Analyzing Plans with Conditional Effects

Elly Winner and Manuela Veloso
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
{elly.veloso}@cs.cmu.edu
fax: (412) 268-4801

Abstract

Several tasks, such as plan reuse and agent modeling, rely on interpreting a given or observed plan to generate the underlying plan rationale. Although there are several previous methods that successfully extract plan rationales, they do not apply to complex plans, in particular to plans with actions that have conditional effects. In this paper, we introduce SPRAWL, an algorithm to find a minimal annotated partially ordered structure that maximizes a given evaluation function for an observed totally ordered plan with conditional effects. The algorithm proceeds in a two-phased approach, first preprocessing the given plan using a novel needs analysis technique that builds a needs tree to identify the dependencies between operators in the totally ordered plan. The needs tree is then processed to construct a partial ordering that captures the complete rationale of the given plan. We also provide a polynomial-time algorithm to find non-optimal minimal annotated partial orderings of observed totally ordered plans with conditional effects. We provide illustrative examples and discuss the challenges we faced.

Introduction

Analyzing example plans and executions is crucial for plan adaptation and reuse, e.g., (Fikes, Hart, & Nilsson 1972), and could be useful for plan recognition and agent modeling, e.g., (Kautz & Allen 1986). One of the most common approaches to plan analysis has been to create an annotated ordering of the example plan, e.g., (Fikes, Hart, & Nilsson 1972; Regnier & Fade 1991; Kambhampati 1989; Kambhampati & Hendler 1992; Veloso 1994), in which an ordered plan is supplemented with a rationale for the ordering constraints. Annotated orderings allow systems not only to reuse more flexibly portions of the plans they have observed, but also to reuse the reasoning that created those plans in order to solve new problems.

In recent years, the focus of the planning and agent modeling community has shifted from the simple STRIPS domain-specification language (Fikes & Nilsson 1971) toward richer languages like ADL (Pednault 1986) that capture the conditional effects of real-world actions. Despite the success of the annotated ordering approach for simple domain-specification languages, it has not been applied to plans with conditional effects.

In this paper, we introduce the SPRAWL algorithm for finding the rationale behind an observed totally ordered plan: the purpose for which each step is used in the plan and the reason behind each of the ordering constraints. We store this information in a structure we call a minimal annotated consistent partial ordering (MACPO). A consistent partial ordering \( P \) of a totally ordered plan \( T \) is one in which all relevant effects (those which affect the fulfillment of the goal active in \( P \)) are also active in \( T \). We call the partial orderings found by SPRAWL minimal because they do not include extraneous ordering constraints; each constraint either:

- provides a term upon which a relevant effect depends, or
- prevents a threat to such a term.

Finally, SPRAWL annotates each ordering constraint with the term the constraint provides or protects. Given an evaluation function for partial order quality, SPRAWL is capable of identifying the optimal MACPO of an observed total order.

We assume that we are given or that we observe a plan that is valid, i.e., all preconditions of the steps are satisfied, and, when executed, the plan produces the goal state. SPRAWL links the steps of the plan through the literals or terms that they support. Partial orderings are capable of representing these dependencies. In additional, partial orderings can isolate independent sub-plans that can be reused or recognized separately, and they also identify potential parallelism.

We assume that observed example plans are totally ordered as plans of single executors. The annotations on the ordering constraints should explain the rationale behind the plans and allow portions of them easily to be matched, removed, and used independently.

Conditional effects make the task much more difficult because they cause the effects of a given step to change depending on what steps come before it, thus making step be-

\(^1\)A partial order is a precedence relation \( \preceq \) with the following three properties 1) reflexivity: \( a \preceq a \); 2) non-symmetric (no cycles): if \( a \preceq b \) then not \( b \preceq a \), unless \( a = b \); and 3) transitivity: if \( a \preceq b \) and \( b \preceq c \), then \( a \preceq c \). The relation is a “partial” order because there may be incomparable elements: i.e., elements \( a, b \) such that neither \( a \preceq b \) nor \( b \preceq a \). Note that a DAG is a partial order if we define \( a \preceq b \) as a path from \( a \) to \( b \).
behavior difficult to predict. In fact, any ordering must treat each conditional effect in the plan in one of three ways:

- **Use**: make sure the effect occurs;
- **Prevent**: make sure the effect does not occur;
- **Ignore**: don’t care whether the effect occurs or not.

