Northwestern University

The Institute for the Learning Sciences

REPRESENTATION AND PERFORMANCE IN A PARTIAL ORDER PLANNER

Technical Report # 35 • October 1992

Gregg Collins
Louise Pryor

Established in 1989 with the support of Andersen Consulting

Representation and Performance in a Partial Order Planner

Gregg Collins and Louise Pryor

The Institute for the Learning Sciences
Northwestern University
1890 Maple Avenue
Evanston IL 60201

This work was supported in part by the AFOSR under grant number AFOSR-91-0341-DEF, and by DARPA, monitored by the AFOSR under contract F49620-88-C-0058. The Institute for the Learning Sciences was established in 1989 with the support of Andersen Consulting, part of The Arthur Andersen Worldwide Organization. The Institute receives additional support from Ameritech and North West Water, Institute Partners, and from IBM.

Contents

1 Introduction

Because planning systems solve problems by reasoning about actions and their expected effects, the representation of actions has long been a central issue in planning research. Many current planning systems use representation schemes based on the so-called STRIPS operator (Fikes & Nilsson, 1971). Such systems often incorporate modifications to the basic format of the STRIPS operator aimed at improving its efficiency, its expressiveness, or both. One common modification is the use of *filter conditions* (Charniak & McDermott, 1985; Tate, 1977; Wilkins, 1988). In this paper we discuss the arguments that have been advanced for adding filter conditions to the STRIPS representation, and consider how filter conditions could be implemented in a partial order planner. We reach three conclusions: first, while filter conditions can be added to a partial order planner, they are not a natural extension of such an algorithm; second, many of the purposes that have been suggested for filter conditions are in fact better achieved through other means; third, Pednault's *secondary preconditions* (Pednault, 1988a; 1988b; 1991), which can, as we demonstrate, be implemented straightforwardly in a partial-order planner, achieve the basic functionality of filter conditions more efficiently.

The basic STRIPS operator comprises the *preconditions* for executing an action, a list of the facts that become true as a result of executing the action (the *add-list*), and a list of the facts that become false as a result of executing the action (the *delete-list*). Figure 1.1 shows a STRIPS operator for pushing an object from one place to another. Roughly speaking, the operator asserts the following: in order to push an object

Action:	(push ?object ?from ?to)
Preconditions:	(at robot ?from) (at ?object ?from)
Delete list:	(at robot ?from) (at ?object ?from)
Add list:	(at robot ?to) (at ?object ?to)

Figure 1.1: STRIPS operator to push ?object from ?from to ?to
(adapted from Fikes & Nilsson (1971))

(*?object*) from a starting point (*?from*) to a destination (*?to*), both the robot and the object must be at the starting point to begin with; as a result of executing the action, the robot and the object cease to be at the starting point and come to be at the destination.

In the STRIPS paradigm the preconditions of an operator become new goals (or *sub*goals) for the planner when that operator is included in the plan. Preconditions that become subgoals are called *enabling* preconditions (Pednault, 1988b). Alternatively, preconditions might be interpreted as stating requirements that must be met if the operator is to be selected for inclusion in the plan to begin with; such an operator will be chosen only if its preconditions are not already true in the world. Because such preconditions in effect act as filters on operator selection, they are commonly referred to as *filter conditions*. Filter conditions are useful because in certain situations planning to achieve an unmet subgoal may lead to inefficient or nonsensical plans. For instance, an operator to turn on a light might have a precondition that the switch be *off* initially, but it would make no sense to turn the switch off in order to apply the operator. The precondition *switch off* should thus be made a filter condition rather than an enabling precondition.

In many systems, the basic STRIPS operator has been extended to allow the specification of filter conditions as well as enabling conditions. This is particularly common in systems that are intended to be of practical use (*e.g.* Currie & Tate, 1985; Currie & Tate, 1991; Tate, 1977; Wilkins, 1988). Such systems typically incorporate other extensions to the basic STRIPS representation as well, mainly involving the introduction of different types of preconditions and annotations to the preconditions. Many of the modifications that have been introduced are in fact special cases of filter conditions, as we shall see later on in this paper.

In general, planning systems that are concerned with formal properties[1] (*e.g.* Chapman, 1987; McAllester & Rosenblitt, 1991) have avoided such modifications. This is not surprising, inasmuch as the difficulty of proving the formal properties of

[1]For example, *completeness* (able to find all valid plans), *correctness* (all the plans produced are valid), and *systematicity* (every point in the search space is visited at most once) of the algorithm (*e.g.* Chapman, 1987).

a planing system will generally increase as the complexity of its representation increases. We became interested in action representations when we tried to add filter conditions into one such system, SNLP[2] (Barrett, et al., 1991; McAllester & Rosenblitt, 1991) in the hope that filter conditions might help to improve SNLP's poor performance on many simple blocks-world problems. Although our implementation resulted in an improvement over the unmodified algorithm, its performance was much worse than we expected, to the extent that there were many problems involving stacking three blocks that the system could not solve within a time-limit of several hours. This result led us to consider the intended functionality of filter conditions in more detail, and ultimately led to the conclusion that filter conditions are fundamentally incompatible with partial order planning. We therefore investigated alternative mechanisms for achieving the required functionality, and found that Pednault's secondary preconditions can be used to achieve much of the functionality of filter conditions, with much greater efficiency in SNLP.

In what follows we describe how we came to these conclusions. Although the use of filter conditions is widespread, very few explicit arguments have been made to justify their utility. We thus describe the various uses to which they have been put and provide possible explanations of these uses. We then describe the alternative mechanisms that can achieve the intended functionality of filter conditions within a partial order planner, and present an extension to SNLP that incorporates the secondary preconditions proposed by Pednault (Pednault, 1988a; Pednault, 1988b; Pednault, 1991).

[2]A Systematic Non-Linear Planner.

2 Partial order planning

Classical planning systems in general carry out a special case of *means-ends analysis* (Newell & Simon, 1963): Given a goal, the planner first chooses an action that, if executed in the right circumstances, will result in goal being achieved. Having thus chosen the action with which it will ultimately achieve the goal, the planner then goes about planning to ensure that circumstances will indeed be right. So, for example, given the goal "be in London", the planner might choose the action "take a taxi to Piccadilly Circus". Since this action will have the desired effect only if the planner is fairly close to Piccadilly Circus to begin with, the planner must then find a plan that will get it close enough to London to make taxi-taking viable. For instance, a reasonable plan might be to get to Heathrow airport, *then* take a taxi from the airport to Piccadilly Circus. In choosing this approach the planner acquires a subgoal to get to Heathrow, and it must then choose an action to achieve that subgoal—"take a flight from JFK", for instance. Since this action will have its own preconditions, the planner will acquire new subgoals for which it must choose another action, and so on. The construction of a plan is thus a recursive process that will, if successful, ground out in actions the preconditions of which are true to begin with.

The plan produced by such a classical planner is an ordering on a set of actions, where the ordering follows mainly from the constraint that if one action achieves a precondition of another action, the former must occur before the latter. This simple picture is complicated by the fact that several actions in conjunction are often needed to achieve the preconditions for a single action—for example, to fly from JFK to Heathrow, one would need to buy a ticket, secure a passport, and get to the gate from which the plane is to leave, among other things. While there is no *a priori* reason to impose a particular ordering on conjunctive actions, such an ordering may turn out to be necessary in cases where one of the actions would undo the effect of another. For instance, getting to the gate, in the above example, should probably be done after securing a passport, since securing the passport is likely to result in the planner no longer being at the gate. The planner must therefore not only find a set of actions that ensures that all subgoals are achieved, but also constrain the ordering of those actions so that they do not undo each other's results.

Two main approaches have been taken to this problem: *Total order* planners always maintain a total ordering on the actions in the (partially completed) plan, if necessary by making arbitrary ordering choices. If a total order planner fails to complete the plan, it may back up and try other possible orderings. *Partial order* planners, on the other hand, impose ordering constraints only when not doing so would demonstrably result in a plan that could fail, and otherwise allow actions to remain unordered with respect to each other. STRIPS (Fikes & Nilsson, 1971) is a total-order planner, while planners such as NONLIN and SIPE (Tate, 1976; Wilkins, 1988) are partial-order planners.

An issue that is sometimes confused with the issue of the ordering of the steps in a plan is the issue of the linearity of subgoals. Total order and partial order planners are sometimes known as *linear* and *non-linear* planners respectively. We prefer to reserve the latter terms to distinguish planners that cannot interleave actions aimed at achieving the preconditions of different actions from those that can so interleave actions. The question of whether a planner is a total order or partial order planner is orthogonal to the question of whether it is, in this sense, a linear planner or a non-linear planner. A partial order planner is always a non-linear planner in this sense, but the reverse is not true.

2.1 The SNLP algorithm

Our investigations were carried out using the Systematic Non-linear Planner (SNLP), which was implemented by (Barrett, et al., 1991) based on the algorithm of (McAllester & Rosenblitt, 1991). SNLP is a complete, correct and systematic partial order planner. The basic action representation in SNLP is the standard STRIPS representation with preconditions, add list and delete list, augmented by *codesignation constraints* on the variables (Chapman, 1987), which are used to specify that two variables must be equal, or that they must not be equal.

A set of **steps**:	The actions to be executed.
A set of **open conditions**:	States to be achieved.
A set of **links**:	Each link is of the form (*step1 state step2*), where *state* is required for the execution of *step2* and is achieved by *step1*.
A **partial ordering**:	Ordering constraints on the steps.
A list of **unsafe links**:	Those links for which there is a step (the **clobbering** step) that could potentially (according to the partial ordering) be executed between *step1* and *step2* and that would unachieve *state*.
A set of **codesignation constraints**:	Required or forbidden bindings for the variables in the partial plan.

Figure 2.1: The structure of partial plans in SNLP

SNLP operates by searching the space of partial plans (figure 2.1). The basic algorithm is as follows:

- The most promising partial plan is chosen from the search queue.
- If the partial plan is complete (i.e., there are no open conditions and no unsafe links) the search process is terminated and the resulting plan is returned.
- If there are any unsafe links, one such link is removed from the unsafe list and each modification shown in figure 2.2 is attempted. Each successful modification produces a new partial plan, which is added to the search queue. Links made unsafe due to the modification are added to the unsafe list.
- Otherwise, there is at least one open precondition, for a step that is therefore not yet enabled. One such precondition is removed from the open list and each modification shown in figure 2.3 is attempted. Each successful modifica-

Promotion:	Introduce an ordering to ensure that the clobbering step occurs after the step at the end of the unsafe link.
Demotion:	Introduce an ordering to ensure that the clobbering step occurs before the step at the beginning of the unsafe link.
Separation:	Introduce codesignation constraints to ensure that the state resulting from the clobbering step cannot unify with the state in the unsafe link.

Figure 2.2: Modifications for unsafe links

Add new **step**:	Find an operator with a proposition in its add list that can be unified with the open condition. Make the operator the new step, add its preconditions to the list of open conditions, and add its codesignation constraints. Add a link from the new step to the unenabled step.
Add new **link**:	Find a step with a proposition in its add list that can be unified with the open condition. Add a link from the found step to the unenabled step.
Add **link**:	Add the bindings necessary for the unification to the set of codesignation constraints. Make the appropriate ordering constraint.

Figure 2.3: Modifications for open preconditions

tion produces a new partial plan, which is added to the search queue. Links made unsafe due to the modification are added to the unsafe list.

3 Filter conditions

A *filter condition* is a precondition that does not become a new subgoal. For example, "the light is off" might be a filter condition for the action of turning the light on. This would mean two things: first, the light must be off in order to turn it on, and second, one would never want to turn the light off in order to turn it on. In general, if the filter conditions of an operator would not be true at the point at which the operator would be executed, the use of that operator is ruled out. This contrasts with *enabling* preconditions, which become new subgoals to be achieved through subsequent planning.

If we are going to consider whether filter conditions should be added to a planner's action representation, we would like to know how and when they should be used. The answer to this question is surprisingly unclear. Charniak & McDermott (1985) state that "It is usually obvious what conditions belong in which category [filter conditions or enabling preconditions]" but do not elaborate further, and in general other researchers do not appear to be much more forthcoming on the subject.[3] In this section we shall examine the four uses that have been proposed for filter conditions, and discuss the justifications that have been put forward for these uses. It should be noted that the four are not mutually exclusive: a single filter condition may serve more than one purpose.

3.1 Plan efficiency

3.1.1 Using filter conditions to avoid inefficient plans

Filter conditions can be used to block the construction of inefficient plans by preventing the use of particular operators in certain situations. For example, Charniak and McDermott describe a blocks world in which there is a special operator for getting a block onto the table when it is glued to the wall (Charniak & McDermott, 1985). They argue that this operator should be given the filter condition "the block is

[3]Possible exceptions are (Feldman & Morris, 1990), and (Currie & Tate, 1991); we will consider their work in some detail below.

Action:	(move-from-wall ?block)
Filter conditions:	(glued-to-wall ?block)
Preconditions:	
Delete list:	(glued-to-wall ?block)
Add list:	(on ?block table)

Figure 3.1: Modified STRIPS operator to move a block from a wall
(adapted from Charniak & McDermott (1985))

glued to the wall" (see figure 3.1), preventing its use in cases in which the block is not initially glued to the wall. The alternative would be to make "the block is glued to the wall" an enabling precondition for the operator, which would allow the possibility that the planner would glue a block to the wall and then remove it in order to get that block onto the table. Charniak and McDermott's point is that such a plan is obviously inefficient, and the planner ought to be prevented from generating it. SIPE makes similar use of filter conditions to avoid inefficient plans (Wilkins, 1988). For example, the SIPE operator that fetches objects from a room has the filter condition "the object is in the room" (see figure 3.2). This prevents the planner from moving an object into a particular room solely in order to fetch it from that room.

The general idea behind the use of filter conditions to avoid inefficient plans is this: if you know in advance that in certain circumstances operator O will not be the best method of achieving its result, then don't use operator O in those circumstances. For example, common sense would dictate that if one is traveling between two locations in the western hemisphere, flying via London will not be the best plan. The planner can be prevented from generating such a plan by using a filter condition on the operator—for example, by adding the filter condition "not traveling between cities in the western hemisphere" to the operator representing the flight to London.

