Commitment Strategies in Planning: A Comparative Analysis

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

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.*

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 sonic 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, we consider 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 stepB 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 and ordering selection. As used in step 4, the *last deleter* of a precondition *c* for a step 0_{need} is a step O_{del} before 0_{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 0_{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 (Minton *et al.,* 1991).

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 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.

```
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 $0_{ADD}$ be an operator in the
   Library that adds c. If there is no such $0_{ADD}$, then ter-
   minate and report failure. Backtrack point: all such
   operators must be considered for completeness,
4. Ordering selection: Let Odd be the last deleter of c.
   Insert $0_{add}$ somewhere between $0_{del}$ and $0_{need}$, 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

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_{t0} 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}. 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, *Oneed 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 nodes actually visited under different search strategies.

[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.

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

```
UA(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 ter-
     minate 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 $0_{add}$ after $O_{del}$ and before $O_{need}$. Repeat until
     there are no interactions:
         o Select a step $O_{int}$ that interacts with $O_{add}$.
         o Order $O_{int}$ either before or after $0_{add}$.
           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

5 Search Space Comparison

Recall that a search space can be characterized as a tree. We denote the search space of TO by $tree_{to}$, and similarly the search space of UA by $tree_{ua}$. We show that for any given problem, $tree_{to}$ has at least as many nodes as $tree_{ua}$. This is done by proving the existence of a function $\mathcal{L}$ which maps nodes in $tree_{ua}$ to sets of nodes in $tree_{to}$ that satisfies the following two conditions.

1. For every node u in $tree_{ua}$, there exists a non-empty set $\{t_1, \ldots, t_m\}$ of nodes in $tree_{to}$ such that $\mathcal{L}(u) = \{t_1, \ldots, t_m\}$.
2. $\mathcal{L}$ maps distinct nodes in $tree_{ua}$ to disjoint sets of nodes 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 nodes in the two search trees. For each node counted in $tree_{ua}$, we use $\mathcal{L}$ to count a non-empty set of nodes in $tree_{to}$. The first property of $\mathcal{L}$ means that every time we count a node in $tree_{ua}$, we count at least one node in $tree_{to}$; this implies that $| tree_{ua} | \leq \sum_{u \in tree_{ua}} | \mathcal{L}(u) |$. Of course, we must further show that each node counted in $tree_{to}$ is counted only once; this is guaranteed by the second property of $\mathcal{L}$, which implies that $\sum_{u \in tree_{ua}} | \mathcal{L}(u) | \leq | tree_{to} |$. Thus, the conjunction of the two properties implies that $| tree_{ua} | \leq | tree_{to} |$.

We can define a function $\mathcal{L}$ that has these two properties as follows. Let u be a node in $tree_{ua}$, let t be a node in $tree_{to}$, and let $parent$ be a function from a node to its parent node. Then $t \in \mathcal{L}(u)$ if and only if the plan at t is a linearization of the plan at 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 node u in $tree_{ua}$ to all linearizations which share common derivation ancestry. This is illustrated in figure 3, where for each node in $tree_{ua}$ a dashed line is drawn to the corresponding set of nodes in $tree_{to}$.