Figure 1 illustrates three totally ordered plans that demonstrate these cases. Note that all three plans have the same initial state and the same operators. We are able to demonstrate all three cases by changing only the goals. The preconditions (pre) are listed, as are the effects, which are represented as conditional effects \{a\} \rightarrow b, i.e., if a then add b. A non-conditional effect that adds a literal b is then represented as \{\} \rightarrow b. Delete effects are represented as negated terms (e.g., \{a\} \rightarrow NOT b). In the first plan, the conditional effect of \text{op1} is used to generate the goal term c. In the second plan, it is prevented from generating the term c, and in the third plan, the effect is irrelevant, so it is ignored.

Figure 2 shows the annotated partial orderings generated by \text{SPRAWL} for each of these cases. The ordering constraints are annotated with a rationale explaining why they are necessary. Although the plans for these three cases are composed of the same steps, \text{SPRAWL} reveals that the partial orderings are very different. In the “use” case, \text{SPRAWL} identifies that \text{op2} threatens the goal term c, which is created by \text{op1}, and enforces the ordering \text{op1} \rightarrow \text{op2} to protect c. In the “prevent” case, \text{SPRAWL} identifies that the step \text{op1} must not be able to execute the conditional effect that adds the term c, and so ensures that the condition of this effect, the term b, is not true before the step executes. In this way, \text{SPRAWL} finds the ordering constraint \text{op2} \rightarrow \text{op1}. It also finds that the \text{START} step, since it adds b, is a threat to this link, and must therefore come before \text{op2}. Finally, \text{SPRAWL} identifies that, in the “ignore” case, the conditional effect is irrelevant, so \text{op1} and \text{op2} may run in parallel.

Treating any conditional effect in a plan in a different way will result in a different partial ordering, creating exponentially (in the number of conditional effects) many partial orderings, many of which may be invalid. One way to deal with this difficulty is to insist that exactly the same conditional effects must be active in the partial ordering as are active in the totally ordered plan, but this will result in an overly restrictive partial ordering in which some ordering constraints may not contribute to goal achievement. Instead, we analyze the totally ordered plan to discover which conditional effects are relevant. This allows us to ignore incidental conditional effects in the totally ordered plan.

Instead of finding the optimal partially ordered plan to solve a given problem, we chose to focus on finding optimal partial orderings consistent with given totally ordered plan, or those in which all relevant effects were also active in the total ordering. There are two reasons for this. The first is that the totally ordered plan contains a wealth of valuable information about how to solve the problem, including which operators to use and which conditional effects are relevant. Using this information reduces the search required to solve the problem. The second is that for many applications, including plan modification and reuse and agent modeling, it is important to be able to analyze an observed or previously generated plan (for example, to find characteristic patterns of behavior or to identify unnecessary steps).

The remainder of this paper is organized as follows. We first discuss related work in plan analysis. We then introduce the needs analysis technique, illustrate its behavior and discuss its complexity. Next, we explain how the \text{SPRAWL} algorithm uses needs analysis to find a partial ordering and

---

\[2\text{The number of possible partial orderings is also exponential in the number of steps and the number of conditions on each step.}\]
discuss the complexity of the entire algorithm. We then discuss the limitations and capabilities of the algorithm, present a suboptimal polynomial-time solution, and present our conclusions.

Related Work

Many researchers have addressed the problems of annotating orderings and of finding partially ordered plans. We discuss a selection of the research investigating annotation and partial ordering.

Triangle tables are one of the earliest forms of annotation (Fikes, Hart, & Nilsson 1972). In this approach, totally ordered plans are expanded into triangle tables that display which add-effects of each operator remain after the execution of each subsequent operator. From this, it is easy to compute which operators supply preconditions to other operators, and thus to identify the relevant effects of each operator and why they are needed in the plan. Fikes, Hart, and Nilsson used triangle tables for plan reuse and modification. The annotations help to identify which sub-plans are useful for solving the new problem and which operators in these sub-plans are not relevant or applicable in the new situation.