The use of filter conditions to rule out inefficient plans depends on our ability to construct a set of propositions like our assertion about flying to London, having the general form "there is a better alternative to operator O in circumstance C." Such propositions have a very interesting property, namely that they anticipate the

Action:	(fetch ?object1 ?room)
Filter conditions:	(inroom ?object ?room)
Preconditions:	(inroom robot ?room) (nextto robot ?object)
Delete list:	(nextto robot ?object)
Add list:	(holding robot ?object)

Figure 3.2: Modified STRIPS operator to fetch an object from a room
(adapted from Wilkins (1988))

outcome of the planning process. That is, they do not merely assert something about the cost of applying a particular operator, but in addition make a claim about the availability of alternative plans to that operator. Since in general it is not possible to find easy-to-compute features that infallibly predict the outcome of the planning process[4], in general such assertions will be *assumptions*. For example, in the airline scheduling problem, the proposition "there is a better alternative to *flying via London* in *traveling between two locations in the western hemisphere*" is an assumption based on the expected availability of flights, among other things. It is a plausible assumption, but it may fail in some unusual circumstance. Likewise, it is an assumption that gluing a block to the wall and then ungluing it will not be the best way to get the block onto the table, which depends on the availability and efficacy of other methods for transporting blocks.

Having clarified the theory behind the use of filter conditions to avoid inefficient plans, we are now in a position to see why there are flaws in this approach. There are two major ones: First, because the use of filter conditions to rule out inefficient plans in general involves making assumptions, as detailed above, it would mean that the planner could not be guaranteed to be complete, since in cases where the assumption failed the planner might not find a plan even though a viable plan did exist.[5] Second, since there is no known method for automatically generating proposi-

[4]If it *is* easy to find such features, then it is probable that "planning" in the domain in question can be carried out by an algorithm that is much simpler than a general-purpose planner.
[5]"Condition typing allows information to be kept [about the role of a condition]. However use of this information itself will almost certainly commit the planner to prune some of the potential search space

tions of the form "there is a better alternative to operator O in circumstance C", we cannot describe a general method for avoiding inefficient plans using filter conditions; we can only describe a method for the special cases in which such propositions can be discovered. There is therefore still a need to find a general method of avoiding inefficient plans, and in fact there appears to be a straightforward approach to the development of such a method, which we describe below. Given this general method there is no clear reason to retain the special-purpose method based on filter conditions.

3.1.2 Inefficient plans in SNLP

In SNLP, the problem of avoiding inefficient plans can be formulated as the problem of controlling the search process efficiently. At each step SNLP extends the most promising partial plan that has been formulated up to that point. The ranking function used to decide which plan is most promising can and should be constructed to consider the costs of the partial plans and the estimated costs of completing them.[6] This approach to avoiding inefficient plans is preferable to the use of filter conditions for two reasons: First, SNLP never completely rules out any valid partial plan; it simply avoids spending time on plans that are relatively unpromising. For instance, in Charniak and McDermott's example, the search process would in general avoid gluing a block to a wall and then ungluing it as a method of moving the block to the table simply because it would first extend plans using less expensive methods, such as picking up the block and setting it down in the desired location. If such methods were for some reason impossible to apply in a particular case, however, the planner would attempt to extend a plan involving gluing a block to the wall and then ungluing it. SNLP thus remains complete while avoiding consideration of inefficient plans. Second, rather than requiring the programmer to discover propositions that imply the unfitness of particular operators in particular circumstances, SNLP simply requires methods for estimating the costs of executing operators and achieving subgoals.

---

thereby losing claims of completeness [if inappropriate condition types are used]." (Currie & Tate, 1991).

[6] In other words, to do an A style search, in which the partial plan that is extended is the one with the lowest estimated total cost.

Action:	(move ?block ?new)
Filter conditions:	(on ?block ?base) (inst ?base block) (inst ?new block)
Preconditions:	(clear ?block) (clear ?new)
Delete list:	(on ?block ?base) (clear ?new)
Add list:	(on ?block ?new) (clear ?base)

Figure 3.3: Modified STRIPS operator to move a block from one block to another
(adapted from (Charniak & McDermott, 1985))

3.2 Indicating unachievable goals

3.2.1 Using filter conditions to avoid unachievable goals

Filter conditions can be used to prevent the planner from pursuing subgoals that cannot be achieved by any known action. For example, figure 3.3 shows an operator for moving a block from one supporting block to another. Since this operator is only applicable if both the original location and its destination are other blocks, "the current support is a block" and "the target support is a block" must be either filter conditions or enabling preconditions. Since there is no action in the standard blocks world that results in the creation of a new block, it has been suggested that such preconditions should in fact be filter conditions (Charniak & McDermott, 1985). O-Plan supports a similar use of filter conditions, termed *only-use-if* preconditions (Currie & Tate, 1991).

We believe that the argument for this use of filter conditions is specious. As in the case of using filter conditions to avoid inefficient plans, the method relies on an implicit assumption, namely the assumption that a particular condition such as "the object's support is a block" is unachievable by the planner. If we view the set of planning operators as being open rather than closed, then such an assumption may be invalidated at any time by the addition of a new operator. For example, the fact

that Charniak and McDermott's planner does not have an operator to make a block does not mean that such an operator will not be added. Building assumptions of this type into planning operators means that when a new operator is added any number of old operators may be invalidated. In effect, using filters in this way constitutes a breakdown of modularity in the definition of operators, which has the undesirable effect that lack of modularity has in any design process.

Such a violation of modularity might be acceptable if it were offset by a corresponding benefit, but there is no such benefit in this case. If an operator is given an enabling precondition that is unachievable by any known operator, the planner will simply fail to find an operator to achieve that precondition, and the plan will fail if the condition is not true *a priori*. The planner will of course pay the cost of checking to see whether any operator can achieve the condition, but there is no reason to believe that in general the processing overhead incurred in treating the condition as a filter condition would be any less than the overhead incurred in treating it as an enabling precondition.

3.2.2 Unachievable goals in SNLP

A dead end in SNLP's search space is an incomplete partial plan that cannot be modified to resolve unsafe links or establish unachieved conditions. When such a partial plan is encountered, that branch of the search simply fails. A partial plan containing an unachievable goal such as "the table is a block" cannot be modified so as to achieve the goal, as SNLP will not be able to find an operator with an add-list containing "the object is a block". SNLP's general mechanism for recognizing dead ends is thus entirely adequate for recognizing unachievable goals.

3.3 Loops

3.3.1 Using filter conditions to avoid loops

Filter conditions can be used to avoid certain kinds of loops. Consider, for example, a blocks world domain with two actions: picking a block up, and putting a block down on another block (see figure 3.4). Suppose that a plan is required that will move block A from block B to block C (see figure 3.5). Block C must be clear before

Action:	(pickup ?block)	(putdown ?base)
Preconditions:	(clear ?block) (empty-hand) (on ?block ?base)	(clear ?base) (holding ?block)
Delete list:	(empty-hand) (on ?block ?base)	(clear ?base) (holding ?block)
Add list:	(holding ?block) (clear ?base)	(on ?block ?base) (empty-hand)

Figure 3.4: STRIPS operators in a blocks world domain

block A can be put on it, and since block C is clear in the initial configuration common sense suggests that the planner should simply take advantage of this and proceed to move block A. However, there is another possible approach, which is to ensure that block C is clear by removing a block from it. For example, consider the following plan:

1. Pickup A
2. Put A onto C (so that something is on C)
3. Pickup A (so that C is clear)
4. Put A onto B (so that the hand is empty)
5. Pickup A
6. Put A onto C (to achieve the main goal)

This is in fact a valid plan, but it is apparent to the observer that it cannot possibly be the best plan, because it contains a loop: the condition A on C is achieved in support of the preconditions of the action of putting A on C. A planner like SNLP

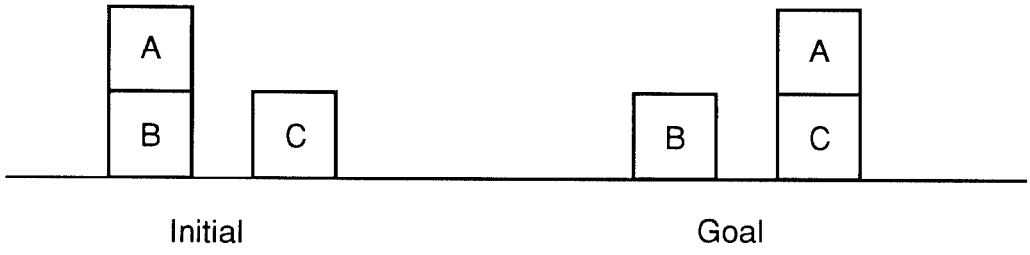


Figure 3.5: A blocks world problem

Action:	(fetch ?object1 ?room)	(move ?object1 ?room)
Filter conditions:	(inroom ?object ?room)	(inroom robot ?old-room)
Preconditions:	(inroom robot ?room) (nextto robot ?object)	(holding robot ?object)
Delete list:	(nextto robot ?object)	(holding robot ?object) (inroom robot ?old-room) (inroom ?object ?old-room)
Add list:	(holding robot ?object)	(inroom robot ?room) (nextto robot ?object)

Figure 3.6: Modified STRIPS operators in the SIPE domain
(adapted from (Wilkins, 1988))

will avoid generating the inefficient plan simply because it will find a better plan first, namely:

1. Pickup A
2. Put A onto C (to achieve the main goal)

This depends, however, on the accuracy of the planner's cost-estimating functions. If the top level goal is simply "get A on C", then the planner is unlikely to extend the undesirable looping plan very far. If, however, the top level goal was something like "get A on C and achieve *G*", where *G* is a subgoal the achievement of which will not interact significantly with the achievement of "get A on C", then the situation would be different because the planner would have much more of an opportunity to make erroneous cost estimates. For example, suppose that the planner initially underestimates the cost of achieving *G*. This will tend to cause it to expand all possible alternatives for achieving "get A on C", since as each partial plan is extended to deal with G it will begin to look more and more expensive. Thus SNLP may generate a partial plan containing the loopy plan for getting A on C described above before its attention shifts back to the preferable alternative. We have observed SNLP in similar situations building many iterations of such loops before it finds a plan. The cost of considering such plans may be very high.

This loop, and others like it, can be avoided through the use of filter conditions. If "the block to be picked up is on another block" is treated as a filter condition for the pickup action, instead of an enabling precondition, the planner can never put a

block on another block simply in order to take it off again. This change would prevent the planner from generating step 2 of the loopy plan described above. A similar use of filter conditions is made in the SIPE domain of (Wilkins, 1988). Consider the two operators shown in figure 3.6. Without filter conditions, a planner might devise a plan to move an object to room1 that involved moving it to room1 so that it could fetch it in room1 enabling it to move it to room1—and so on *ad infinitum*. If, however, the object's being in a room is a filter condition rather than a precondition for fetching it from that room, these loops cannot arise.

The use of filter conditions in avoiding loops was recognized by (Feldman & Morris, 1990). The result of their analysis is to eliminate two simple types of loops: those in which a goal is an immediate subgoal of itself, and those in which a goal repeats after two steps. However, these loops are eliminated only in certain special circumstances that apply only in very restricted domains.[7] It would not, for instance, work in any of the common representations of blocks world.

Thus we have once again an application of filter conditions to solve a problem that is a special case of one that the planner must solve in any case. This use of filter conditions does not address the general question of loop detection and elimination, but only rules out certain simple cases of loops. Furthermore, there does not seem to be anything especially natural about using filter conditions in this way: they are simply a device for causing the planner to skip some portions of the search space, and any such device could in principle be manipulated in such a way as to prevent some loops.

In fact, to our knowledge, no one has yet proposed a method that would rule out loopy plans in general, but there is certainly reason to pursue the creation of such a method. It appears to be somewhat tricky to state the conditions under which a loop may be recognized, since in a partial order planner a partial plan that appears to contain a loop may be modified by the addition of steps into the apparent loop in such a way that the completed plan is free of loops. This is a problem for future research.

[7]The example they present in their paper does not seem to meet the conditions for the analysis they propose.

3.3.2 Loops in SNLP

SNLP does not avoid loops directly, so it will in fact extend plans containing loops if its current estimate ranks this as the most promising partial plan. However, for every plan containing a loop there is a shorter (and thus less expensive) plan that does not contain the loop, and SNLP will generate this possibility as well. Eventually SNLP can generally be expected to extend this plan to completion in preference to the loopy plan[8], although as discussed above it may expend considerable effort extending the loopy plan in the interim. Of course, if an inexpensive loop-detector did exist, it could easily be incorporated in SNLP .

3.4 Accounting for effects

An action will in general have different effects in different contexts, and any scheme of action representation must be able to represent these context-dependent effects. In STRIPS operators, context-dependence is accounted for through the use of variables, which may appear in the preconditions, add list, and delete list of the operator. Each possible instantiation of these variables represents a different context in which the action associated with the operator might be executed. For example, consider the TIP operator in figure 3.7. Here the container that is to be tipped is represented by a variable (*?container*) so that instances of tipping one container can be differentiated from instances of tipping another. Since the operator's effects are stated in terms of the variables, different effects will be asserted in different contexts, allowing the planner to represent, for example, the fact that the container that is tipped is the same container that is subsequently empty.

[8]In fact, if SNLP's cost-estimating functions were admissible—that is, always underestimates of the actual cost—it could be guaranteed that SNLP would find the cheaper plan. Of course, the likelihood of finding admissible cost-estimating functions in real-world planning domains is virtually nil.

Action:	(tip ?container)
Preconditions:	(container ?container) (full ?container) (contains-liquid ?container) (holding robot ?container) (directly-above ?container ?target)
Delete list:	(full ?container) (contains-liquid ?container)
Add list:	(empty ?container) (wet ?target)

Figure 3.7: A STRIPS operators for TIPping a container

There are two ways in which the variables in an operator may become instantiated: they may be bound when the operator is chosen, or they may be bound during subsequent planning. A variable is bound at the time the operator is chosen if a particular binding of that variable is necessary in order to ensure that the operator will in fact achieve the goal. For example, if the TIP operator is chosen to achieve the goal "Container A is empty", then clearly the variable representing the container must be bound to A. A variable is bound during subsequent planning when its binding does not matter for the achievement of the goal. For example, the TIP operator has a variable (*?target*) that represents the object directly under the container. Since whatever is under the container will get wet when the container has liquid in it and is tipped, this variable is needed to represent the context dependency of this effect of the TIP operator. However, since the identity of the object beneath the container does not affect the achievement of the goal to empty the container, the variable will be left unbound when the operator is chosen.

If the planner is to correctly project all the effects of executing the TIP operator, it must at some point bind all its variables. The variables that are not bound when the operator is selected will be bound in the course of choosing methods to achieve the operator's preconditions. For example, the TIP operator has the precondition "the container to be picked up is directly above a particular object", represented as:

(directly-above ?container ?target).

The planner's options in establishing this precondition are: (1) to assume that the container will remain directly above whatever object it is directly above at the start of the plan, (2) to assume the container will remain directly above whatever object it came to be above as the result of a previous action in the plan, or (3) to schedule an operator designed to position the container over a particular object. The third possibility, scheduling an operator to position the container over a particular object, seems pointless, since the operator will achieve the goal of emptying the container regardless of what object the container is positioned over.