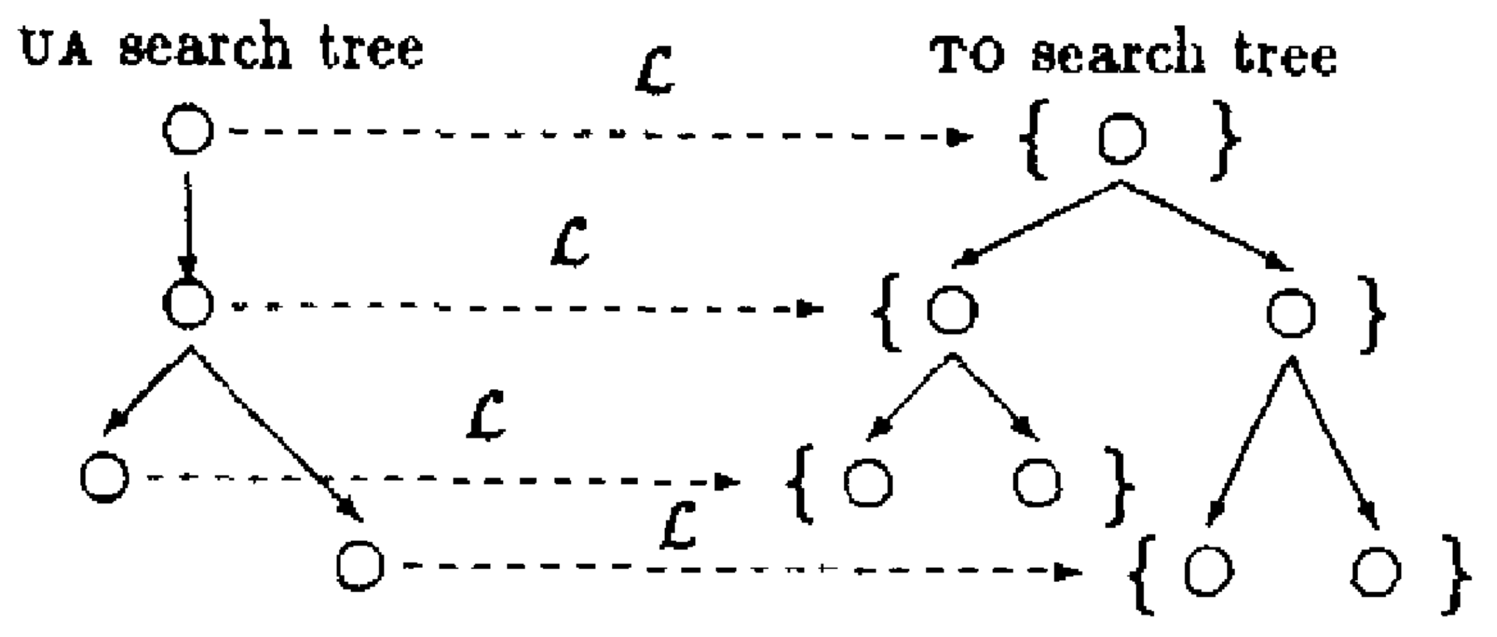

Figure 3: How C maps from $tree_{ua}$ to $tree_{to}$

We can show that C 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 *trveto*. 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 C maps nodes u_1 and U_2 at the same depth in $tree_{va}$ to disjoint sets of nodes in *treeto:* 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 at u_1 and U_2 are distinct (by the definition of UA), in which case their linearizations are disjoint.

How much smaller is $iree_{ua}$ than $tree_{io}$? The mapping described above provides an answer. For each node u in $tree_{ua}$ there are | L(u) | distinct nodes in TO, where | L(u) | is the number of linearizations of the plan contained at node u. The exact number depends on how unordered the plan at node u is. 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 $tree_{Ha}$ equals the size of $tree_{to}$ is when every plan in $tree_{ua}$ *is* totally ordered; otherwise, $tree_{ua}$ is strictly smaller than *treeto,* 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 node during search (discussed in this section) and the size of the subtree actually explored to find a plan (discussed below).

In this section we show that while UA can indeed take more time per node, the extra time is relatively small and grows only poly normally with the size of the plan. In our analysis, the size of the plan is simply the number of steps in the plan.[3] In comparing the relative efficiency of UA and TO, we first consider the number of times that each algorithm step is executed per node visited and then consider the time complexity of each step.

For both UA and TO, steps 1 and 2 are each executed once per node, and the number of executions of step 3 per node is bounded by a constant (the number of operators in the library). Analyzing the number of times step 4

[3] 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.

is executed might seem more complicated, since it may be executed many times at an internal node and is not executed at a leafnode. However, notice that a new node is generated each time step 4 is executed. Consequently, step 4 is executed once per node. Step 5 is also executed once per node since it always follows step 4, Thus, both algorithms execute steps 1, 2, 4, and 5 once per node, and step 3 is executed 0(1) times per node.

In examining the costs for each step, we first note that for both algorithms, steps 1, 2, and 3 can be accomplished in 0(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}, Bomewhere 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. This step can be accomplished in 0(e) time, where e is the number of edges in the graph required to represent the partially ordered plan (Minton *et* al., 1991), If n is the number of steps in the plan, then in the worst case, there may be $0(n^2)$ edges in the graph, and in the best case, *0(n)* edges. To carry out step 5 may require examining the entire plan, and thus, for UA, takes O(e) time and for TO, 0(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 0(e) time per node, whereas TO only requires *O(n)* time.

Step	Executions Per Node	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 Node Comparisons

7 Overall Efficiency Comparison

The previous sections compared TO and UA in terms of relative search space size and relative time cost per node. The extra processing time required by UA for each node would appear to be justified since its search space may contain exponentially fewer nodes. To complete our analysis, we must consider the number of nodes actually visited by each algorithm under a given search strategy.