Regnier and Fade alter the calculation of the triangle table by finding which add-effects of each operator are needed by subsequent operators (instead of which add-effects remain after the execution of subsequent operators) (Regnier & Fade 1991). They use the dependencies computed in this modified triangle table to create a partial ordering of the totally ordered plan.

The triangle table approach has been applied only to plans without conditional effects. When conditional effects are introduced, it is no longer obvious what conditions each operator “needs” in order for the plan to work correctly. Although we do not use the triangle table structure, our needs analysis approach can be seen as an extension of the triangle table approach to handle conditional effects.

Another powerful approach to annotation is the validation structure (Kambhampati 1989; Kambhampati & Hendler 1992; Kambhampati & Kedar 1994). This structure is an annotated partial order created during the planning process. Each partial order link is a 4-tuple called a validation: \( <e, t', c, t> \), where the effect \( e \) of step \( t' \) satisfies the condition \( C \) of the step \( t \). The validation structure acts as a proof of correctness of the plan, and allows plan modification to be cast as fixing inconsistencies in the proof. This approach is shown to be effective for plan reuse and modification (Kambhampati & Hendler 1992) and for explanation-based generalization of partially ordered and partially instantiated plans (Kambhampati & Kedar 1994). The approach has not been applied to plans with conditional effects. Although (Kambhampati 1989) presents an algorithm for using the validation structures of plans with conditional effects to enable modification and reuse, no method is presented for finding these structures. And since the structures are created during the planning process, no method is presented for finding validation structures of any observed plans, even those without conditional effects.

Derivational analogy (Veloso 1994) is another interesting approach to and use of annotation. In this approach, decisions made during the planning process are explicitly recorded along with the justifications for making them and unexplored alternate decisions. This approach has been shown to be effective for reusing not only previous plans, but also previous lines of reasoning. The approach can handle conditional effects, but, like the validation structure approach, is applicable only to plans that have been created and annotated by the underlying planner.

There has been some previous work on finding partial orderings of totally ordered plans. As previously mentioned, Regnier and Fade (Regnier & Fade 1991) used triangle tables to do this for plans without conditional effects. Veloso et al also presented a polynomial-time algorithm for finding a partial ordering of a totally ordered plan without conditional effects (Veloso, Pérez, & Carbonell 1990). The algorithm adds links between each operator precondition and the most recent previous operator to add the condition. It then resolves threats and eliminates transitive edges. However, Bäckström shows that this method is not guaranteed to find the most parallel partial ordering, and that, in fact, finding the optimal partial ordering according to any metric is NP-complete (Bäckström 1993).

There has been a great deal of research into generating partially ordered plans from scratch. UCPOP (Penberthy & Weld 1992) is one of the most prominent partial-order planners that can handle conditional effects. One of the strengths of UCPOP is its non-determinism; it is able to find all partially ordered plans that solve a particular problem.

Graphplan (Blum & Furst 1997), another well-known planner, is also able to find partially ordered plans in domains with conditional effects (Anderson, Smith,
Weld 1998). However, it produces suboptimal and non-minimal (overconstrained) partial orderings, which does not suit our purpose. Consider the plan in which the steps $op\_a_1\ldots op\_a\_n$ may run in parallel with the steps $op\_b_1\ldots op\_b\_n$. Graphplan would find the partial ordering shown in Figure 3 because it only finds parallelism within an individual time step. In the first time step, $op\_a\_1$ and $op\_b\_1$ may run in parallel, but there is no other operator that may run in parallel with them, so Graphplan moves to the second time step (in which $op\_a\_2$ and $op\_b\_2$ may run in parallel). Graphplan constrains the ordering so that no operators from one time step may run in parallel with operators from another. None of the ordering constraints between $op\_a$ steps and $op\_b$ steps help achieve the goal, so they are not included in the partial ordering created by S PRAWL, shown in Figure 4. S PRAWL reveals the independence of the two sets of operators.

Figure 3: This partial ordering, found by Graphplan, contains many irrelevant ordering constraints.

Figure 4: This partial ordering, found by S PRAWL, contains only necessary ordering constraints.