This represents an efficiency problem similar to that described in the section on loops. While it is clear that for any plan in which an action is taken solely to position the container there is a more efficient alternative plan in which the action is not taken, the planner may nonetheless waste considerable amounts of time in considering plans that involve the less efficient choice. One way to fix this problem is to represent as filter conditions those preconditions that establish the bindings of variables not bound at selection time. For example, the condition "the container is directly above a particular object" could be made a filter condition for the TIP operator. This would mean that the planner would never move the container in order to achieve this condition. The filter condition would still fulfill the objective of discriminating the context in which the operator is being applied, by binding the variable representing the object under the container.

By using filter conditions to discriminate between contexts in which an operator might be executed, we prevent the planner from taking actions designed to switch between these contexts. The argument for this approach is that, since it does not matter to the achievement of the goal which context the operator is executed in, there is no point in expending effort solely to make such a switch. Notice, however, that this reasoning applies only to the use of an operator to achieve a particular goal. Since a STRIPS operator may be used to achieve any condition on its add list, this is problematic. For instance, in our example the TIP operator was chosen to achieve the condition "a particular container is empty", but it might equally well have been chosen to achieve the condition "a particular object is wet". In this case it would not make sense to have "the container is over a particular object" be a filter condition. In fact, it would be essential that it be an enabling precondition, since the planner

Action:	(tip ?container)	(tip ?container)
Filter conditions:	(directly-above ?container ?target)	
Preconditions:	(container ?container) (full ?container) (contains-liquid ?container) (holding robot ?container)	(container ?container) (full ?container) (contains-liquid ?container) (holding robot ?container) (directly-above ?container ?target)
Delete list:	(full ?container) (contains-liquid ?container)	(full ?container) (contains-liquid ?container)
Add list:	(wet ?target)	(empty ?container)
Only use for:	(empty ?container)	(wet ?target)

Figure 3.8: Modified STRIPS operators for the TIP action

should consider plans in which the container is moved in order to position it over the object that is to be made wet.

One solution to this problem would be to incorporate the distinction made in O-Plan between *only-use-for* effects and regular effects (i.e., side-effects). An operator such as TIP could be split into two (or more) operators , each one having a single effect as its *only-use-for* effect (see figure 3.8). This would allow the planner to have one operator for emptying containers, for which "the container is directly above a particular object" would be a filter condition, and another operator for getting things wet, for which this would be an enabling precondition. Preconditions that are only required in order to make variable bindings, like this filter condition, are termed *query* conditions in O-plan (Currie & Tate, 1991).

This scheme appears to be workable, and it represents the first solid argument that we have found for the use of filter conditions. However, as we discuss in the following sections, filter conditions cannot in theory work effectively in many types of planners, and in particular we empirically demonstrate their inefficacy in SNLP. An alternative representation mechanism, the secondary preconditions introduced by Pednault (1988a; 1988b; 1991), is as effective as the method based on filter conditions for dealing with context-dependent effects, and can be more effectively implemented within partial-order planners such as SNLP.

Action:	eat-dry-cereal	eat-wet-cereal	buy-cereal
Filter conditions:	lack-milk	have-milk	
Preconditions:	have-cereal	have-cereal	lack-cereal
Delete list:	hungry	hungry	lack-cereal lack-milk
Add list:	thirsty		have-cereal have-milk

Figure 4.1: Modified STRIPS representation for a cereal domain

4 Filter conditions in a partial order planner

Filter conditions are meant to act as constraints on the selection of operators. If the operator's filter conditions are true, it may be selected; if they are not, it may not. This deceptively simple description hides some complexity, however. In particular, the condition on selection is more accurately stated as "the operator's filter conditions would be true at the time when it would be executed, were the operator in fact selected". The computation of this complex condition may be tricky.

For example, consider the following problem involving a hungry graduate student. She can satisfy her hunger by eating cereal. If she has milk, she will eat the cereal with milk, but if she has no milk she must eat it dry, in which case she becomes thirsty. If she has no cereal, she can go to the store and buy some; whenever she is at the store to buy cereal, she also buys milk. The modified STRIPS operators for this (somewhat contrived) domain are shown in figure 4.1.

Now suppose that the student is hungry, and lacks both cereal and milk. Her goal is to assuage her hunger. There are two operators she could use: she could eat cereal either wet or eat cereal dry. In order to decide which she will do at the time that she is constructing her plan, she must make a judgment as to which operator will be applicable. Examining the filter conditions at planning time, she sees that the *eat-wet-cereal* operator requires that there be milk. Since she lacks milk, she rejects this operator and decides on the *eat-dry-cereal* operator instead. Since she does not have any cereal, she sets up a new subgoal to acquire some, which she can achieve by us-

ing the *buy-cereal* operator. At this point her plan consists of the two operators: *buy-cereal* and *eat-dry-cereal,* in that order. Although all the preconditions in the plan are now satisfied, it turns out that the filter condition for *eat-dry-cereal* will not in fact be satisfied at the time when the action comes to be executed, because the action of buying cereal results in the purchase of milk as well as the purchase of cereal, so that the condition *lack-milk* will not be true. The planner has thus made a mistake due to its inability to guess correctly whether a filter condition would hold at the time when an operator was executed.

The problem in the above example came about because the planner added an operator to the plan affecting the value of the filter condition of an operator that had previously been scheduled. If new operators can be inserted at any point in the plan, then it is impossible to be sure what the situation will be when a particular action is executed until the plan is complete. It is therefore impossible to use filter conditions in such a planner to rule out alternative operator choices, since they cannot be used to rule out partial plans. An important aspect of the rationale for filter conditions—ruling out unreasonable uses of individual operators in partial plans—is therefore missing. This argument implies that filter conditions will be ineffective in all non-linear[9] planners (because the scheduling of operators may be changed during the planning process). Our prediction that a partial order planner using filter conditions will perform inefficiently is supported by the empirical results we describe in section 6 below.

4.1 A basic implementation of filter conditions in SNLP

To test the feasibility of a straightforward implementation of filter conditions, we constructed a version of the SNLP planner that uses a STRIPS representation augmented to include filter conditions. The modified algorithm proceeds in exactly the same way as the standard SNLP algorithm, ignoring filter conditions, until all outstanding subgoals of a partial plan are satisfied. At this point, the planner attempts to show that the existing plan satisfies all the filter conditions of its operators; if any filter condition is not satisfied, the plan is rejected. In attempting to show that a fil-

[9]Note that all partial order planners are non-linear.

ter condition is satisfied, the planner may add constraints to the plan involving the ordering of operators or the identity of variables, but it may not add a new step in order to satisfy a filter condition.[10]

4.2 Using filter conditions in search

In the implementation of SNLP with filter conditions we have just described, the filter conditions play no role in the planning process until the end, and thus have no influence on the search process used in constructing partial plans. While filter conditions cannot be used to rule out operator choices during plan construction, they can be used to help guide the search. This can be done by prioritizing the search through the plan space based on an estimate of the likelihood that the filter conditions in a given partial plan will ultimately be established. While the information necessary to make an accurate estimate of this probability is not readily available, by counting the number of currently unestablished filter conditions a partial plan contains, the planner can make a crude guess as to the probability that the plan will eventually achieve these conditions. To test this alternative mechanism for implementing filter conditions, we implemented a version of SNLP in which partial plans that include unestablished filter conditions are penalized in the search process. As the empirical results in section 6 show, this implementation performs better than the basic implementation of filter conditions, despite the extra work involved in checking the filter conditions after each modification.

[10]Our implementation uses the normal mechanism for adding links to test the satisfiability of the filter conditions, and can thus be viewed as an adaptation of the technique suggested by (McAllester & Rosenblitt, 1991) for dealing with abstractions.

5 Operator selection and context-dependent effects

Although we have demonstrated that it is possible to implement filter conditions directly in SNLP, there are, as we have just seen, severe drawbacks to any such implementation, stemming from the very nature of partial order planning. These drawbacks led us to explore the alternative approach described below.

5.1 Secondary preconditions

Pednault presents a method that accounts for context-dependent effects of operators that avoids the difficulties described in section 3.4 (Pednault, 1988a; 1988b; 1991). In Pednault's method, a distinction is drawn between preconditions that are necessary in order for it to be possible to execute an action, which are termed *feasibility* or *primary* preconditions, and preconditions that specify the effect of an action on a particular state, termed *secondary* preconditions. A further distinction is drawn between two types of secondary preconditions:

- A *causation* precondition is a condition that must be true in order for a particular state to become true as a result of a particular action.
- A *preservation* precondition is a condition that must be true in order for a particular state to remain true through the execution of a particular action.

For example, consider the *pickup* operator from blocks world. Figure 5.1 shows the operator for the pickup action formulated in terms of Pednault's representation. Each item on the add list is associated with a set of causation preconditions, with the interpretation that the item is added only if its associated causation preconditions hold. Similarly, each item on the delete list is associated with a set of preservation preconditions, with the interpretation that the item is deleted only if its associated preservation conditions do *not* hold. For example, a block may become clear as the result of executing the pickup operator, but this effect results only if the block was previously supporting the block that was picked up.

Action:	(pickup ?block)	
Preconditions:	(empty-hand) (clear ?block)	
Add list:	(holding ?blocka)	(clear ?base)
Causation preconditions:	(= ?blocka ?block)	(on ?block ?base) (not (= ?base table))
Delete list:	(empty-hand)	(on ?block ?other)
Preservation preconditions:		(on ?block ?other2) (not (= ?other2 ?other))

Figure 5.1: The blocks world *pickup* operator using secondary preconditions

5.2 Secondary preconditions in SNLP

The basic SNLP algorithm can accommodate the addition of secondary preconditions without major modification. The required changes concern the structure of open conditions, and the generation of possible modifications to a partial plan that may be made in order to resolve an unsafe link (see figure 5.2) or establish an open condition (see figure 5.3). An open condition may now be required to be either true or false. The truth of a condition is established, as before, by a link from an add condition of a prior step: the falsity of a condition is established analogously by a link from a delete condition of a prior step.

A condition may be required to be true if it is:

- An enabling precondition of a step.
- A causation precondition of an effect possibly added by a step when the addi-

Prevention	*If the clobbering arises through the [addition, deletion] of a desired state, prevent if by adding the [causation, preservation] preconditions of the relevant proposition on the [add, delete] list (required to be [false, true]) to the list of open conditions, and adding the [negation of the causation, preservation] codesignation constraints to the set of codesignation constraints.*

Figure 5.2: New modification for unsafe links *with secondary preconditions*

Add new **step**	*If the open condition is required to be false,* find an operator with a proposition in its *delete list* that can be unified with the open condition. Make the operator the new step, add its preconditions (required to be true) and its codesignation constraints. Add a new link from the new step to the unenabled step.
Add new **link**	*If the open condition is required to be false,* find a step with a proposition in its *delete list* that can be unified with the open condition. Add a link from the found step to the unenabled step.
Add link	*Ensure that the proposition is deleted by adding one of the preservation preconditions of the proposition on the delete list (required to be false) to the list of open conditions, or adding the negation of one of the preservation codesignation constraints of the proposition on the delete list.* Add the bindings necessary for the unification to the set of codesignation constraints. Make the appropriate ordering constraint.

Figure 5.3: New modifications for open preconditions *required to be false*

tion is required in order to establish the truth of a further effect.

- A preservation precondition of an effect possibly deleted by a step when the deletion must be prevented so as not to clobber another condition.

A condition may be required to be false if it is:

- A causation precondition of an effect possibly added by a step when the addition must be prevented so as not to clobber another condition.
- A preservation precondition of an effect possibly deleted by a step when the deletion is required in order to establish the falsity of a further effect.

The secondary preconditions of an effect are never expanded into subgoals unless the effect either is used to establish a condition or must be prevented from clobbering an established condition. If neither of these cases applies, the truth or falsity of the effect has no influence on any other step in the plan, and it is ignored.

From this description of the modified algorithm it can be seen that secondary preconditions fit straightforwardly into the structure of SNLP, providing a neat way of representing context-dependent effects without multiplying the number of operators that are needed to represent the actions in a domain. Operators in this framework make the nature of context-dependent effects more explicit than do STRIPS op-

erators with filter conditions. The modified algorithm preserves completeness and correctness (see Appendix A).

5.3 Related work

Pednault introduced secondary preconditions in the context of ADL (Action Description Language), a planning formalism combining much of the expressive power of the situation calculus with the notational and computational power of the STRIPS representation (Pednault, 1988b; Pednault, 1989). The first implementation to use ADL was Pedestal (McDermott, 1989), a total order planner. More recently, Penberthy and Weld have implemented a partial order planner for ADL, which they call UCPOP (also based on SNLP) (Penberthy & Weld, 1992). Unlike these implementations, our work does not attempt to implement ADL: instead, we have tried to compare a single representation mechanism used in ADL (secondary preconditions) with an alternative mechanism (filter conditions).

Conditional preconditions of a slightly different form were used by Chien (Chien, 1990; Chien & DeJong, 1992). His notation allows negated clauses, thus allowing a single list of effects instead of the separate add and delete lists required by the STRIPS representation. Each effect on the single list may have conditional preconditions associated with it: the operator will not cause the effect unless the conditional preconditions are true. The conditional preconditions associated with a non-negated effect are thus the effect's causation preconditions in Pednault's terminology; those associated with a negated effect are the negation of the effect's preservation preconditions.

In SIPE (Wilkins, 1988) context-dependent effects are represented independently of the operators. Instead, they are deduced through the use of rules representing a causal theory of the domain.

6 Empirical results

We performed experiments using four versions of the SNLP algorithm:

SNLP-B The basic algorithm using basic STRIPS operators with no filter conditions or secondary preconditions (see section 2.1).

SNLP-F The algorithm using STRIPS operators with filter conditions, making use of the filter conditions only at the end of the planning process (see section 4.1).

SNLP-FF The algorithm using STRIPS operators with filter conditions, using the filter conditions to guide the search process (see section 4.2).

SNLP-SP The algorithm using STRIPS operators with secondary preconditions (see section 5.2).

We applied these four algorithms to 150 random test problems in a specially designed domain. This domain is described in Appendix B, and was designed to highlight context-dependent effects by eliminating the effects of expensive actions, unachievable goals, and loops. The test set of problems consists of 50 each with one, two and three goal conditions. The method of constructing the problems is described in Appendix B.

Three statistics were collected and analyzed:

- CPU time taken to solve the problem. This measures the overall efficiency of the algorithms.
- The length of the search path to a solution. This measures the search performance of the algorithms, ignoring such effects as the costs of matching and unification.
- The average branching factor in the search path. This gives some idea of how the algorithms might perform on more complex problems.