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 node *u* in $tree_{ua}$, we denote the number of steps in the plan at *u* by n_u, and the number of edges by e_u. Then for each node *u* that UA generates, UA incurs time cost $0(e_u)$; whereas, TO incurs time cost $0(n_u) \cdot | L(u) |$, where $| L(u) \backslash$ is the number of linearizations of the plan at node 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.

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

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 node in a search tree is said to *succeed* if it is on a path to a solution, otherwise, it is said to *fail* When a node in UA'S search tree fails, all corresponding nodes for TO also fail. If a UA *node* succeeds, then *some* subset *of* the corresponding TO nodes succeed. If, under a depth-first strategy, UA and TO generate corresponding plans in the same order, then *(i)* for every failed node *u* that UA generates, TO generates all nodes in *C(u)* and *(it)* for every UA node *u* that succeeds, TO generates at least one node in 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. Nevertheless, the relative size of the subtree explored by TO under depth-first search can be *expected* to be proportional to the average size of | *C* |, since the relative size of TO's full search space is proportional to this value.

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 nodes 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 nodes. For example, on a representative problem, with a solution depth (and depth-bound) of eight, TO explored 8.0 times as many nodes 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 nodes. Although UA required more time per node, 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 nodes, and when relatively few steps interact, UA explores many fewer nodes 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 nodes 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. Although space does not permit analysis of this search strategy, we note that by using dependency-directed backtracking, To *will* behave almost identically to UA. Specifically, when TO backtracks to a node, a dependency analysis can indicate whether or not the failure below was independent of the ordering decision at that node. Of course, this dependency analysis increases the cost per node.

