = \{T_1, \ldots, T_m\}$.

Proof: It suffices to show that if plan $U_1 =< \theta_1, \prec_{u1}>$ exists at depth d in $tree_{ua}$ and $< \theta_1, \prec_{t1}>$ is a linearization of U_1, then a plan $T_1 \simeq< \theta, \prec_{t1}>$ exists at depth d in $tree_{to}$.
Base case: The statement trivially holds for depth 0.
Induction step: Under the hypothesis that the statement holds for depth n, we now prove that the statement holds for depth $n+1$. Suppose that $U_1 =< \theta_1, \prec_{u1}>$ exists at depth $n+1$ in $tree_{ua}$ and $< \theta_1, \prec_{t1}>$ is a linearization of U_1. Let U_0 be the parent of U_1; thus, U_1 is a 1-step extension of U_0 with respect to UA. By the Mapping lemma, there exists a plan T_0 such that T_0 is a linearization of U_0 and $< \theta_1, \prec_{t1}>$ is an extension of T_0 with respect to TO. By the induction hypothesis, T_0 exists at depth n in $tree_{to}$. Therefore, by the Existence Lemma, a plan $T_1 \simeq< \theta_1, \prec_{t1}>$ exists at depth $n+1$ in $tree_{to}$. *Q.E.D.*

Disjointness Property: $\mathcal{L}$ maps distinct plans in $tree_{ua}$ to disjoint sets of plans in $tree_{to}$; that is, if $U_1, U_2 \in tree_{ua}$ and $U_1 \neq U_2$, then $\mathcal{L}(U_1) \cap \mathcal{L}(U_2) = \{\}$.

Proof: By the definition of $\mathcal{L}$, if $T_1, T_2 \in \mathcal{L}(U)$, then T_1 and T_2 are at the same tree depth d in $tree_{to}$; furthermore, U is also at depth d in $tree_{ua}$. Hence, it suffices to prove that if plans U_1 and U_2 are at depth d in $tree_{ua}$ and $U_1 \neq U_2$, then $\mathcal{L}(U_1) \cap \mathcal{L}(U_2) = \{\}$.
Base case: The statement vacuously holds for depth 0.
Induction step: Under the hypothesis that the statement holds for plans at depth n, we prove, by contradiction, that the statement holds for plans at depth $n+1$. Suppose that there exist two distinct plans, $U_1 =< \theta_1, \prec_1>$ and $U_2 =< \theta_2, \prec_2>$, at depth $n+1$ in $tree_{ua}$ such that $T \in \mathcal{L}(U_1) \cap \mathcal{L}(U_2)$. Then (by definition of $\mathcal{L}$), $parent(T) \in \mathcal{L}(parent(U_1))$ and $parent(T) \in \mathcal{L}(parent(U_2))$. Since $parent(U_1) \neq parent(U_2)$ contradicts the induction hypothesis, suppose that U_1 and U_2 have the same parent U_0. Thus, U_1 and U_2 are distinct 1-step extensions, with respect to UA, of the same (parent) plan. There are two cases to consider: either *(i)* $\theta_1 \neq \theta_2$ or *(ii)* $\theta_1 = \theta_2$ and $\prec_1 \neq \prec_2$. In the first case, since the two

plans do not contain the same set of plan steps, they have disjoint linearizations, and hence, $\mathcal{L}(U_1) \cap \mathcal{L}(U_2) = \{\}$, which contradicts the supposition. In the second case, $\theta_1 = \theta_2$; hence, both plans resulted from adding plan step O_{add} to the parent plan. Since $\prec_1 \neq \prec_2$, there exists a plan step O_{int} that interacts with O_{add} such that in one plan O_{int} is ordered before O_{add} and in the other plan O_{add} is ordered before O_{int}. Thus, in either case, the linearizations of the two plans are disjoint, and hence, $\mathcal{L}(U_1) \cap \mathcal{L}(U_2) = \{\}$, which contradicts the supposition. Therefore, the statement holds for plans at depth $n+1$. *Q.E.D.*

B.4 Completeness Proof for TO

We now prove that TO is complete under a breadth first search control strategy. Given an arbitrary solvable problem, there must exist a compact solution. (This follows from the definition of compactness.) Consequently, to prove that TO is complete under breadth-first search, it suffices to prove that a compact solution to the problem exists in $tree_{to}$. Before doing so, we first prove the following lemma.