	Number of goal conditions			
Plan steps	1	2	3	Total
0	22	8	3	33
1	8	4	4	16
2	9	20	10	39
3	11	12	17	40
4		4	11	15
5		2	4	6
6				
7			1	1
Total	50	50	50	150

Figure 6.1: Problem complexity—numbers of problems by plan steps and goal conditions.

A limit of 25000 units was placed on the amount of CPU time allowed for each problem. The detailed statistics that were collected are shown in Appendix C. In our analyses we performed pairwise T-tests on planned comparisons between SNLP and SNLP-F, SNLP-F and SNLP-FF, and SNLP-FF and SNLP-SP, with a 95% significance level in all cases.

6.1 The test problems

The 150 randomly generated problems that we used included 50 each with one, two and three goal conditions. However, the number of goal conditions is not the best guide to the complexity of each problem, as the goal conditions may be true in the initial state, leading to a simple plan with no steps and one link establishing each goal condition. A better guide to problem complexity is given by the number of steps in the plan that solves the problem. An analysis of the complexity of the problems in our test set is shown in figure 6.1.

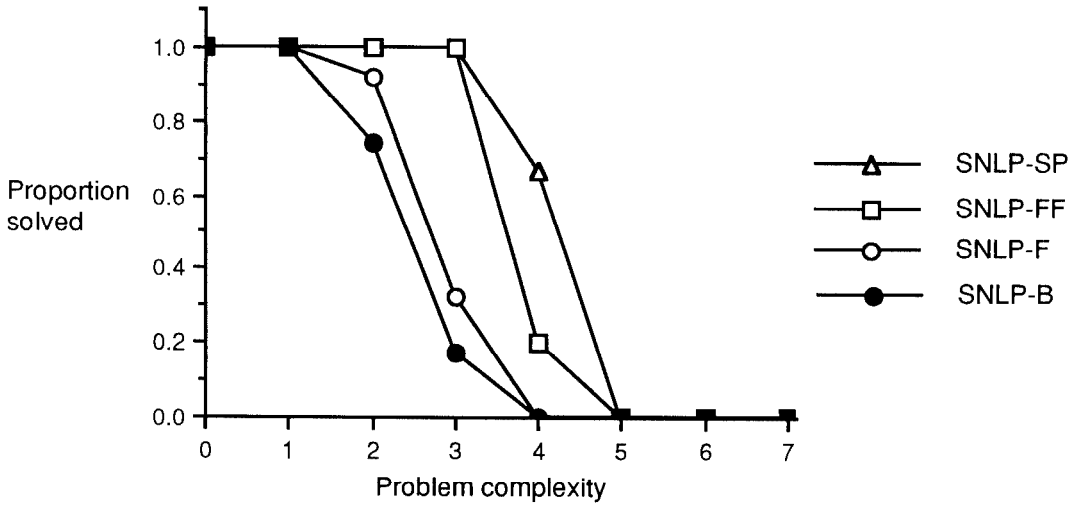


Figure 6.2: Success rate by problem complexity

6.2 Successful solutions

Not every algorithm solved each problem successfully within the time limit we imposed. Figure 6.2 shows how the algorithms performed by problem complexity. None of the algorithms solved the more complex problems within the time limit, with SNLP-SP performing best and SNLP-B worst. SNLP-SP solved all the problems with 3 steps in their solution, whereas SNLP-B solved only 7 out of these 40 problems, or 18%. In no case did another algorithm solve a problem on which SNLP-SP failed. It is somewhat discouraging to note that all the algorithms failed to solve even fairly simple problems within the time allowed, which was fairly generous (at least as measured by the patience of a researcher waiting for the experiments to run).

The results of the pairwise T-test are shown in Figure 6.3. All the comparisons are significant at the 95% level, so that SNLP-SP, SNLP-FF, SNLP-F, and SNLP-B are significantly more successful than each other in that order.

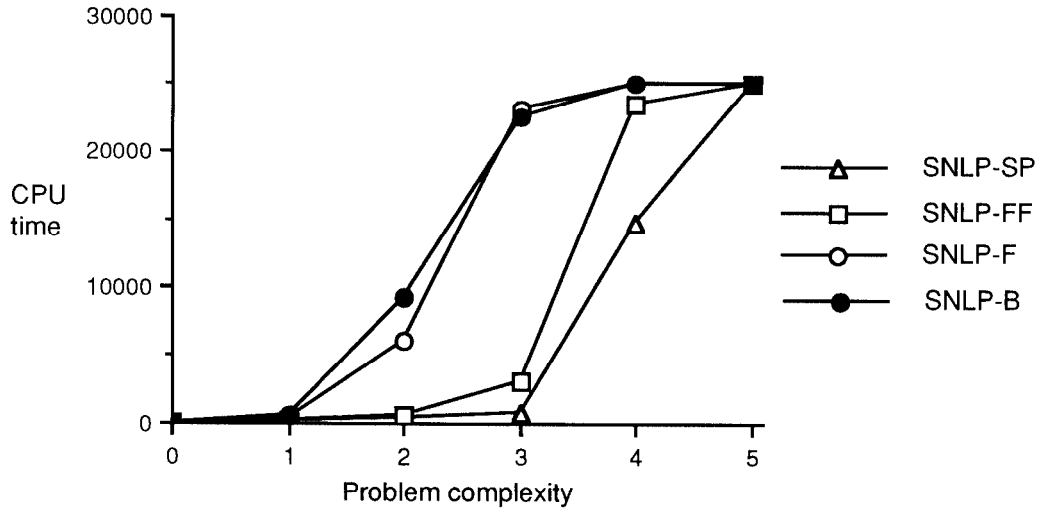


Figure 6.4: Mean CPU time taken

6.3 CPU time

The performance of the algorithms also varied widely as measured by the CPU time taken to solve each problem. In the analysis in this section problems that were not solved are taken into account at the time limit. This of course enhances the apparent performance of the unsuccessful algorithms. Figure 6.4 shows the mean CPU time taken by each algorithm on problems of varying complexity: the results for problems with solutions longer than five steps are not shown, because none of the algorithms solved them within the time limit.

SNLP-FF and SNLP-SP appear to be better than the other two algorithms, which seem to perform at roughly similar levels. However, the results may be distorted by

Comparison	Mean	T-value
SNLP-SP to SNLP-FF	0.047	2.701
SNLP-FF to SNLP-F	0.220	6.483
SNLP-F to SNLP-B	0.087	2.906

Figure 6.3: Success rate comparisons

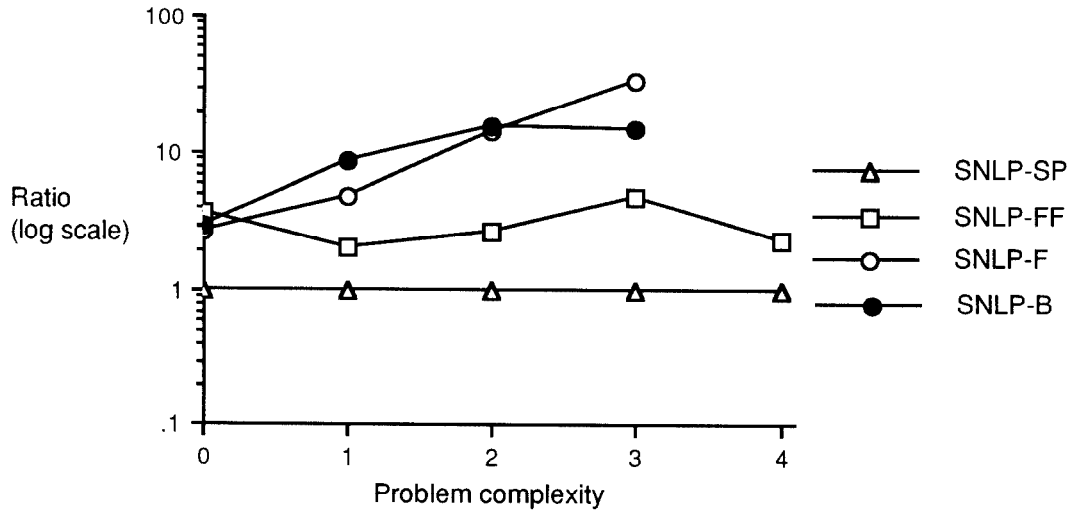


Figure 6.5: Mean CPU ratios

the under-estimates used for those problems that were not solved within the time limit. This effect is not present in Figure 6.5, which gives some idea of the relative efficiency of each algorithm on those problems that they solved. For each algorithm the time taken to solve the problem was divided by the time taken by SNLP-SP, and the graph shows the means of these ratios. Only those problems that were solved by the algorithm under consideration are included.

With the ceiling effect of the time limit removed, we can see that the difference in performance is really quite dramatic—SNLP-F taking as much as 35 times longer on average than SNLP-SP on problems with 3 steps in their solutions, and SNLP-B and SNLP-F taking over ten times as long on all except the simplest problems. SNLP-FF takes between two and five times as long on average as SNLP-SP on all the problems that it solves. The results for SNLP-F on problems with three steps in the solution are probably strongly affected by the fact that it solved eight of these problems in a CPU time at about the time limit, and another two took more than 90% of the time limit. All other problems that were solved by any of the algorithms within the time limit were solved in less than 70% of the time limit.

The results of the pairwise T-test are shown in Figure 6.6, by problem complexity and in total. The time limit is used for problems that were not solved, which is an underestimate. Cells in which no problems were solved are not shown, but are in-

Comparison	SNLP-B to SNLP-F		SNLP-F to SNLP-FF		SNLP-FF to SNLP-SP	
Complexity	Mean	T value	Mean	T value	Mean	T value
0	* 5.424	* 0.500	* -13	* -1.651	23	2.430
1	293	2.272	240	2.871	* 35	* 1.975
2	3219	3.376	5398	4.573	367	15.209
3	* -747	* -0.910	20127	22.139	2330	10.249
4			* 1478	* 1.796	8880	4.324
Total	* 670	* 1.905	6943	8.886	1611	5.407

Figure 6.6: Mean CPU time comparisons

cluded in the total. All results are significant at the 95% level except those marked with an asterisk. These results show that SNLP-F does not have a significantly better performance than SNLP-B, and that SNLP-SP does perform better than SNLP-F which in turn outperforms SNLP-F. These results are affected by including unsolved problems, which results in understating the difference between algorithms.

6.4 Path length

The CPU time taken to solve a problem is affected by the amount of matching and

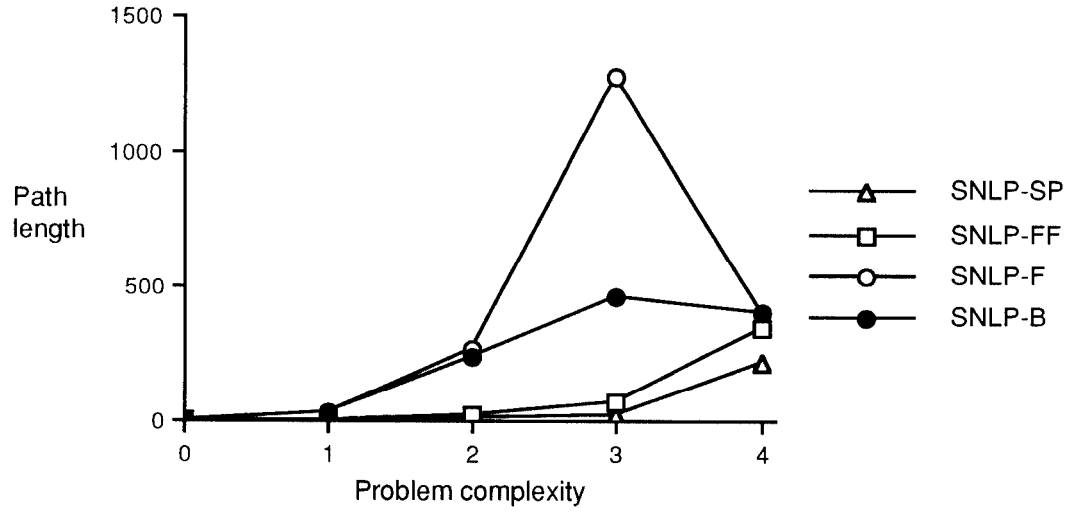


Figure 6.7: Mean length of search path

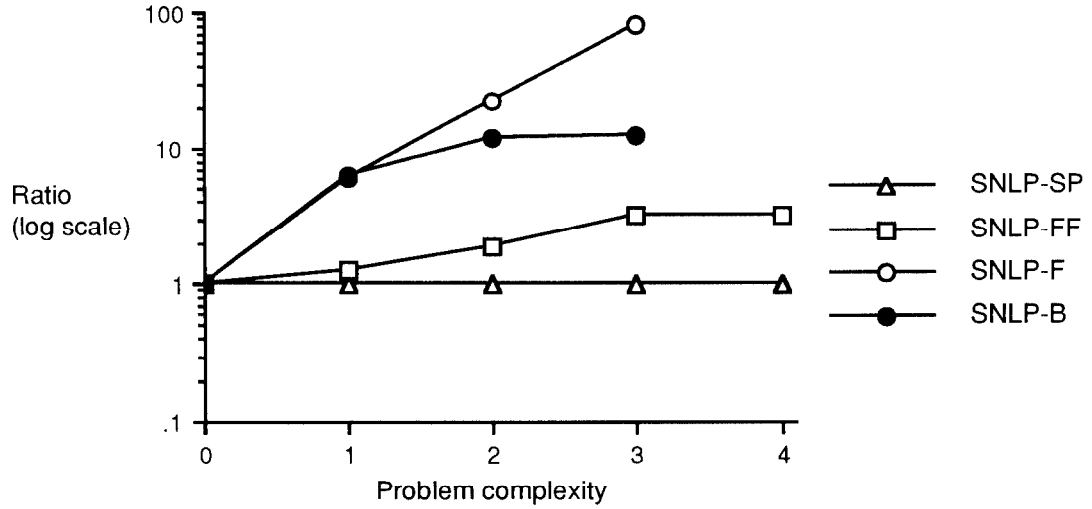


Figure 6.8: Mean path length ratios

unification that is performed as well as by the efficiency of the search. We can measure the latter by looking at the number of nodes on the search path to a solution. Again, there are large differences in performance among the algorithms. Figure 6.7 shows the average path length to solution by problem complexity for the four algorithms. For problems that were not solved within the time limit, the length of the search path at the time of stopping was used. This is, of course, an underestimate and introduces the same types of distortion that we saw in the previous section.

Figure 6.8 gives some idea of the relative search efficiency of each algorithm on those problems that they solve. For each algorithm the path length to solution was divided by the length of the path taken by SNLP-SP, and the graph shows the means of these ratios. Only those problems that were solved by the algorithm under consideration are included. Again, the results are affected by the closeness of the solution time to the time limit for SNLP-F on some problems with three solution steps. If the time limit had been slightly lower, the results could have been significantly different. However, it appears clear that SNLP-B and SNLP-F search much less efficiently than SNLP-SP, with path lengths of over ten times as long. The effect is less marked for SNLP-FF, which has an average path length of about three times the length of that of SNLP-SP.