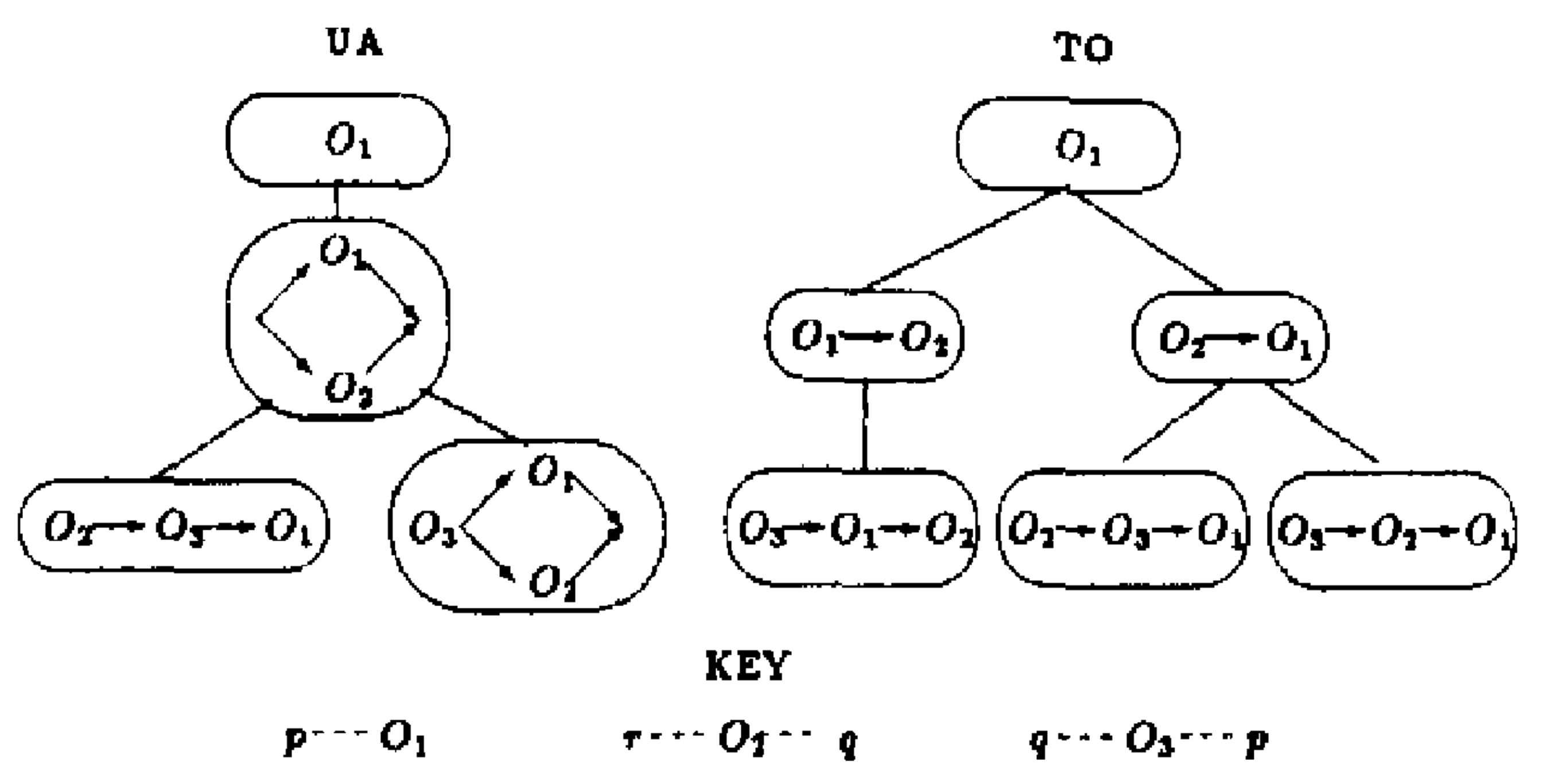

Figure 4: Comparison of UA and TO on an example.

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 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 4, 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 are not shown. Consider the node in $tree_{ua}$ containing the plan with unordered steps *01* and 02- When UA introduces *O3* to achieve precondition *p* of 01, step 4 of UA will order O3 with respect to *O2,* since these steps interact. However, it makes more sense to order O2 before O3, since *O2* achieves precondition *q* of O3. 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 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 *0* and *0%.* Due to this inability to postpone an ordering decision, TO must choose arbitrarily between the plans *0* $\to 0_2$ and *O2* $\to$ *01,* before the impact of this decision can be evaluated.