**Needs Analysis**

Needs analysis, the first step of the S PRAWL algorithm, computes a tree of needs for the totally ordered plan. We first create a goal step called FINISH with the terms of the goal state as preconditions. Needs analysis calculates which terms need to be true before the last step in the plan in order for the preconditions of FINISH to be true afterward. Then it calculates which need to be true before the second-to-last plan step in order for those terms to be true. This calculation is executed for each step of the plan, starting from the last step and finishing at the START step, creating a tree of “needs.” This needs tree allows us to identify the relevant effects of a given step and most of the dependencies in the plan. However, not all threats are identified in Needs Analysis; S PRAWL uses the needs tree to calculate the remaining threats.

**Needs Tree Structure**

In this section, we will discuss the needs that compose the needs tree as well as the structure of the tree. The needs tree consists of three kinds of needs:

1. **Precondition Needs:** the preconditions of a step are called precondition needs of the step—they must be true for the step to be executable. For example, the precondition needs of the FINISH step are the goals of the plan.

2. **Existence Needs:** terms that must be true before a step $n$ in order for $n$ to create a particular term or to maintain a previously existing term are called existence needs of the term at the step $n$. In the “use” example in Figure 2, one existence need of the term $c$ at the step $op_1$ is $b$, since $op_1$ will create $c$ if $b$ is true before it executes.

3. **Protection Needs:** terms that must be true before step $n$ in order for $n$ not to delete a particular term are called protection needs of the term at the step $n$. In the “prevent” example in Figure 2, a protection need of the term NOT $c$ at the step $op_1$ is NOT $b$, since if NOT $b$ is not true before step $op_1$, then $op_1$ will add $c$ (thereby deleting NOT $c$).

For the sake of simplicity, instead of abstract plan steps, we will illustrate the three kinds of needs using plan steps from a domain in which we have a sprinkler that, if on, can wet the yard as well as any object that may be in the yard. Figure 5 shows the operator $\text{sprinkle front-yard}$. The term on $\text{sprinkler}$ is a precondition need of the step $\text{sprinkle front-yard}$.

**sprinkle front-yard**

- pre: {} $\Rightarrow$ wet front-yard
- on sprinkler at ?obj front−yard $\Rightarrow$ wet ?obj

Figure 5: The step $\text{sprinkle front-yard}$.

To illustrate existence needs, let us assume that, after executing the step $\text{sprinkle front-yard}$, wet shoe must be true. This could be accomplished in two ways:

- by ensuring that at shoe front-yard, wet shoe was true before $\text{sprinkle front-yard}$ executed, or
- by ensuring that wet shoe was already true before $\text{sprinkle front-yard}$ executed, as shown in Figure 6. 

These two terms are called existence needs of wet shoe at the step $\text{sprinkle front-yard}$, since they provide ways for the term wet shoe to be true after the step $\text{sprinkle front-yard}$.

We must also make a distinction between maintain existence needs and create existence needs. As mentioned above, there are two ways to ensure that wet shoe is true

---

---

3In the remainder of the sprinkler examples, we abbreviate the literals sprinkler as sp, front-yard as fy, back-yard as by, and shoe as sh.

4Precondition needs and protection needs are always create needs.
after the execution of the step \textit{sprinkle front-yard}, both illustrated in Figure 6. One way is for \textit{wet shoe} to have been true previously. We call this a \textit{maintain} existence need since the step does not generate the term, but simply maintains a term that was previously true. However, the step \textit{sprinkle front-yard} could generate the term \textit{wet shoe} if at shoe front-yard were true before the step executed. We call this an \textit{create} existence need, since we have introduced a new need in order to satisfy another.

Note that, because there may be multiple ways to ensure the existence of a term, the description of needs must include the \textit{OR} logical operator, as shown in Figure 6. It must also include the \textit{AND} logical operator, since we allow a conditional effect to have multiple conditions, and in order to guarantee that the effect occurs, we must be able to specify that all must be true.