Subplan Lemma: Let totally-ordered plan T_0 be strict subplan of a compact solution T_s. Then there exists a plan T_1 such that T_1 is a subplan of T_s and is a 1-step extension of T_0 with respect TO.

Proof: Since T_0 is a strict subplan of T_s and T_s is a compact solution, the set of false preconditions in T_0, G, must be non-empty. Let $c = goal - select(G)$, let O_{need} be the step in T_0 with precondition c, and let O_{add} be the step in T_s that achieves c. Consider the totally ordered plan $T_1 = < \theta_0 \cup \{O_{add}\}, \prec_1 >$, where $\prec_1 \subset \prec_s$. Clearly, T_1 is a subplan of T_s. Furthermore, by the TO-Extension Lemma, T_1 is a one-step extension of T_0 by TO. To see this, note that O_{add} is ordered before O_{need} in T_1, since it is ordered before O_{need} in T_s. Similarly, O_{add} is ordered after the last deleter of c in T_0, since any deleter of c in T_0 is a deleter of c in T_s, and O_{add} is ordered after the deleters of c in T_s. Thus, the conditions of the TO-Extension Lemma hold. *Q.E.D.*

TO Completeness Theorem: If $< \theta_s, \prec_s >$ is a totally-ordered compact solution, then some plan $T_s \simeq < \theta_s, \prec_s >$ is a member of $tree_{to}$.

Proof: It is straightforward to show that a plan with j steps can only exist at depth $j-2$ in $tree_{to}$. Let k be the cardinality of θ_s. Then, if $T_s \simeq < \theta_s, \prec_s >$ exists in $tree_{to}$, it must be depth $k-2$. To prove our result, it suffices to show that that for all $d \leq k-2$, there exists a plan at depth d that is a subplan of T_s. Note that any subplan of T_s at depth $k-2$ must be equivilent to T_s.

Base case: The root plan of $tree_{to}$ (the empty plan) is a subplan of T_s.

Induction step: Assume that the statement holds for plans at depth n, where $n < k-2$. Then there exists a plan T_0 at depth n that is a strict subplan of T_s. By the Subplan Lemma, there exists a plan $< \theta_1, \prec_1 >$ that is both a subplan of T_s and a 1-step extension of T_0 with respect to TO. By the Existence Lemma, $T_1 \simeq < \theta_1, \prec_1 >$ is a child of T_0. Thus there exists a subplan of T_s at depth $n+1$. *Q.E.D.*

B.5 Completeness Proof for UA

We now prove that UA is complete under a breadth-first search strategy. The result follows from the search space correspondence defined by $\mathcal{L}$ and the fact that TO is complete. In particular, we show below that for every totally ordered plan T in $tree_{to}$, there exists a plan U in $tree_{ua}$ such that T is a linearization of U. Since UA produces only unambiguous plans, it must be the case that if T is a solution, U is also a solution. From this, it follows immediately that UA is complete.

Inverse Mapping Lemma: Let $T_0 =< \theta_0, \prec_{t0}>$ be a totally-ordered plan. Let $T_1 =< \theta_1, \prec_{t1}>$ be a 1-step extension of T_0 with respect to TO. Let $U_0 =< \theta_0, \prec_{u0}>$ be a plan produced by UA such that T_0 is a linearization of U_0. Then there exists a plan U_1 such that T_1 is a linearization of U_1 and U_1 is a 1-step extension of U_0 with respect to UA.

Proof: By the definition of TO, $\theta_1 = \theta_0 \cup \{O_{add}\}$, where O_{add} added some c that is a false precondition of some plan step O_{need} in U_0. Consider $U_1 =< \theta_1, \prec_{u1}>$ where $\prec_{u1}$ is the minimal subset of $\prec_{t1}$ such that:

- $\prec_0 \subseteq \prec_1$, and
- $(O_{add}, O_{need}) \in \prec_1$, and
- $(O_{del}, O_{add}) \in \prec_1$, where O_{del} is the last deleter of c in U_1, and
- $(O', O'') \in \prec_1$ if O' and O'' would interact if they were in parallel.