Comparison	SNLP-B to SNLP-F		SNLP-F to SNLP-FF		SNLP-FF to SNLP-SP	
Complexity	Mean	T value	Mean	T value	Mean	T value
0	0		0		0	
1	* -0.8	* -0.880	22.4	2.761	0.8	8.062
2	* -27.2	* -0.854	243.1	6.638	9.2	10.748
3	-824.2	-6.453	1216.8	8.930	44.6	8.318
4			* 55.7	* 1.652	129.0	4.425
Total	-224.7	-4.903	396.4	7.122	30.7	6.622

Figure 6.9: Mean path length comparisons

The results of the pairwise T-test are shown in Figure 6.9, by problem complexity and in total. The length of path at stopping is used for problems that were not solved, which is an underestimate. Cells in which no problems were solved are not shown, but are included in the total. All results are significant at the 95% level except those marked with an asterisk. These results show that SNLP-F performs worse than SNLP-B, and that the overall comparison is significant. SNLP-SP performs significantly better than SNLP-FF which in turn outperforms SNLP-F. These results are affected by including unsolved problems, which results in understating the difference between algorithms.

6.5 Branching factor

In these algorithms based on SNLP, the branching factor of the search space depends on the number of modifications that can be made to a partial plan. There are two situations in which a partial plan is modified: in order to protect a threatened link, and in order to add a new link. The use of secondary preconditions increases the number of ways of protecting threatened links, but typically decreases the number of ways in which a new link can be added, due to the reduction in the number of operators required to represent a domain. It is therefore not immediately clear how the branching factor of the search space will be affected by the introduction of secondary preconditions.

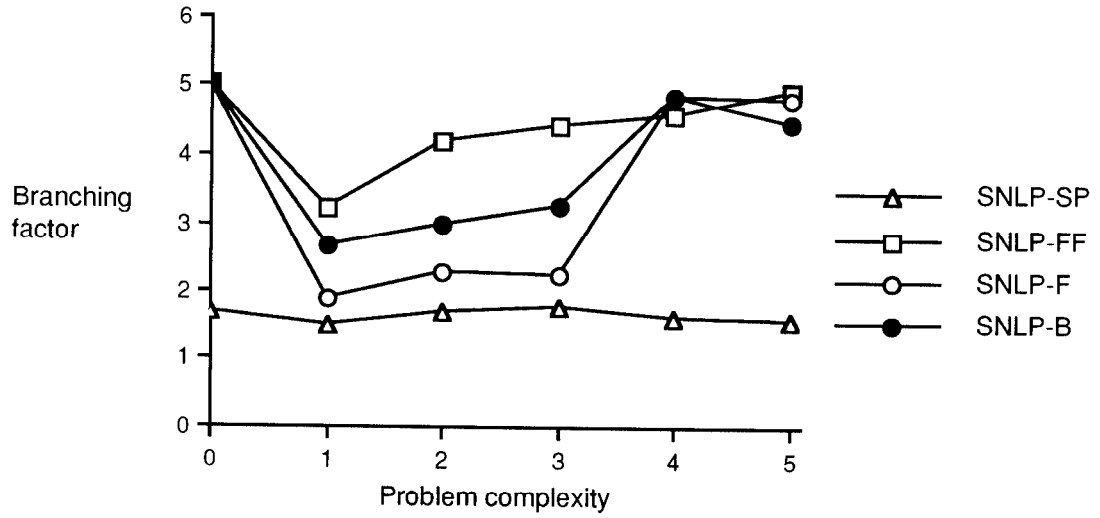


Figure 6.10: Mean branching factor—all problems

Figure 6.10 shows the mean branching factors for the four algorithms under consideration for all problems, whether solved or not. It appears that SNLP-SP has a much lower branching factor, with the advantage increasing on the more complex problems. Other things being equal, a low branching factor is an advantage, and we demonstrated in the previous section that SNLP-SP typically has a shorter search path to solution than the other algorithms.

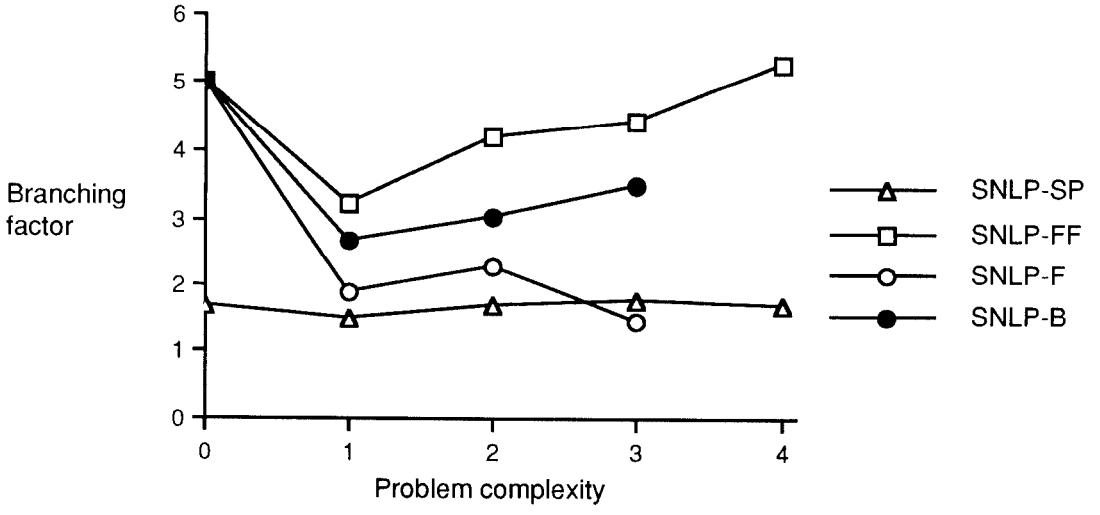


Figure 6.11: Mean branching factor—solved problems

An indication of how the branching factor affects the search process might be gained by looking at the difference between the searching factor on those problems that were solved within the time limit and those that were not. Figure 6.11 shows the mean branching factor for only those problems that were solved. The mean branching factors for SNLP-B and SNLP-F are much lower than for all problems, many of which these two algorithms found difficult (they failed to solve them within the time limit). This may indicate that, for this type of algorithm, the branching factor is a strong influence on the ease with which a problem can be solved. From the graphs, it appears that the effect may increase with the complexity of the problem, giving rise to the expectation that the use of secondary preconditions may be more advantageous with more complex problems. However, this effect could very possibly be a result of the particular domain we used for these experiments, which was designed to highlight the influence of context-dependent effects.

7 Conclusions

Many researchers have modified the basic STRIPS representation by introducing filter conditions. In this paper we have analyzed the purposes to which filter conditions have been put, and concluded that in only one case are they actually appropriate for the purpose, namely as a method of representing the context-dependent effects of operators. We then considered the problem of implementing filter conditions in a partial order planner while preserving the completeness and correctness of the planning algorithm. We concluded that it is possible to construct such an implementation, but that it will be inefficient. Pednault's secondary preconditions, however, provide a good way of achieving the required functionality.

We implemented secondary preconditions in a version of SNLP, a partial order planner, and performed an empirical comparison of filter conditions with secondary preconditions. The modified algorithm we present is provably correct (see Appendix A), and it improves on the performance of the unmodified algorithm. We believe that the theoretical analysis of planning algorithms should not ignore the functional considerations faced by practical planners, and that such analysis can provide valuable help to practical planners. In this paper we have shown how these two approaches to planning can be combined.

Acknowledgments

Thanks to Larry Birnbaum, Matt Brand, Mike Freed and Bruce Krulwich for many useful discussions, to Brian Drabble, Austin Tate, Dan Weld and the AAAI reviewers for their comments on parts of earlier drafts, and to Dan Weld for supplying the SNLP code. Parts of this work first appeared in (Collins & Pryor, 1992).

References

Allen, J., Hendler, J., & Tate, A. (Ed.). (1990). *Readings in Planning*. San Mateo, CA: Morgan Kaufmann.

Barrett, A., Soderland, S., & Weld, D. S. (1991). *Effect of Step-Order Representations on Planning* Technical Report 91-05-06. Department of Computer Science and Engineering, University of Washington, Seattle.

Barrett, A., & Weld, D. S. (1992). *Partial-Order Planning: Evaluating Possible Efficiency Gains* Technical Report 92-05-01. Department of Computer Science and Engineering, University of Washington.

Chapman, D. (1987). Planning for Conjunctive Goals. *Artificial Intelligence, 32*, 333-337. Also in (Allen *et al* 1990).

Charniak, E., & McDermott, D. (1985). *Introduction to artificial intelligence.* Reading, MA: Addison-Wesley.

Chien, S. A. (1990). *An Explanation-Based Learning Approach to Incremental Planning* Report No. UIUCDCS-R-90-1646. Department of Computer Science, University of Illinois at Urbana-Champaign.

Chien, S. A., & DeJong, G. F. (1992). Constructing Simplified Plans via Truth Criteria Approximation. In *Proceedings of the AAAI Workshop on Abstraction and Approximation of Computational Theories*, San Jose, CA: AAAI.

Collins, G., & Pryor, L. (1992). Achieving the functionality of filter conditions in a partial order planner. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, CA: AAAI.

Currie, K., & Tate, A. (1985). O-Plan—Control in the Open Planning Architecture. *Expert Systems, 85*, 225-240. .

Currie, K., & Tate, A. (1991). O-Plan: the open planning architecture. *Artificial Intelligence, 52*, 49-86. .

Feldman, R., & Morris, P. (1990). Admissible criteria for loop control in planning. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, MA: AAAI.

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence, 2*, 189-208. .

McAllester, D., & Rosenblitt, D. (1991). Systematic Nonlinear Planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, (pp. 634-639). Anaheim, CA: AAAI.

McDermott, D. (1989). *Regression Planning* Technical Report YALEU/CSD/RR #752. Computer Science Department, Yale University.

Newell, A., & Simon, H. A. (1963). GPS, a program that simulates human thought. In E. A. Feigenbaum & J. Feldman (Eds.), *Computers and Thought* (pp. 179-293). New York: McGraw-Hill. Also in (Allen *et al* 1990).

Pednault, E. P. D. (1988a). Extending Conventional Planning Techniques to Handle Actions with Context-Dependent Effects. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, St Paul, MN: AAAI.

Pednault, E. P. D. (1988b). Synthesizing plans that contain actions with context-dependent effects. *Computational Intelligence, 4*, 356-372. .

Pednault, E. P. D. (1989). ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, .

Pednault, E. P. D. (1991). Generalizing Nonlinear Planning to Handle Complex Goals and Actions with Context-Dependent Effects. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, Sydney, Australia.

Penberthy, J. S., & Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, Boston, MA: Morgan Kaufmann.

Tate, A. (1977). Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA: IJCAI. Also in (Allen *et al* 1990).

Wilkins, D. E. (1988). *Practical Planning: Extending the Classical AI Planning Paradigm*. San Mateo, CA: Morgan Kaufmann.

Appendix A: Formal descriptions

This appendix contains formal algorithm specifications and sketches of proofs. They follow closely those in (McAllester & Rosenblitt, 1991) and (Barrett & Weld, 1992), where further details my be found.

A.1 The basic SNLP algorithm

We start with some basic definitions:

Definition A.1.1: A *STRIPS operator schema* consists of an operator name, a *precondition list*, an *add list*, a *delete list*, and a set of *binding constraints*. The elements of the precondition, add, and delete lists are all proposition expressions. the elements of the set of binding constraints mention only those variables appearing elsewhere in the schema. A *STRIPS planning problem* is a triple $<\vartheta, \Sigma, \Omega>$, where ϑ is a set of STRIPS operator schemas, Σ is a set of initial propositions, and Ω is a set of goal propositions.

Definition A.1.2: A *solution* to a STRIPS planning problem $<\vartheta, \Sigma, \Omega>$ is a sequence α of operator schemas in ϑ together with a set of binding constraints, such that the result of successively applying the operators in α, instantiated according to the binding constraints, starting in the initial state Σ results in a state containing the goal state Ω.

Definition A.1.3: A *symbol table* T is a mapping from a finite set of step names to operator schemas in ϑ and an associated set of binding constraints. Every symbol table must contain START, a step name mapped to an operator that has empty precondition and delete lists and a set of initial conditions as an add list, and FINISH, a step name mapped to an operator that has empty add and delete lists and a set of goal conditions as a precondition list.

Hereafter when we refer to a step name's precondition list, add list or delete list we mean the lists of the associated operator schema instantiated in a manner consistent with the binding constraints. So far these definitions have been fairly standard: we now move on to some that are more specifically concerned with the SNLP algorithm.

Definition A.1.4: A *causal link* is a triple $<s, P, w>$ where P is a proposition symbol, w is a step name in T that has P as a precondition, and s is a step name that has P in its add list. Causal links are written as $s \xrightarrow{P} w$.

Definition A.1.5: A step name v in T is a *threat* to a causal link $s \xrightarrow{P} w$ if v has a proposition that possibly unifies with P in its add list or delete list, and v is not s or w. An *ordering constraint* is an ordering $s_i < s_j$ or $s_j > s_i$ where s_i and s_j are step names in T.

Causal links are the way in which effects are established; a threat to a link is a possibly clobbering step.

Definition A.1.6: A *partial order plan* is a tuple <T, *L*, *O*, G> where T is a symbol table, *L* is a set of causal links, and *O* is a set of ordering constraints. G is the set of open conditions *P* such that *P* is a precondition of some *w* in T, and there is no causal link in *L* of the form $s \xrightarrow{P} w$.

Definition A.1.7: A partial order plan <T, *L*, *O*, G>is *complete* if:

- Every step name appearing in the elements of *L* or *O* is a member of T.
- G is empty.
- If *L* contains a causal link $s \xrightarrow{P} w$ and T contains a step name *v* that is a threat to $s \xrightarrow{P} w$, then *O* contains either the ordering constraint $v < s$ or the ordering constraint $v > w$.

In other words, a plan is complete if the preconditions of every step are established by links (there are no open conditions), and if every step that threatens a link is constrained to either precede or follow the threatened link. Note that the goal conditions are preconditions of the FINISH step, and the initial conditions may be used to establish preconditions because they comprise the add list of the START step.

Definition A.1.8: A *topological sort* of a partial order plan <T, *L*, *O*, G> is a linear sequence of all the members of T such that:

- The first step in the sequence is START.
- The last step in the sequence is FINISH.
- For each causal link $s \xrightarrow{P} w$ in *L*, the step *s* precedes the step *w*.
- For each ordering constraint $u < v$ (or $v > u$) in *O*, the step *u* precedes the step *v*.

