Suspending Recursion in Causal-link Planning

David E. Smith
Rockwell Science Center
444 High St.
Palo Alto, CA 94301
de2smith@rpal.rockwell.com

Mark A. Peot
Rockwell Science Center
444 High St.
Palo Alto, CA 94301
peot@rpal.rockwell.com

Abstract

We present a strategy for suspending recursive open conditions during planning. We also show conditions under which plans with suspended open conditions can be pruned. To make this suspension and pruning strategy efficient, we use an *operator graph* to analyze potential recursion before the planning process begins.

This approach covers a broader range of recursive problems than the approaches of Morris and Kambhampati, and is much more tractable than Kambhampati's approach. We give experimental results that indicate 1) significant improvement on recursive problems and 2) negligible overhead when applied to recursive and non-recursive problems alike.

Introduction

Recursion is prevalent in many planning domains. In fact, any time some of the operations are reversible, there is the possibility of recursion in the planning process. For transportation problems recursion occurs whenever it is possible to travel both directions along a road or airway, or whenever it is possible to both load and unload cargo. In Russell's tire-changing problem, there is an operator for opening the car trunk and another for closing it. Both operators are necessary to achieve the goals of changing the tire and putting all the tools away. Unfortunately, the presence of these two operators allows arbitrarily long sequences of Open-trunk, Close-trunk, Open-trunk, Close-trunk ...

Recursion problems like this can often be alleviated by smart search strategies. For example, a planner can rank partial plans according to the amount of recursion and prefer to work on those partial plans with less recursion. Once a partial plan is chosen for elaboration, the planner can prefer working on open conditions that are not recursive. While such control strategies are often useful, if the planning problem has no solution, recursive plans and open conditions will eventually be found and doggedly explored by the planner until the end of time (or memory).

Many planning systems avoid recursion by pruning recursive subgoals (6, 7, 11, 14) or restricting the expansion of recursive open conditions (18). Feldman and Morris (5) have pointed out that these techniques are, in general, not admissable. (Sometimes they prune the only viable plans). In (5), Feldman and Morris prove certain conditions under which it is possible to stop expansion of recursive open conditions. In some cases, it is necessary for them to add filter preconditions to operators in order for the theorems to apply. It is not clear what fraction of recursion problems can, in fact, be handled by their method.

More recently, Kambhampati (9) has presented methods for pruning *non-minimal* partial plans during search. Roughly, a plan is non-minimal if (for every completion of the plan) a subset of steps can be removed from the plan. While the theory is general, there are two significant problems with its practical use:

1. The technique is computationally expensive for causal-link planners.
2. The conditions for pruning require that all remaining open-conditions in a partial plan temporally precede the root of the recursion (see (9) for a precise description). This means that a partial plan must be at an advanced stage of development before pruning is possible.

In this paper we also present admissable techniques for pruning recursion. In the section on Exact Recursion we present a method that suspends and prunes repeating open conditions. As with the techniques of Feldman and Morris, and Kambhampati, this method applies to recursion where the open condition is repeated exactly. The technique we develop is relatively cheap and catches recursion early.

While the exact recursion technique is applicable as described, there are ways to make it significantly more efficient and powerful. In the section on Detection and Early Pruning, we discuss the use of an *operator graph* (introduced in (16)) for:

1. Quick detection of recursion during planning

2. Early pruning of plans with suspended open conditions

After we discuss operator graphs, we consider a much broader class of recursion that we call *instance recursion.* In this type of recursion, the predicate repeats, but the variable bindings may be different each time. We develop a suspension and pruning technique that controls this type of recursion.

For purposes of this paper we will consider a simple SNLP style causal-link planner (10, 2). The expression $S_1 \xrightarrow{C} S_2$ will be used to denote the causal link from the producer step S_1 to the consumer step S_2 with the condition C. We say that there is a *causal-link path* from a step S_1 to a step S_n if the plan contains a set of steps $S_1,...,S_n$ and causal-links $S_i \xrightarrow{C_i} S_{i+1}$ for i=1,...,n-1. Similarly, we say that there is a causal-link path from an open condition C to a step S_n if C is an open condition of S_1 and there is a causal-link path from S_1 to S_n. We say that a step S_1 (open condition C) is a *causal-predecessor* of a step S_n if there is a causal-link path from S_1 (C) to S_n.

Due to space limitations we provide only informal arguments for the theorems in this paper.

Exact Recursion

Consider the partial plan shown in Figure 1. In this plan, the open condition O:C appears as a subgoal of the condition C in the causal link $S_j \xrightarrow{C} S_i$. Intuitively it seems that we should be able to discard this partial plan. If there is some way of achieving C then it could presumably be used to achieve C directly, without the intervening steps from S_k to S_j. Unfortunately, this argument is not generally sound.