To illustrate \textit{protection needs}, assume that, after executing the step \textit{sprinkle front-yard}, the term \textit{NOT wet shoe} must be true. In order to protect the term \textit{NOT wet shoe}, we must ensure that \textit{NOT at shoe front-yard} is true before \textit{sprinkle front-yard} executes. This is called a \textit{protection need} because it protects the term from being deleted (i.e., prevents \textit{wet shoe} from being added).

It is not always necessary to generate new needs to satisfy a need term; it may also be satisfied if a non-conditional effect of the step satisfies it, as illustrated in Figure 7. We call such needs \textit{accomplished}, and indicate this in our diagrams with a double circle.

\textbf{Needs Analysis Algorithm}

The needs analysis algorithm is shown in Table 1. We now describe in detail how needs analysis generates the needs of an individual term. Each needed term \(t\) must be created and protected from deletion; we represent this as two branches of needs: existence needs and protection needs. As explained previously, \(t\)'s existence needs at a particular step \(n\) are terms which must be true before step \(n\) to ensure that \(t\) is true after step \(n\). There are two possibilities for existence needs: either \(t\) may have been true before step \(n\), or a conditional effect of step \(n\) may generate \(t\) \footnote{Non-conditional effects of step \(n\) that add \(t\) do not add needs—nothing needs to be true before step \(n\) in order for them to occur}. The protection needs of \(t\) at step \(n\) are terms which must be true before step \(n\) to ensure that step \(n\) does not delete \(t\). Prevention needs are therefore negated conditions of any conditional effects of step \(n\) that delete \(t\). Figure 8 illustrates the needs tree created to satisfy each needed term.

We will use the totally ordered plan from the sprinkler domain shown in Figure 9 to illustrate the behavior of the needs algorithm. First, the algorithm will analyze the last plan step (\textit{sprinkle front-yard}), which has one precondition need (\textit{on sprinkler}), to determine how to satisfy the needs of the subsequent step \textit{FINISH (wet shoe and wet front-yard)}. As previously discussed, there are two ways for the step \textit{sprinkle front-yard} to satisfy \textit{wet shoe}: either \textit{wet shoe} could be true before this step executes, or at shoe front-yard must be true before this step executes. So the needs of the term \textit{wet shoe} are \textit{maintain wet shoe OR create at shoe front-yard}. As for \textit{wet front-yard}, the other precondition need of the \textit{FINISH} step, it is accomplished by the step \textit{sprinkle front-yard} since it is a non-conditional effect of the step. However, the algorithm continues to look for other ways to accomplish the term. Since there are no conditional effects of \textit{sprinkle front-yard} that either generate or delete \textit{wet front-yard}, the algorithm just adds the maintain existence need, \textit{maintain wet fy}.

Next, the algorithm moves back to the previous plan step, \textit{move shoe back-yard front-yard}, which has the precondition need at shoe back-yard. The needs carried over from previous steps are \textit{maintain wet shoe OR create at shoe front-yard}, the existence needs of \textit{wet shoe} from the \textit{FINISH} step; \textit{maintain wet front-yard}, the existence need of \textit{wet front-yard} from the \textit{FINISH} step; and on sprinkler, the precondition need of the step \textit{sprinkle front-yard}. The term at \textit{shoe front-yard} is a non-conditional effect of this step, so it is accomplished. But, as with \textit{wet fy} in the previous step, the algorithm adds a maintain existence need \textit{(maintain at shoe front-yard)} in order to find other ways to accomplish the term. The terms \textit{maintain wet shoe}, \textit{maintain wet }}
Figure 8: The existence needs of a need at a particular step $n$ are calculated by finding all possible ways it can be generated in the previous step and ensuring that at least one of these occurs. The protection needs are calculated by finding all possible ways it can be deleted in the previous step and ensuring that none of these occurs.

Figure 9: A totally ordered plan in the sprinkler domain and its complete needs tree.
**Input:** A totally ordered plan \( T = S_1, S_2, \ldots, S_n \),
  the START operator \( S_0 \) with add effects set to the initial state, and the FINISH operator \( S_n + 1 \) with preconditions set to the goal state.