A topological sort of a partial order plan is a *solution* if executing the sequence of operators of the steps between the START and FINISH steps, starting in the state given by the add list of the START step, results in a state that contains all the preconditions of the FINISH step. A partial order plan is called *order inconsistent* if it has no topological sort.

These definitions lead directly to the following:

Lemma: Any topological sort of a complete partial order plan is a solution.

We now define a nondeterministic procedure that will find the completion of a partial plan if such a completion exists.

Procedure FIND-COMPLETION <T, *L*, *O*, G>

1. If <T, *L*, *O*, G> is order inconsistent then fail.

2. If <T, *L*, *O*, G> is complete then return <T, *L*, *O*, G>.

3. If there is a causal link $s \xrightarrow{P} w$ in L and a threat v to this link with v in T with neither $v<s$ nor $v>w$ in O, then nondeterministically do one of:
   a) Return FIND-COMPLETION <T, L, O + ($v<s$), G>
   b) Return FIND-COMPLETION <T, L, O + ($v>w$), G>
   c) Find (nondeterministically) binding constraints b that prevent the possible unification of the threatening add or delete condition of v with P, then return FIND-COMPLETION <T+ b, L, O, G>

4. There must now be some P in G, i.e. there is some step name w in T with a precondition P such that there is no causal link of the form $s \xrightarrow{P} w$ in L. Nondeterministically do one of:
   a) Let s be (nondeterministically) some step name in T and b be (nondeterministically) binding constraints such that s has an add condition that possibly unifies with P consistent with b and return the plan FIND-COMPLETION <T + b, $L + s \xrightarrow{P} w$, O , G - P>.
   b) Select (nondeterministically) an operator schema o_i with preconditions p_i from the allowed set of operator schemas and select (nondeterministically) binding constraints b such that an add condition of o_i possibly unifies with P. Create a new entry in T that maps a new step name s to o_i, and return the plan FIND-COMPLETION <T+s + b, L + $s \xrightarrow{P} w$, O , G - P + p_i>.

It is straightforward to check that every completion of <T, L, O, G> is equivalent (up to renaming of steps) to a value found by this procedure. This procedure is therefore a complete and correct planning algorithm. It can also be shown that no two distinct execution paths can produce equivalent complete plans.

A.2 The SNLP^S algorithm

The formal specification of the SNLP algorithm modified to include secondary preconditions is very similar to that of the basic algorithm. Again, we start with basic definitions.

Definition A.2.1: A *STRIPS^S operator* consists of an operator name, a *precondition list*, an *add list*, a *delete list*, and a set of *binding constraints*. The elements of the precondition list are proposition expressions. The elements of the add list each consist of a proposition expression (an *add condition*) together with a set of proposition expressions and binding constraints (the *causation preconditions*). The elements of the delete list each consist of a proposition expression (a *delete condition*) together with a set of proposition expressions and binding constraints (the *preservation preconditions*). The elements of the set of binding constraints mention only those variables appearing elsewhere in the schema. A *STRIPS^S planning problem* is a triple <ϑ, Σ, Ω>, where ϑ is a set of STRIPS^S operators, Σ is a set of initial propositions, and Ω is a set of goal propositions.

Definition A.2.2: A *solution* to a STRIPS^S planning problem <ϑ, Σ, Ω> is a sequence α of operator schemas in ϑ together with a set of binding constraints, such that the result of successively

applying the operators in α, instantiated according to the binding constraints, starting in the initial state Σ results in a state containing the goal state Ω.

Definition A.2.3: A *symbol table* T is a mapping from a finite set of step names to operator schemas in ϑ and an associated set of binding constraints. Every symbol table must contain START, a step name mapped to an operator that has empty precondition and delete lists and a set of initial conditions as an add list, and FINISH, a step name mapped to an operator that has empty add and delete lists and a set of goal conditions as a precondition list.

Hereafter when we refer to a step name's precondition list, add list or delete list we mean the lists of the associated operator schema instantiated in a manner consistent with the binding constraints. So far these definitions have differed from those in the previous section only as a result of the different operator format.

Definition A.2.4: A *causal link* is a triple <*s*, *P*, *w*> where *P* is a proposition symbol, *w* is a step name in T that has *P* as an enabling precondition, a causation precondition, or a preservation precondition, and *s* is a step name that has *P* as an add condition. Causal links are written as $s \xrightarrow{P} w$. A *prevention link* is a triple <*s*, *R w*> where *R* is a proposition symbol, *w* is a step name in T that has *R* as an enabling precondition, a causation precondition, or a preservation precondition, and *s* is a step name that has *R* as a delete condition. Prevention links are written as $s \overset{R}{\nrightarrow} w$.

Definition A.2.5: A step name *v* in T is a *threat* to a causal link $s \xrightarrow{P} w$ or a prevention link $s \overset{R}{\nrightarrow} w$ if *v* has an add condition or a delete condition that possibly unifies with *P*, and if v is not *s* or *w*. An *ordering constraint* is an ordering $s_i < s_j$ or $s_j > s_i$ where s_i and s_j are step names in T.

Causal links establish effects, as before; prevention links prevent the establishment of effects. A threat to either type of link is a possibly clobbering step.

Definition A.2.6: A *partial order plan* is a tuple <T, *L*, *O*, G, F> where T is a symbol table, *L* is a set of causal and prevention links, and *O* is a set of ordering constraints.

G is the set of open conditions *P* such that either:

- *P* is a precondition of some *w* in T, and there is no causal link in *L* of the form $s \xrightarrow{P} w$ or
- There is some causal link $s \xrightarrow{P} w$ in L (so that *P* is an add condition of *s*), and *Q* is a causal precondition associated with *P* in s, and there is no causal link in L of the form $u \overset{Q}{\nrightarrow} s$.

F is the set of negated-open conditions such that:

- There is some prevention link $s \overset{R}{\nrightarrow} w$ in *L* (so that *R* is a delete condition of *s*), and *Q* is a preservation precondition associated with *R* in *s*, and there is no prevention link in L of the form $u \overset{Q}{\nrightarrow} s$.

Definition A.2.7: A partial order plan <T, *L*, *O*, G, F> is *complete* if:

- Every step name appearing in the elements of *L* or *O* is a member of T.

- G is empty.
- F is empty
- If L contains a causal link $s \xrightarrow{P} w$ (so that P is an add condition of s), and T contains a step name v that is a threat to $s \xrightarrow{P} w$, then either:
  - O contains the ordering constraint $v < s$ or
  - O contains the ordering constraint $v > w$ or
  - L contains a prevention link of the form $u \overset{Q}{\nrightarrow} s$ for at least one causation precondition Q associated with P.
- If L contains a prevention link $s \overset{R}{\nrightarrow} w$ (so that R is a delete condition of s), and T contains a step name v that is a threat to $s \xrightarrow{P} w$, then either:
  - O contains the ordering constraint $v < s$ or
  - O contains the ordering constraint $v > w$ or
  - L contains a causation link of the form $u \overset{Q}{\rightarrow} s$ for all preservation preconditions Q associated with R.

In other words, a plan is complete if the preconditions of every step and of every effect used to establish a precondition are established by links, and if every step that threatens a link is either constrained to precede or follow the threatened link, or the effect that would threaten the link is prevented from occurring. Note that the goal conditions are preconditions of the FINISH step, and the initial conditions may be used to establish preconditions because they comprise the add list of the START step.

This definition of a complete plan is the crux of the proof of the completeness and correctness of the modified SNLP algorithm. The proof proceeds by demonstrating that the procedure used by the algorithm to construct completions of partial order plans results in a complete plan as defined here.

Definition A.2.8: A topological sort of a partial order plan <T, L, O, G, F> is a linear sequence of all the members of T such that:

- The first step in the sequence is START.
- The last step in the sequence is FINISH.
- For each causal link $s \xrightarrow{P} w$ in L, the step s precedes the step w.
- For each prevention link $s \overset{R}{\nrightarrow} w$ in L, the step s precedes the step w.
- For each ordering constraint $u < v$ (or $v > u$) in O, the step u precedes the step v.

A topological sort of a partial order plan is a *solution* if executing the sequence of operators of the steps between the START and FINISH steps, starting in the state given by the add list of the START step, results in a state that contains all the preconditions of the FINISH step. A partial order plan is called *order inconsistent* if it has no topological sort.

These definitions lead directly to the following:

Lemma: Any topological sort of a complete partial order plan is a solution.

We now define a nondeterministic procedure that will find the completion of a partial plan if such a completion exists.

Procedure FIND-COMPLETION^S <T, *L*, *O*, G, F>

1. If <T, *L*, *O*, G, F> is order inconsistent then fail.

2. If <T, *L*, *O*, G, F> is complete then return <T, *L*, *O*>.

3. If there is a threatened link of either type then nondeterministically do one of:

   i) If there is a causal link $s \xrightarrow{P} w$ in *L* (so that *P* is an add condition of *s*) and a threat *v* to this link with *v* in T with neither $v<s$ nor $v>w$ in *O*, nor any prevention links of the form $u \overset{Q}{\nrightarrow} s$ in *L* where *Q* is a causal precondition associated with *P* in *s* then nondeterministically do one of:
      a) Return FIND-COMPLETION^S<T, *L*, *O* + ($v<s$), G, F>
      b) Return FIND-COMPLETION^S <T, *L*, *O* + ($v>w$), G, F>
      c) Find (nondeterministically) binding constraints *b* that prevent the possible unification of the threatening add or delete condition of *v* with *P*, then return FIND-COMPLETION^S <T+ *b*, *L*, *O*, G, F>
      d) Let *Q* be (nondeterministically) a causal precondition associated with *P* in *s* and return FIND-COMPLETION^S <T+ *b*, *L*, *O*, G, F + *Q*>

   ii) If there is a prevention link $s \overset{R}{\nrightarrow} w$ in *L* and a threat *v* to this link with *v* in T with neither $v<s$ nor $v>w$ in *O*, then nondeterministically do one of:
      a) Return FIND-COMPLETION^S <T, *L*, *O* + ($v<s$), G, F>
      b) Return FIND-COMPLETION^S <T, *L*, *O* + ($v>w$), G, F>
      c) Find (nondeterministically) binding constraints *b* that prevent the possible unification of the threatening add or delete condition of *v* with *P*, then return FIND-COMPLETION^S <T+ *b*, *L*, *O*, G, F>
      d) Let Q_i be (nondeterministically) the preservation preconditions associated with *R* in *s* and return FIND-COMPLETION^S <T+ *b*, *L*, *O*, G + Q_i, F>

4. Either G or F is now non-empty. Nondeterministically do one of:

   i) If G is non-empty, so there is some step name *w* in T with a precondition *P* such that there is no causal link of the form $s \xrightarrow{P} w$ in *L*, then nondeterministically do one of:
      a) Let *s* be (nondeterministically) some step name in T and *b* be (nondeterministically) binding constraints such that *s* has an add condition *A* (with associated causation preconditions A_i) that possibly unifies with *P* consistent with *b* and return the plan FIND-COMPLETION^S <T + *b*, *L* + $s \xrightarrow{P} w$, *O* , G - *P* + A_i, F>
      b) Select (nondeterministically) an operator schema o_i (with preconditions P_i) from the allowed set of operator schemas and select (nondeterministically) binding constraints *b* such that an add condition *A* (with associated causation preconditions

A_i) of o_i possibly unifies with P. Create a new entry in T that maps a new step name s to o_i, and return the plan FIND-COMPLETION^S <T+s + b, $L + s \xrightarrow{P} w$, O , G - P + P_i + A_i, F>.

i) If F is non-empty, so there is some prevention link $s \overset{R}{\nrightarrow} w$ in L (so that R is a delete condition of s), and Q is a preservation precondition associated with R in s, and there is no prevention link in L of the form $u \overset{Q}{\nrightarrow} s$, then nondeterministically do one of:
   a) Let v be (nondeterministically) some step name in T and b be (nondeterministically) binding constraints such that v has a delete condition D (with preservation preconditions D_i) that possibly unifies with Q consistent with b and return the plan FIND-COMPLETION^S <T + b, $L + v \overset{Q}{\nrightarrow} u$, O , G, F - Q + D_i>
   b) Select (nondeterministically) an operator schema o_i (with preconditions P_i) from the allowed set of operator schemas and select (nondeterministically) binding constraints b such that a delete condition D (with preservation preconditions D_i) of o_i possibly unifies with Q consistent with b. Create a new entry in T that maps a new step name v *to* o_i, and return the plan FIND-COMPLETION^S <T+v + b, $L + v \overset{Q}{\nrightarrow} u$, O , G + P_i, F - Q + D_i>.

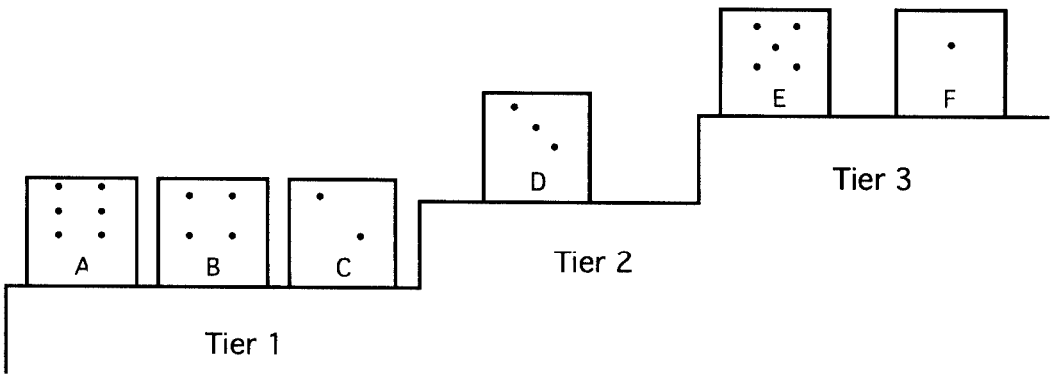
It is straightforward to check that every completion of <T, L, O, G, F> is equivalent (up to renaming of steps) to a value found by this procedure. This procedure is therefore a complete and correct planning algorithm. It can also be shown that no two distinct execution paths can produce equivalent complete plans.

Appendix B: Test domain

We tested the planning algorithms described in this paper on an artificial domain designed to highlight the effects of context-dependent effects and operator selection on planning efficiency. The domain consists of a series of tiers, each of which can hold an infinite number of blocks. Each block may be in any of six orientations (like dice) (figure B.1). There is just one type of action:

Raise moves a block up a tier while changing its orientation by a quarter turn to the next position in a cycle of the six.