Since $\prec_{u1} \subseteq \prec_{t1}$, T_1 is a linearization of U_1. In addition, U_1 is an extension of U_0, since it meets the conditions of the UA-Extension Lemma, as follows. First, since c must have been the goal selected by TO in extending T_0, c must likewise be selected by UA in extending U_0. Second, O_{add} adds c, since O_{add} achieves c in T_0. Finally, by construction, $\prec_{u1}$ satisfies the "minimality" part of the fourth condition of the UA-Extension Lemma. All of the orderings required by the fourth condition exist in $\prec_{t1}$ by the definition of TO. *Q.E.D.*

UA Completeness Theorem: Let T_s be a totally-ordered compact solution. Then an unambiguous plan U_s exists in $tree_{ua}$ such that T_s is a linearization of U_s.

Proof: Since TO is complete, it suffices to show that if a plan T_1 exists at depth d in $tree_{to}$, then a plan U_1 exists depth d in $tree_{ua}$ such that T_1 is a linearization of U_1.

Base case: The statement trivially holds for depth 0.

Induction step: Under the hypothesis that the statement holds for depth n, we now prove that the statement holds for depth $n+1$. Assume $T_1 =< \theta_1, \prec_{t1}>$ exists at depth $n+1$ in $tree_{to}$ and let $T_0 =< \theta_0, \prec_{t0}>$ be the parent of T_1. Thus, T_1 is a 1-step extension of T_0 with respect to TO. By the induction hypothesis, there exists a plan U_0 at depth n in $tree_{ua}$ such that T_0 is a linearization of U_0. By the Inverse Mapping Lemma, $< \theta_1, \prec_{u1}>$ is both a linearization of T_1 and a one-step extension of U_0 with respect to UA. Therefore, by the Existence Lemma, there exists a plan $U_1 \simeq< \theta_1, \prec_{u1}>$ that is a child of U_0 in $tree_{ua}$. *Q.E.D.*

References

[1] D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, Vol. 32.

[2] T. Dean and M. Boddy. Reasoning About Partially Ordered Events. *Artificial Intelligence*, Vol. 36.

[3] M. Drummond and K.W. Currie. Goal-ordering in Partially Ordered Plans. *Proceedings of IJCAI-89*, Detroit, MI.

[4] M. Drummond. Situated Control Rules. *Proceedings of the Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Canada.

[5] J. Hertzberg and A. Horz. Towards a Theory of Conflict Detection and Resolution in Nonlinear Plans. *IJCAI-89*, Detroit, MI.

[6] D. McAllester and D. Rosenblitt. Systematic Nonlinear Planning. *AAAI-91*, Anaheim, CA.

[7] S. Minton, J.G. Carbonell, C.A. Knoblock, D.R. Kuokka, O. Etzioni and Y. Gil. Explanation-Based Learning: A Problem-Solving Perspective, *Artificial Intelligence*, Vol. 40.

[8] E. Sacerdoti. *A Structure for Plans and Behavior.* American Elsevier, New York.

[9] S. Soderland and D.S. Weld. Evaluating Nonlinear Planning. Technical report 91-02-03, Univ. of Washington, Computer Science Dept.

[10] A. Tate. Interplan: A Plan Generation System Which Can Deal With Interactions Between Goals. Univ. of Edinburgh, Machine Intelligence Research Unit Memo MIP-R-109.

[11] A. Tate. Generating Project Networks. In *Proceedings of IJCAI-77*, Boston, MA.

[12] R. Waldinger. Achieving Several Goals Simultaneously. SRI AI Center Technical Note 107, SRI, Menlo Park, CA.

[13] M.M. Veloso, M.A. Perez and J.G. Carbonell. Nonlinear Planning with Parallel Resource Allocation. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, San Diego, CA.

[14] D. Warren. Warplan: A System for Generating Plans. Memo 76, Computational Logic Dept., School of AI, Univ. of Edinburgh.