**Output:** A needs tree \( N \).

**procedure Needs_Analysis** \((T, S_0, S_n + 1)\):
1. for \( c \leftarrow n+1 \) down-to 1 do
   2. for each precond of \( S_c \) do
   3. Expand_Term(c, precond)

**procedure Expand_Term**\((c, \text{term})\):
4. Find_Existence\((c, \text{term})\)
5. Find_Protection\((c, \text{term})\)

**procedure Find_Existence**\((c, \text{term})\):
6. for each conditional effect of \( S_c \) do
7. if effect unconditionally adds term then
   8. term.accomplished \( \leftarrow \) true
9. otherwise if effect conditionally adds term then
10. Add_Conditions_To_Existence_Needs\((\text{effect}, \text{term})\)
11. for each condition of effect do
12. Expand_Term\( (c-1, \text{condition}) \)

**procedure Find_Protection**\((c, \text{term})\):
13. for each conditional effect of \( S_c \) do
14. if effect unconditionally deletes term then
15. term.impossible \( \leftarrow \) true
16. return
17. otherwise if effect conditionally deletes term then
18. Add_Conditions_To_Protection_Needs\((\text{effect}, \text{term})\)
19. for each condition of effect do
20. Expand_Term\( (c-1, \text{condition}) \)

| Table 1: Needs Analysis algorithm. |

front-yard, and on sprinkler cannot be prevented or created by this step, so each is satisfied by a maintain existence need (maintain wet shoe, maintain wet front-yard, and maintain on sprinkler).

Finally, the algorithm reaches the initial state, or START step, and is able to determine which branches of the needs tree can be accomplished and which cannot. The remaining branches of the tree are at shoe back-yard, maintain at shoe front-yard, maintain wet shoe, maintain wet front-yard, and maintain on sprinkler. Two of the needs, at shoe back-yard and maintain on sprinkler are accomplished by the START step. However, all of the other remaining needs are not accomplished by the START step. We call these needs unsatisfiable and indicate this in our diagrams with a dashed circle.

The complexity of needs analysis is exponential in the number of conditional effects and the bound on the number of conditions in each effect, or \( O(mP(\text{EC})^n) \), where \( m \) is the number of steps without conditional effects, \( n \) is the number of steps with conditional effects, \( P \) is the bound on the number of preconditions, \( E \) is the bound on the number of conditional effects in each step, and \( C \) is the bound on the number of conditions per conditional effect. The complexity of needs analysis on a plan with no conditional effects is linear: \( O(mP) \).

**The SPRAWL Algorithm**

Table 2 shows the SPRAWL partial ordering algorithm. SPRAWL performs needs analysis, then performs a depth-first search on the needs tree, adding causal links in the partial ordering between steps that need terms and the steps that generate them.

| Table 2: The SPRAWL algorithm. |

Resolving Threats

We rely heavily on the totally ordered plan to help us resolve threats. There are three ways to resolve threats in a plan with conditional effects, as described in (Weld 1994):
1. **Promotion** moves the threatened operators before the threatening operator;
2. **Demotion** moves the threatened operator after the threatening operator;
3. **Confrontation** may take place when the threatening effect is conditional. It adds preconditions to the threatening operator to prevent the effect causing the threat from occurring.

To find all possible partial orderings, all these possibilities should be explored. However, since we are provided the totally ordered plan, we do not need to search at all to find a feasible way to resolve the threat; we can simply resolve it in the same way it was resolved in the totally ordered plan. In fact, if threats are resolved in a different way, then the resulting partial ordering would not be consistent with the totally ordered plan.

If, in the totally ordered plan, the threatening operator occurs before the threatened operators, then promotion should be used to resolve the threat in the partial ordering. Similarly, if it occurs after the threatened operators, demotion should be used to resolve the threat in the partial ordering. If the threatening operator occurs between the threatened operators, then we know that confrontation must have been used in the totally ordered plan to prevent the threatening conditional effect from occurring.

Needs analysis takes care of confrontation with protection needs, shown in Figure 8, which ensure that steps that occur between a needed term’s creation and use in the totally ordered plan do not delete the term.

**Discussion**

The SPRAWL algorithm does not create a partially ordered plan from scratch; its purpose is to create an annotated partial ordering of the steps of a given totally ordered plan to aid in our understanding of the structure of the plan. Because of this, we restrict SPRAWL to partial orderings consistent with the totally ordered plan.