In order to move a block from a tier, there must be another block on the same tier (no tier may be left empty as the result of a move). Goals are expressed as a required position, for example (on A tier3). This domain therefore includes many context-dependent effects, as the orientation of a block after an action depends on its orientation before the action but not on the particular action that is performed. The domain also isolates the effects of context-dependency: none of the other possible uses of filter conditions are possible, as loops are impossible, and all actions are equally expensive.



In order to move block D to Tier 3, one of A,B,C must be moved first. After D has been moved, it will have face 4 uppermost.

Figure B.1: Our test domain.

B.1 Domain operators

In this section we show the action representations that we used with the various algorithms. Note that the basic STRIPS representation and the representation using filter conditions both require 12 operators, while the representation using secondary preconditions requires just one.

B.1.1 Basic STRIPS operators

Raise a block from tier 1 starting in orientations 1 through 3:

Action:	(move11 ?block)	(move12 ?block)	(move13 ?block)
Preconditions:	(on ?block tier1) (on ?other tier1) (up ?block face1)	(on ?block tier1) (on ?other tier1) (up ?block face2)	(on ?block tier1) (on ?other tier1) (up ?block face3)
Delete list:	(on ?block tier1) (up ?block face1)	(on ?block tier1) (up ?block face2)	(on ?block tier1) (up ?block face3)
Add list:	(on ?block tier2) (up ?block face2)	(on ?block tier2) (up ?block face3)	(on ?block tier2) (up ?block face4)
Codesignation constarints:	(not (?block ?other))	(not (?block ?other))	(not (?block ?other))

Raise a block from tier 1 starting in orientations 4 through 6:

Action:	(move14 ?block)	(move15 ?block)	(move16 ?block)
Preconditions:	(on ?block tier1) (on ?other tier1) (up ?block face4)	(on ?block tier1) (on ?other tier1) (up ?block face5)	(on ?block tier1) (on ?other tier1) (up ?block face6)
Delete list:	(on ?block tier1) (up ?block face 4)	(on ?block tier1) (up ?block face5)	(on ?block tier1) (up ?block face6)
Add list:	(on ?block tier2) (up ?block face5)	(on ?block tier2) (up ?block face6)	(on ?block tier2) (up ?block face1)
Codesignation constarints:	(not (?block ?other))	(not (?block ?other))	(not (?block ?other))

Raise a block from tier 2 starting in orientations 1 through 3:

Action:	(move21 ?block)	(move22 ?block)	(move23 ?block)
Preconditions:	(on ?block tier2) (on ?other tier2) (up ?block face1)	(on ?block tier2) (on ?other tier2) (up ?block face2)	(on ?block tier2) (on ?other tier2) (up ?block face3)
Delete list:	(on ?block tier2) (up ?block face1)	(on ?block tier2) (up ?block face2)	(on ?block tier2) (up ?block face3)
Add list:	(on ?block tier3) (up ?block face2)	(on ?block tier3) (up ?block face3)	(on ?block tier3) (up ?block face4)
Codesignation constarints:	(not (?block ?other))	(not (?block ?other))	(not (?block ?other))

Raise a block from a tier 2 starting in orientations 4 through 6:

Action:	(move24 ?block)	(move25 ?block)	(move26 ?block)
Preconditions:	(on ?block tier2) (on ?other tier2) (up ?block face4)	(on ?block tier2) (on ?other tier2) (up ?block face5)	(on ?block tier2) (on ?other tier2) (up ?block face6)
Delete list:	(on ?block tier2) (up ?block face 4)	(on ?block tier2) (up ?block face5)	(on ?block tier2) (up ?block face6)
Add list:	(on ?block tier3) (up ?block face5)	(on ?block tier3) (up ?block face6)	(on ?block tier3) (up ?block face1)
Codesignation constarints:	(not (?block ?other))	(not (?block ?other))	(not (?block ?other))

B.1.2 STRIPS operators with filter conditions

Raise a block from tier 1 starting in orientations 1 through 3:

Action:	(move11 ?block)	(move12 ?block)	(move13 ?block)
Filter conditions:	(up ?block face1)	(up ?block face2)	(up ?block face3)
Preconditions:	(on ?block tier1) (on ?other tier1)	(on ?block tier1) (on ?other tier1)	(on ?block tier1) (on ?other tier1)
Delete list:	(on ?block tier1) (up ?block face1)	(on ?block tier1) (up ?block face2)	(on ?block tier1) (up ?block face3)
Add list:	(on ?block tier2) (up ?block face2)	(on ?block tier2) (up ?block face3)	(on ?block tier2) (up ?block face4)
Codesignation constarints:	(not (?block ?other))	(not (?block ?other))	(not (?block ?other))

Raise a block from tier 1 starting in orientations 4 through 6:

Action:	(move14 ?block)	(move15 ?block)	(move16 ?block)
Filter conditions:	(up ?block face4)	(up ?block face5)	(up ?block face6)
Preconditions:	(on ?block tier1) (on ?other tier1)	(on ?block tier1) (on ?other tier1)	(on ?block tier1) (on ?other tier1)
Delete list:	(on ?block tier1) (up ?block face 4)	(on ?block tier1) (up ?block face5)	(on ?block tier1) (up ?block face6)
Add list:	(on ?block tier2) (up ?block face5)	(on ?block tier2) (up ?block face6)	(on ?block tier2) (up ?block face1)
Codesignation constarints:	(not (?block ?other))	(not (?block ?other))	(not (?block ?other))

Raise a block from tier 2 starting in orientations 1 through 3:

Action:	(move21 ?block)	(move22 ?block)	(move23 ?block)
Filter conditions:	(up ?block face1)	(up ?block face2)	(up ?block face3)
Preconditions:	(on ?block tier2) (on ?other tier2)	(on ?block tier2) (on ?other tier2)	(on ?block tier2) (on ?other tier2)
Delete list:	(on ?block tier2) (up ?block face1)	(on ?block tier2) (up ?block face2)	(on ?block tier2) (up ?block face3)
Add list:	(on ?block tier3) (up ?block face2)	(on ?block tier3) (up ?block face3)	(on ?block tier3) (up ?block face4)
Codesignation constarints:	(not (?block ?other))	(not (?block ?other))	(not (?block ?other))

Raise a block from a tier 1 starting in orientations 4 through 6:

Action:	(move24 ?block)	(move25 ?block)	(move26 ?block)
Filter conditions:	(up ?block face4)	(up ?block face5)	(up ?block face6)
Preconditions:	(on ?block tier2) (on ?other tier2)	(on ?block tier2) (on ?other tier2)	(on ?block tier2) (on ?other tier2)
Delete list:	(on ?block tier2) (up ?block face 4)	(on ?block tier2) (up ?block face5)	(on ?block tier2) (up ?block face6)
Add list:	(on ?block tier3) (up ?block face5)	(on ?block tier3) (up ?block face6)	(on ?block tier3) (up ?block face1)
Codesignation constarints:	(not (?block ?other))	(not (?block ?other))	(not (?block ?other))

B.1.3 STRIPS operator with secondary preconditions

If secondary preconditions are used, just a single operator is needed in this domain:

Action:	(move ?block)
Preconditions:	(on ?block ?tier) (on ?other ?tier)
Codesignation constarints:	(not (?block ?other))

Each delete condition has associated preservation preconditions and codesignation constraints:

Delete condition	Preservation preconditions	Preservation codesignation constraints
(on ?block tier1)		(not (?tier tier1))
(on ?block tier2)		(not (?tier tier2))
(up ?block face1)	(up ?block ?f1)	(not (?f1 face1))
(up ?block face2)	(up ?block ?f2)	(not (?f2 face2))
(up ?block face3)	(up ?block ?f3)	(not (?f2 face3))
(up ?block face4)	(up ?block ?f4)	(not (?f2 face4))
(up ?block face5)	(up ?block ?f5)	(not (?f2 face5))
(up ?block face6)	(up ?block ?f6)	(not (?f2 face6))

Each add condition has associated causation preconditions and codesignation constraints:

Add condition	Causation preconditions	Causation codesignation constraints
(on ?block tier2)		(?tier tier1)
(on ?block tier3)		(?tier tier2)
(up ?block face1)	(up ?block face6)	
(up ?block face2)	(up ?b lock face1)	
(up ?block face3)	(up ?block face2)	
(up ?block face4)	(up ?block face3)	
(up ?block face5)	(up ?block face4)	
(up ?block face6)	(up ?block face5)	

B.2 Test problems

We constructed 150 random problems in our test domain, 50 each for problems involving one, two, and three blocks. Each problem included two extra blocks, not mentioned in the goals, that start on the lowest level. The problems were constructed so that a block specified in a goal condition starts out on a level equal to or lower than its goal level. The inclusion of the extra blocks ensures that all such problems are solvable. The orientation and level of the goal and extra blocks were fully specified in the initial conditions. The order of the conjuncts within the initial conditions and the goal conditions was randomized.

The algorithm we used to construct a problem with n goals was as follows:

Algorithm	Example
1. Make two lists, LISTA and LISTB, each of length n, each element randomly picked from integers 1, 2, 3.	LISTA = (3 2 1) LISTB = (1 3 2)
2. Make a list, LISTC, n elements long composed of pairs of elements from LISTA and LISTB, the lower number first in each pair.	LISTC = ((1 3) (2 3) (1 2))
3. Make a list, LISTI, composed of the first element in each pair of LISTC. This list will be used to provide the initial positions of the blocks in the problem.	LISTI = (1 2 1)
4. Make a list, LISTG, composed of the second element in each pair of LISTC. This list will be used to provide the goal positions of the blocks in the problem. We have now guaranteed that each block will start out on a level equal to or lower than its goal level.	LISTG = (3 3 2)
5. Make a list, LETTERS, consisting of the first n letters of the alphabet in random order.	LETTERS = (B A C)
6. The initial position of each element of LETTERS is given by the corresponding element in LISTI, and its initial orientation is chosen randomly from the integers one to six. Two extra blocks, X and Y, have initial postitions on level one and initial orientations chosen randomly. These $2n + 4$ initial conditions are formed into a list, the order of which is randomized.	Initial conditions: (up X face1) (on C tier1) (on B tier1) (up Y face5) (up B face5) (up A face4) (up C face3) (on Y tier1) (on A tier2) (on X tier1)
7. The goal position of each element of LETTERS is given by the corresponding element in LISTG, These n goal conditions are formed into a list, the order of which is randomized.	Goal conditions: (on C tier2) (on A tier3) (on B tier3)

Appendix C: Experimental results

We ran four algorithms on each problem, and analyzed three resulting statistics: the CPU time taken, the length of the search path, and the branching factor. The four algorithms were:

- SNLP-B, the basic form of the algorithm in which there are neither filter conditions nor secondary preconditions. The statistics resulting from this algorithm are CPUB, PATB, BCHB.

- SNLP-F, in which filter conditions are used in the action representations but are not used to guide the search (see section 4.1). The statistics resulting from this algorithm are CPUF, PATF, BCHF.

- SNLP-FF, in which filter conditions are used in the action representations and are used to guide the search (see section 4.2). The statistics resulting from this algorithm are CPUFF, PATFF, BCHFF.

- SNLP-SP, in which the algorithm is modified to use secondary preconditions. The statistics resulting from this algorithm are CPUSP, PATSP, BCHSP.

We also show for each problem the least number of steps in a plan that solves the problem. The number of goal conditions is shown in the first part of the the problem's identifying number. A statistic *in italics* denotes that the relevant algorithm failed to solve the problem within the time limit.

C.1 Problems with one goal condition

Prob id	Plan steps	CPUB	CPUF	CPUFF	CPUSP	PATB	PATF	PATFF	PATSP	BCHB	BCHF	BCHFF	BCHSP
1-1	0	13	10	94	52	1	1	1	1	7.000	7.000	7.000	2.000
1-2	0	15	6	6	4	1	1	1	1	1.000	1.000	1.000	1.000
1-3	1	453	225	37	23	24	24	4	3	1.958	1.042	2.500	1.333
1-4	0	12	18	13	6	1	1	1	1	7.000	7.000	7.000	2.000
1-5	3	*25112*	*25011*	2777	368	*509*	*2523*	53	15	*2.413*	*1.071*	5.151	1.800
1-6	0	6	7	6	5	1	1	1	1	1.000	1.000	1.000	1.000
1-7	0	6	8	7	4	1	1	1	1	1.000	1.000	1.000	1.000
1-8	3	3199	5825	2793	362	89	407	53	15	3.876	1.526	5.151	1.800
1-9	3	*25100*	25003	2678	458	*521*	2524	53	15	*2.342*	1.071	5.170	1.800
1-10	2	1682	1114	505	100	61	101	18	8	2.902	1.574	5.000	1.625
1-11	3	*25186*	6254	2804	346	*378*	498	53	15	*3.365*	1.428	5.151	1.800
1-12	0	6	14	6	4	1	1	1	1	1.000	1.000	1.000	1.000
1-13	3	3307	5728	2796	359	95	416	53	15	3.758	1.514	5.170	1.800
1-14	2	1792	1318	574	109	71	134	18	8	2.704	1.396	5.000	1.625
1-15	1	343	172	37	22	16	16	4	3	2.062	1.188	2.500	1.333
1-16	0	11	13	13	5	1	1	1	1	7.000	7.000	7.000	2.000
1-17	0	14	12	96	5	1	1	1	1	7.000	7.000	7.000	2.000
1-18	3	*25220*	*25013*	2962	350	*547*	*2750*	53	15	*2.289*	*1.066*	5.151	1.800
1-19	3	*25194*	*25004*	2736	342	*542*	*2758*	53	15	*2.378*	*1.066*	5.151	1.800
1-20	0	6	7	7	5	1	1	1	1	1.000	1.000	1.000	1.000
1-21	0	6	6	6	4	1	1	1	1	1.000	1.000	1.000	1.000
1-22	0	79	9	13	6	1	1	1	1	7.000	7.000	7.000	2.000
1-23	1	51	110	89	65	4	4	4	3	3.000	2.500	2.500	1.333
1-24	2	1040	837	527	103	46	73	18	8	3.022	1.863	5.000	1.625
1-25	0	93	10	12	5	1	1	1	1	7.000	7.000	7.000	2.000