In the genera] case, UA is more informed than TO by any heuristic *h* that satisfies the following property: for any UA node u and corresponding To node t, *h(u)* $\succeq$ *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.

```
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 $O_{need}$ he
    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 ter-
    minate and report failure. Backtrack point: alt 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 $O_{need}$.
          0 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 5: A Propositional Planner based on the MTC

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 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), NowLin (Tate, 1977), and Tweak (Chapman, 1987), for example. How do these less-commit ted 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.[4] See figure 5 for a propositional planner, MT, based on Chapman's (1987) Modal Truth Criterion, the formal statement that char*acUrizes* 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 *C* elaborated

[4]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.

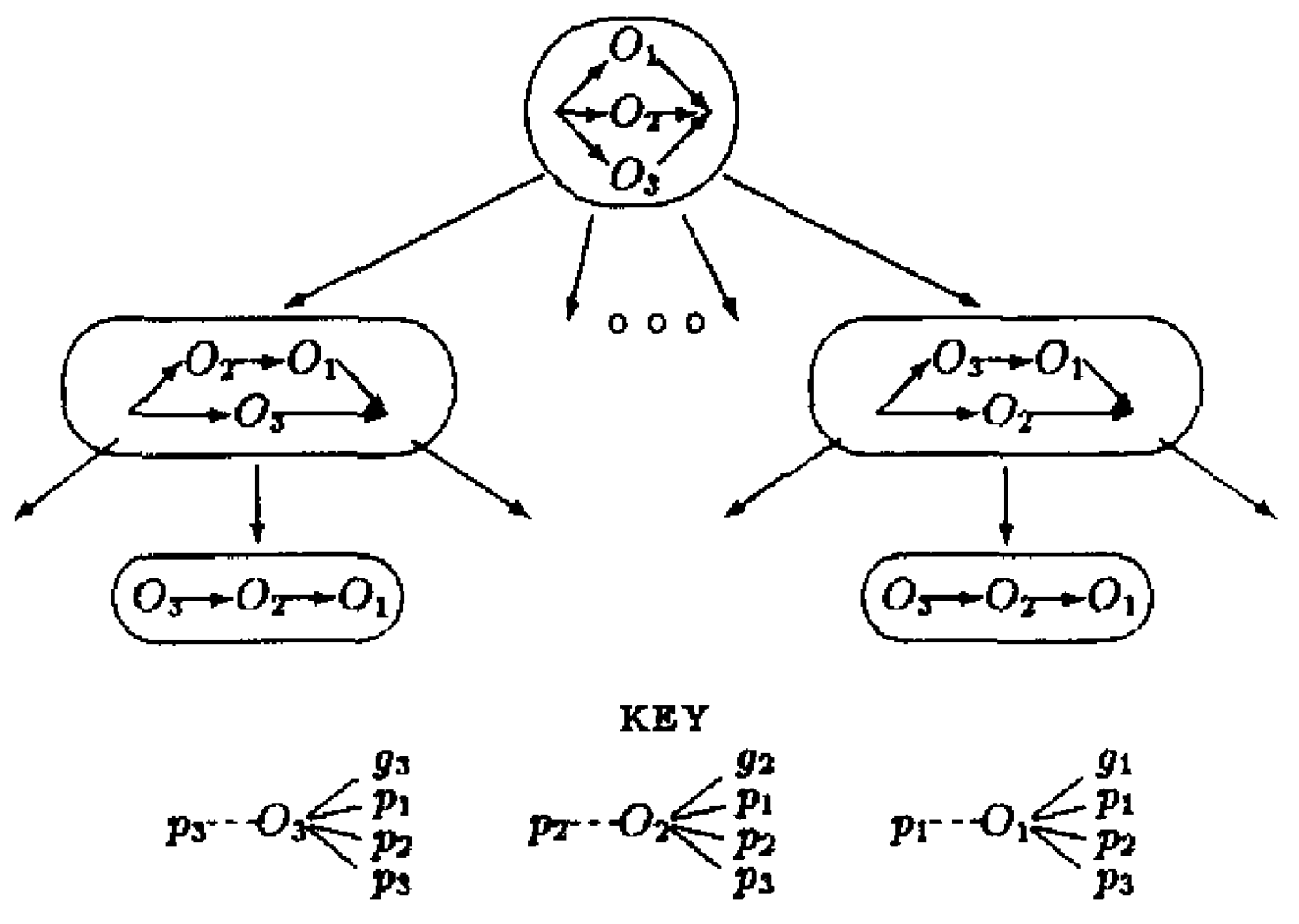

Figure 6: "Overlapping" plans.

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 nodes than TO (and consequently UA) on certain problems. The second property guarantees that UA does not generate "overlapping" plans. The example in figure 6 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, 0_1, 0_2, and 03, where each 0_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. The node at the top of the figure contains a plan constructed by MT where 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 $O_3 \longrightarrow$ -$_2$ $\longrightarrow$ 0i (among others). In comparison, both TO and UA only generate the plan $O_3 \longrightarrow O_2 \longrightarrow 0_1$ once. In fact, it is simple to show that, under breadth-first search, MT generates many more nodes than TO on this example, due to redundancy (and also more than UA, by transitivity).

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 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, partially ordered plans with linearization overlap should be avoided.

10 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.[5]

[6]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.

We have shown that the search space of a partial-order planner, UA, is *never* larger than the search space of a 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 increment per node 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-Uke 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.

What about more expressive operator languages? We have defined TO and UA to use propositional operators, but many problems demand operators with variables, conditional effects, or conditional preconditions. UA and TO can be extended to use such operators 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. For example, if library operators have variables, but are fully instantiated during the operator selection step, then our analysis holds without modification. The work required to demonstrate step interaction tends to increase with the expressiveness of the operator language used (Dean &. Boddy, 1988; Hertzberg & Horz, 1989); nevertheless, we believe the cost of detecting interactions can often be kept low by using a conservative definition of step interaction.

The general lesson from this work is that partial-order planning *can* be better than total-order planning, but this isn't necessarily so. 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.

Acknowledgements: The TORPOR group thanks Mark Boddy for helping start this project, and Andy Philips, for implementing and helping design the algorithms. Thanks also to our various reviewers.

A Proof of Correspondence $\mathcal{L}$

Definition: 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 $(O_1, O_2) \in \prec$.

Definition: A plan P' is a *1-step extension* of a plan P with respect to a planning algorithm, if P' can be produced from P in one invocation of the algorithm.

Property 1: For every node u in $tree_{ua}$, there exists a non-empty set $\{t_1, \ldots, t_m\}$ of nodes in $tree_{to}$ such that $\mathcal{L}(u) = \{t_1, \ldots, t_m\}$.

Proof: It suffices to show that if a plan P is contained in a node u at depth d in $tree_{ua}$, then each linearization of P is contained in some node t at depth d in $tree_{to}$, such that $t \in \mathcal{L}(u)$.