However, frequently, there are many partial orderings consistent with the totally ordered plan. SPRAWL searches through these possibilities to find the optimal partial ordering. Here, we discuss the space of possibilities explored by SPRAWL and discuss a polynomial solution for finding a suboptimal minimal annotated consistent partial ordering.

**Different Total Orderings of the Same Steps May Produce Different Partial Orderings**

In some cases, a different total ordering of the same plan steps would produce a different partial ordering, but these are cases in which the relevant effects differ. For example, the use and prevent cases shown in Figure 1 consist of the same initial states and the same operators. However, the relevant effects differ. SPRAWL would never produce the same partial ordering for both of them; the partial orderings would each preserve the same relevant effects as are active in the respective totally ordered plans. The minimal annotated consistent partial orderings found by SPRAWL are shown in Figure 2.

**Active Conditional Effects May Differ from Those in Totally Ordered Plan**

Though SPRAWL is restricted to partial orderings consistent with the totally ordered plan it is given, this does not mean that all conditional effects active in the totally ordered plan must be active in the partial ordering, or vice versa. There are sometimes irrelevant conditional effects in the totally ordered plan or in the partial ordering, and SPRAWL does not seek to maintain or prevent these irrelevant effects. The ignore case shown as a totally ordered plan in Figure 1 demonstrates this. In this problem, one of the active conditional effects in the totally ordered plan is the effect $b \rightarrow c$ from step op1. However, this effect does not affect the fulfillment of the goal state, and so is not a relevant effect. In fact, as is shown in Figure 2, SPRAWL would enforce no ordering constraints between the two steps in its partial ordering. Though the different orderings produce different final states, the goal terms are true in each of these final states, so it doesn’t matter which occurs.

**Finding Multiple Partial Orderings**

Although, as we discussed, SPRAWL is restricted to partial orderings with no relevant effects not active in the given totally ordered plan, this does not mean that all relevant effects in the totally ordered plan must be relevant effects in the partial ordering. Thus, there could be several possible minimal annotated consistent partial orderings.

Sometimes, there are several relevant effects in the totally ordered plan that achieve the same aim. Bäckström presented an example that neatly illustrates this (Bäckström 1993). The totally ordered plan is shown with its needs tree in Figure 10. In this plan, two different relevant effects provide the term $q$ to step $c$—both step $a$ and step $b$ generate $q$. Choosing a different relevant effect to generate $q$ creates a different partial order. The two partial orders representing each of the two relevant effect choices are shown in Figures 11 and 12.

The needs analysis algorithm shown in Table 1 produces a needs tree that encompasses all possible partial orderings consistent with the totally ordered plan, and SPRAWL searches through these orderings to identify the optimal one according to a given measure. However, finding the optimal partial ordering “under reasonable optimality criteria” has been shown to be NP-hard (Bäckström 1993). In Table 3, we provide a polynomial-time algorithm for finding one (not necessarily optimal) minimal annotated consistent partial ordering. This algorithm is a variation on the one presented by (Veloso, Pérez, & Carbonell 1990), however, in order to handle conditional effects, we must calculate the state between each step to determine whether the conditional effects were active in the totally-ordered plan.

**Conclusions**

In this paper, we have described our SPRAWL algorithm for finding optimal minimal annotated consistent partial orderings of observed totally ordered plans. We first described some of the previous work in plan analysis. We then described our novel needs analysis approach to finding the relevant effects and needs of each operator, presented the needs analysis algorithm in detail, illustrated its behavior with an example, and discussed its complexity. We then presented and explained the complete SPRAWL algorithm for finding
Figure 10: Bäckström’s example plan and the needs tree created by needs analysis. Note that the term \( q \) is accomplished by two different steps: \( a \) and \( b \). This means that two partial orderings are possible: one in which step \( a \) provides \( q \) to step \( c \), and one in which \( b \) does.

Figure 11: One possible partial ordering of Bäckström’s example plan.

Figure 12: Another partial ordering of Bäckström’s example plan, found by SPRAWL when the evaluation function favors shorter partial orderings.

Table 3: A polynomial-time algorithm for finding one (not necessarily optimal) MACPO