Prob id	Plan steps	CPUB	CPUF	CPUFF	CPUSP	PATB	PATF	PATFF	PATSP	BCHB	BCHF	BCHFF	BCHSP
1-26	1	261	169	115	23	12	12	4	3	2.167	1.333	2.500	1.333
1-27	1	259	154	38	55	12	12	4	3	2.167	1.333	2.500	1.333
1-28	0	12	9	13	5	1	1	1	1	7.000	7.000	7.000	2.000
1-29	0	27	10	20	6	1	1	1	1	7.000	7.000	7.000	2.000
1-30	2	2716	1630	520	103	100	178	18	8	2.540	1.258	5.000	1.625
1-31	3	*25181*	25002	2810	345	*943*	2536	53	15	*1.655*	1.070	5.151	1.800
1-32	3	*25205*	*25019*	2705	385	*559*	*2752*	53	15	*2.315*	*1.066*	5.151	1.800
1-33	0	12	9	12	6	1	1	1	1	7.000	7.000	7.000	2.000
1-34	2	639	672	560	117	30	54	18	8	3.733	2.222	5.000	1.625
1-35	2	1584	970	492	109	59	98	18	8	2.949	1.602	5.000	1.625
1-36	2	1664	978	536	107	61	78	18	8	2.951	1.808	5.000	1.625
1-37	1	365	193	37	53	20	20	4	3	2.000	1.100	2.500	1.333
1-38	2	2718	1949	557	107	112	214	18	8	2.429	1.187	5.000	1.625
1-39	0	6	5	6	4	1	1	1	1	1.000	1.000	1.000	1.000
1-40	0	14	9	14	6	1	1	1	1	7.000	7.000	7.000	2.000
1-41	0	9	88	21	7	1	1	1	1	7.000	7.000	7.000	2.000
1-42	1	263	147	113	25	12	12	4	3	2.167	1.333	2.500	1.333
1-43	0	6	6	7	42	1	1	1	1	1.000	1.000	1.000	1.000
1-44	0	11	18	13	4	1	1	1	1	7.000	7.000	7.000	2.000
1-45	0	14	11	25	5	1	1	1	1	7.000	7.000	7.000	2.000
1-46	1	408	196	125	22	20	20	4	3	2.000	1.100	2.500	1.333
1-47	3	*25184*	25005	2792	357	*528*	2519	53	15	*2.356*	1.071	5.151	1.800
1-48	0	20	22	14	6	1	1	1	1	7.000	7.000	7.000	2.000
1-49	3	*25037*	*25018*	2724	397	*510*	*2539*	53	15	*2.420*	*1.070*	5.170	1.800
1-50	2	1232	1517	508	129	68	165	18	8	2.485	1.285	5.000	1.625

C.1 Problems with two goal conditions

Prob id	Plan steps	CPUB	CPUF	CPUFF	CPUSP	PATB	PATF	PATFF	PATSP	BCHB	BCHF	BCHFF	BCHSP
2-1	3	23784	*25057*	1181	670	694	*642*	26	23	2.905	*3.234*	3.846	1.783
2-2	2	1703	1526	628	301	38	57	13	7	4.868	3.193	4.692	1.714
2-3	2	1039	441	358	218	38	25	17	10	4.211	1.840	2.294	1.700
2-4	1	483	352	183	129	31	31	6	6	1.935	1.226	3.000	1.500
2-5	0	17	32	20	11	2	2	2	2	4.000	4.000	4.000	1.500
2-6	2	7519	3863	852	306	234	228	26	13	2.744	2.000	4.000	1.462
2-7	2	*25115*	3583	644	369	*520*	82	14	8	*2.904*	3.890	4.500	1.750
2-8	0	22	33	27	12	2	2	2	2	7.000	7.000	7.000	2.000
2-9	2	*25068*	13751	631	405	*614*	279	14	8	*2.704*	4.039	4.500	1.750
2-10	2	997	728	329	221	38	51	9	10	4.132	1.647	3.556	1.700
2-11	2	*25040*	22377	681	380	*704*	498	14	8	*2.396*	4.118	4.500	1.750
2-12	0	15	27	159	10	2	2	2	2	4.000	4.000	4.000	1.500
2-13	2	1677	1454	383	258	74	142	17	10	2.892	1.380	2.294	1.700
2-14	2	4503	2381	781	253	170	316	33	13	2.253	1.130	3.182	1.615
2-15	3	11148	*25063*	1186	831	300	*604*	26	23	3.573	*3.233*	3.846	1.783
2-16	3	*25016*	*25007*	2428	803	*510*	*746*	49	19	*3.075*	*3.083*	4.041	1.632
2-17	2	7452	4994	593	329	149	189	13	7	4.342	2.884	4.692	1.714
2-18	3	12864	*25007*	1110	733	350	*604*	26	23	2.854	*3.233*	3.846	1.783
2-19	3	*25316*	21541	3377	688	*501*	1518	52	14	*2.667*	1.424	5.346	1.786
2-20	3	*25021*	*25048*	3963	807	*332*	*354*	69	26	*5.479*	*5.511*	4.884	1.654
2-21	0	33	22	139	11	2	2	2	2	7.000	7.000	7.000	2.000
2-22	2	2530	949	538	311	58	37	13	7	3.534	3.189	4.692	1.714
2-23	2	896	930	770	326	29	32	26	13	4.069	3.594	4.000	1.462
2-24	0	32	38	42	18	2	2	2	2	7.000	7.000	7.000	2.000
2-25	1	1446	486	213	190	49	25	5	4	2.837	2.280	3.400	1.500

Prob id	Plan steps	CPUB	CPUF	CPUFF	CPUSP	PATB	PATF	PATFF	PATSP	BCHB	BCHF	BCHFF	BCHSP
2-26	2	16388	13771	742	449	441	301	14	8	2.492	4.130	4.500	1.750
2-27	2	880	912	823	352	29	32	26	13	4.069	3.594	4.000	1.462
2-28	4	*25085*	*25068*	*25092*	14434	*515*	*455*	*398*	214	*3.819*	*4.308*	*4.344*	1.421
2-29	4	*25067*	*25014*	*25036*	13253	*354*	*354*	*317*	198	*5.797*	*5.797*	*5.508*	1.591
2-30	5	*25063*	*25006*	*25101*	*25171*	*344*	*340*	*282*	*251*	*5.326*	*5.444*	*5.489*	*1.606*
2-31	4	*25062*	*25006*	15776	13219	*354*	*355*	233	198	*5.797*	*5.797*	5.240	1.591
2-32	5	*25129*	*25068*	*25079*	*25211*	*344*	*341*	*295*	*263*	*5.326*	*5.446*	*5.512*	*1.601*
2-33	2	*25080*	14803	603	375	*646*	318	14	8	*2.550*	4.110	4.500	1.750
2-34	2	7402	2269	352	184	221	293	23	10	2.416	1.154	1.739	1.600
2-35	3	6637	*25058*	1540	638	175	*632*	67	23	2.777	*3.242*	2.104	1.783
2-36	2	2457	2276	357	178	105	221	10	11	2.448	1.208	3.300	1.636
2-37	3	*25077*	*25661*	3364	621	*425*	*1727*	52	14	*2.807*	*1.405*	5.346	1.786
2-38	2	2710	1393	702	247	86	141	20	13	2.651	1.426	4.650	1.615
2-39	3	*25019*	*25081*	4990	728	*352*	*351*	160	26	*5.509*	*5.507*	2.662	1.654
2-40	3	*25014*	25003	3760	645	*420*	2773	90	25	*2.650*	1.071	3.467	1.640
2-41	2	*25026*	13959	724	352	*599*	279	14	8	*2.691*	4.039	4.500	1.750
2-42	3	*25078*	*25059*	4024	780	*354*	*354*	70	26	*5.511*	*5.511*	4.829	1.654
2-43	0	236	25	41	117	2	2	2	2	7.000	7.000	7.000	2.000
2-44	4	*25051*	*25159*	*25071*	12235	*372*	*370*	*328*	198	*5.806*	*5.805*	*5.524*	1.591
2-45	0	16	27	18	115	2	2	2	2	4.000	4.000	4.000	1.500
2-46	1	453	244	158	38	21	21	5	5	2.238	1.381	3.400	1.600
2-47	1	588	408	227	209	13	13	5	4	5.846	5.385	5.600	1.750
2-48	0	25	132	120	11	2	2	2	2	4.000	4.000	4.000	1.500
2-49	2	5186	5193	537	333	230	626	14	8	1.896	1.035	3.857	1.875
2-50	3	*25032*	*25082*	3777	783	*349*	*348*	62	24	*5.605*	*5.603*	5.306	1.708

C.3 Problems with three goal conditions

Prob id	Plan steps	CPUB	CPUF	CPUFF	CPUSP	PATB	PATF	PATFF	PATSP	BCHB	BCHF	BCHFF	BCHSP
3-1	2	*25009*	*25044*	1400	678	*604*	*988*	28	22	*2.613*	*1.910*	4.071	1.864
3-2	2	*25022*	*25097*	1234	659	*635*	*804*	22	10	*2.913*	*2.238*	5.682	1.900
3-3	5	*25005*	*25062*	*25099*	*25012*	*624*	*285*	*331*	*197*	*2.628*	*4.589*	*4.598*	*1.635*
3-4	1	1697	383	86	252	66	30	6	6	2.439	1.300	3.167	1.667
3-5	4	*25092*	*25001*	*25080*	*25091*	*278*	*289*	*336*	*355*	*5.644*	*5.633*	*4.943*	*1.546*
3-6	2	5678	5324	703	313	236	632	20	10	2.025	1.092	4.800	1.900
3-7	3	*25058*	*25016*	1331	957	*411*	*591*	19	13	*4.290*	*3.129*	4.842	1.923
3-8	3	*25102*	*25027*	4383	876	*705*	*917*	75	21	*2.407*	*1.949*	4.413	1.667
3-9	3	*25011*	*25002*	3192	909	*435*	*1548*	46	18	*3.313*	*1.553*	5.587	1.833
3-10	3	*25014*	*25075*	1284	1031	*432*	*602*	19	13	*4.285*	*3.108*	4.842	1.923
3-11	0	34	177	127	24	3	3	3	3	3.000	3.000	3.000	1.333
3-12	2	4096	11223	1150	601	160	280	28	16	2.088	3.321	4.071	1.625
3-13	4	*25070*	*25051*	*25135*	*25003*	*444*	*466*	*371*	*403*	*4.385*	*4.343*	*4.580*	*1.467*
3-14	0	37	38	163	24	3	3	3	3	7.000	7.000	7.000	2.000
3-15	1	840	1730	311	285	28	148	7	6	4.036	1.419	5.143	1.667
3-16	5	*25042*	*25018*	*25039*	*25119*	*283*	*296*	*240*	*262*	*5.880*	*5.885*	*5.883*	*1.595*
3-17	2	13235	1885	975	287	302	261	45	16	2.606	1.218	2.622	1.562
3-18	3	9542	11915	6571	1036	167	607	89	19	4.605	1.857	5.978	1.842
3-19	2	4586	2597	859	527	185	400	40	21	2.011	1.108	2.825	1.619
3-20	4	*25017*	*25104*	*25089*	*25187*	*299*	*298*	*373*	*371*	*5.301*	*5.299*	*4.416*	*1.523*
3-21	3	*25067*	*25149*	4193	1114	*313*	*1068*	54	15	*3.470*	*1.550*	5.519	1.867
3-22	3	*25004*	25005	1597	819	*493*	1075	41	29	*3.396*	1.702	3.683	1.690
3-23	1	1784	669	208	158	62	30	6	5	2.935	3.067	4.000	1.600
3-24	3	*25063*	*25012*	2548	1000	*667*	*588*	165	26	*3.081*	*3.709*	1.448	1.769
3-25	3	*25016*	*25057*	3959	1128	*363*	*2064*	90	28	*2.848*	*1.278*	3.533	1.571

Prob id	Plan steps	CPUB	CPUF	CPUFF	CPUSP	PATB	PATF	PATFF	PATSP	BCHB	BCHF	BCHFF	BCHSP
3-26	4	*25073*	*25035*	*25005*	*25018*	*287*	*370*	*584*	*362*	*6.000*	*4.616*	*2.366*	*1.561*
3-27	3	*25062*	*25048*	6865	1284	*558*	*1755*	131	28	*2.523*	*1.382*	4.382	1.679
3-28	4	*25002*	*25004*	18016	7977	*349*	*304*	217	80	*5.140*	*5.941*	5.406	1.912
3-29	2	*25058*	8705	1119	558	*629*	235	28	16	*2.405*	3.166	4.071	1.625
3-30	7	*25070*	*25573*	*25062*	*25075*	*268*	*276*	*422*	*247*	*5.239*	*5.217*	*4.408*	*1.538*
3-31	4	*25091*	*25010*	*25092*	*25222*	*562*	*490*	*409*	*426*	*3.254*	*3.757*	*3.819*	*1.319*
3-32	3	*25152*	25023	6568	896	*628*	843	105	25	*2.393*	2.163	4.667	1.600
3-33	0	217	45	46	27	3	3	3	3	5.000	5.000	5.000	1.667
3-34	4	*25049*	*25021*	*25241*	7923	*350*	*304*	*269*	80	*5.146*	*5.941*	*5.803*	1.912
3-35	3	*25038*	25003	4171	867	*416*	2009	89	18	*2.887*	1.285	3.562	1.722
3-36	5	*25136*	*25016*	*25129*	*25363*	*399*	*390*	*306*	*226*	*3.845*	*3.733*	*4.261*	*1.473*
3-37	3	*25118*	*25019*	2640	667	*540*	*850*	149	22	*3.196*	*2.065*	1.570	1.864
3-38	4	*25052*	*25017*	*25018*	14299	*439*	*405*	*370*	188	*4.084*	*4.514*	*4.492*	1.617
3-39	2	*25181*	4998	512	307	*322*	663	15	9	*3.478*	1.045	3.800	2.000
3-40	3	*25008*	*25023*	1078	1293	*375*	*944*	20	20	*4.107*	*2.140*	4.200	1.850
3-41	3	*25047*	*25165*	4176	1103	*457*	*908*	54	15	*4.022*	*1.707*	5.519	1.867
3-42	2	*25114*	*25025*	760	440	*365*	*567*	15	9	*3.603*	*3.780*	4.733	1.889
3-43	3	*25006*	*25059*	1272	1272	*766*	*749*	20	20	*2.201*	*2.501*	4.200	1.850
3-44	1	1077	441	270	142	34	18	6	5	3.059	3.222	4.000	1.600
3-45	4	*25159*	*25023*	18617	3265	*317*	*362*	236	42	*5.539*	*5.022*	5.242	1.762
3-46	4	*25032*	*25016*	*25106*	6210	*700*	*417*	*371*	78	*2.394*	*4.156*	*4.175*	1.590
3-47	3	*25040*	25015	1977	828	*565*	1249	74	26	*2.660*	1.606	2.486	1.769
3-48	2	5550	4268	680	386	114	170	15	11	3.974	2.518	4.267	1.636
3-49	5	*25018*	*25043*	*25065*	*25010*	*466*	*466*	*398*	*314*	*3.803*	*3.803*	*3.854*	*1.475*
3-50	4	*25045*	*25133*	*25112*	1950	*368*	*755*	*347*	31	*4.818*	*1.962*	*3.092*	1.839