Base case: The statement trivially holds for depth 0.
Induction step: Under the hypothesis that the statement holds for depth n, we now prove, by contradiction, that the statement holds for depth $n+1$. Suppose that there exists a node u with plan $P_u = < \theta, \prec_u >$ at depth $n+1$ in $tree_{ua}$ such that $P_t = < \theta, \prec_t >$ is a linearization of P_u and there is no node t containing P_t at depth $n+1$ in $tree_{to}$ such that $t \in \mathcal{L}(u)$. Let $P'_u = < \theta', \prec'_u >$ be at node u', the parent of u; *i.e.*, P_u is a 1-step extension of P'_u with respect to UA. By the definition of UA, $\theta = \theta' \cup \{O_{add}\}$, where O_{add} added some c that is a precondition of some plan step O_{need} that is necessarily false in P'_u. Hence, $P'_t = < \theta', \prec'_t >$ is a linearization of P'_u, where $\prec'_t = \{(O_i, O_j) \mid (O_i, O_j) \in \prec_t \text{ and } O_i, O_j \neq O_{add}\}$; that is, P'_t is the result of removing O_{add} from P_t. Thus, by the induction hypothesis, there exists a node t' at depth n in $tree_{to}$ that contains P'_t, such that $t' \in \mathcal{L}(u')$.

For the supposition to hold, it must be the case that P_t is not a 1-step extension of P'_t with respect to TO. Since c is necessarily false in P'_u, it is false in P'_t, one of its linearizations. Likewise, since c is necessarily true in P_u, it is also true in P_t. Let O_{del} be the last deleter of c in P'_t. Then, since P'_t is a linearization of P'_u, O_{del} must be a last deleter of c in P'_u. Therefore, in P_u (by definition of UA), and hence in P_t (by definition of linearization), O_{del} is ordered before O_{add}, which is ordered before O_{need}. Hence, P_t is a 1-step extension of P'_t with respect to TO, which contradicts the supposition. Therefore, the statement holds for nodes at depth $n+1$. *Q.E.D.*

Property 2: $\mathcal{L}$ maps distinct nodes in $tree_{ua}$ to disjoint sets of nodes 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 nodes 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 depth n, we prove, by contradiction, that the statement holds for nodes at depth $n+1$. Suppose that there exist two distinct nodes, u_1 and u_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. Let $< \theta_1, \prec_1 >$ be the plan at u_1, and let $< \theta_2, \prec_2 >$ be the plan at u_2. These two plans 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 nodes at depth $n+1$. *Q.E.D.*

References

[Chapman, 1987] D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence,* Vol. 32.

[Dean & Boddy, 1988] T. Dean and M, Boddy. Reasoning About Partially Ordered Events. *Artificial Intelligence,* VoL 36.

[Drummond & Currie, 1989] M. Drummond and K.W. Currie. Goal-ordering in Partially Ordered Plana. *Proceedings of IJCAI-89,* Detroit, MI.

[Drummond, 1989] M. Drummond. Situated Control Rules. *Proceedings of the Conference on Principles of Knowledge Representation and Reasoning,* Toronto, Canada.

[Hertzberg & Horz, 1989] J. Hertzberg and A, Horz. Towards a Theory of Conflict Detection and Resolution in Nonlinear Plans. *IJCAI-89,* Detroit, MI.

[Minton *et al,* 1989] S. Minton, J.G. Carbonell, CA. Knoblock, D.R. Kuokka, O. Etzioni and Y. Gil Explanation-Based Learning; A Problem-Solving Perspective, *Artificial Intelligence,* VoL 40.

[Minton *et al,* 1991] S. Minton, J. Bresina, M. Drummond, and A. Philips. An Analysis of Commitment Strategies in Planning: The Details, Technical report 91-08, NASA Ames, AI Research Branch.

[Sacerdoti, 1977] E. Sacerdoti. *A Structure for Plans and Behavior* American Elsevier, New York.

[Soderland & Weld, 1991] S. Soderland and D.S. Weld. Evaluating Nonlinear Planning. Technical report 91-02-03, Univ. of Washington, Computer Science Dept.

[Tate, 1974] 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.

[Tate, 1977] A. Tate. Generating Project Networks. In *Proceedings of IJCAI-77,* Boston, MA.

[Waldinger, 1975] R, Waldinger. Achieving Several Goals Simultaneously. SRI AI Center Technical Note 107, SRI, Menlo Park, CA.

[Veloso *et al,* 1990] 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.

[Warren, 1975] D. Warren. Warplan: A System for Generating Plans, Memo 76, Computational Logic Dept., School *of* AI, Univ. *of* Edinburgh.