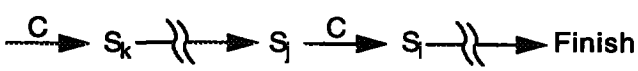

Figure 1: Simple example of exact recursion.

There are two circumstances where the steps from S_k to S_j are still needed:

1. Some step between S_k and S_j (inclusive) might be used for another purpose, i.e., a causal link might be added from one of these steps to another open condition in the plan. This situation is illustrated in Figure 2.
2. Some other step that clobbers C might be forced to come between S_k and S_j. In other words, a threat to C might arise that cannot be resolved by promotion after S_i or by demotion before the action used to achieve the open condition C. This situation is illustrated in Figure 3.

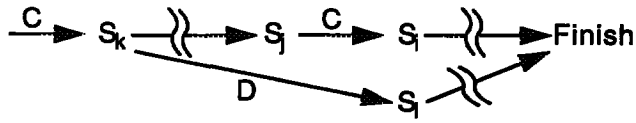

Figure 2: Link into a step between repeated open conditions.

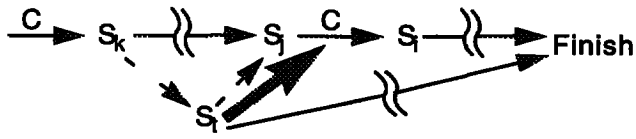

Figure 3: Threat to C that can only be resolved by placing S_t between S_k and S_j. The dashed edges indicate ordering constraints, and the thick grey edge indicates the (resolved) threat.

This second condition is a bit tricky. If the threat to C were resolved by forcing it to come after step S_i then the recursion steps S_k through S_j would still be unnecessary. Likewise, if the threat could be resolved by forcing it to come before whatever step is used to establish the open condition C the recursion steps through S_j would still be unnecessary. Thus, it is only when the threatening step S_t is forced to come between S_k and S_j that this partial plan must be saved.

Our approach for controlling such recursion has two parts. First, we use an ordering strategy that prevents expansion of recursive open conditions (like C) unless/until 1) a link is added from a step in the loop to some outside open condition (as in Figure 2) or 2) a threat appears to the loop condition that can only be resolved by forcing the threatening step to come within the loop (as in Figure 3). Second, we specify conditions where such recursive plans can actually be pruned without sacrificing completeness. To make all of this precise, we first need some definitions:

Definition 1: An open condition C is said to be *exactly recursive* if:

1. For every causal link path from C to the goal, there is a causal link $S_j \xrightarrow{C_i} S_i$ in the path.
2. None of the S_i are ordered with respect to each other.

We refer to the causal links $S_j \xrightarrow{C_i} S_i$ as the *root-links* of the recursion. ■

In the example in Figure 1, $S_j \xrightarrow{C} S_i$ is the only root link for the recursive open condition C.

Definition 2: Let C be a recursive open condition, and let L be its root links. We say that an open condition P is a *loop predecessor* for C if every causal link path from P to a goal contains a link $l \in L$. ■

Note that the recursive open condition C will always be a loop predecessor, but there may be many more other (non-recursive) open conditions generated by steps in the loop, and these conditions will also be loop predecessors.

In the example in Figure 1, the open condition C is a loop predecessor, but any other open conditions resulting from steps from S_k through S_j (inclusive) would also be loop predecessors.

Definition 3: Let C be a recursive open condition, and let L be the root links of the recursion. A step T is said to be a *loop threat* for C if:

1. T plausibly threatens C (T has an effect $\neg C$).
2. T necessarily precedes some $l \in L$. ■

In Figure 3, the step S_t is a loop threat for C.

We can now give a more precise statement of our strategy for preventing exact recursion.

Exact Recursion Strategy: Let C be a recursive open condition and suppose there is no loop threat for C. The condition C is *suspended* (the planner is prevented from working on C) until either:

1. A causal link is added from a step between C and one of its root links to an open condition outside the loop.
2. A loop threat for C appears.

In addition, all other loop predecessors P for C are suspended until either one of the above conditions holds or:

3. A causal link is added from a step between a suspended loop predecessor P and a root link to an open condition outside the loop. ■

Conditions 1 and 2 correspond to the cases illustrated earlier in Figures 2 and 3. Condition 3 is necessary because a link from a predecessor step to an outside open condition can (later) lead to a loop threat.

To see how this works, consider the partial plan in Figure 1. The open condition C would be suspended, along with any other loop predecessors from the steps $S_k,\ldots,S_j$. The planner could continue to work on any other open condition that is not suspended. In doing so, suppose that the link $S_k \xrightarrow{D} S_l$ is established, as shown in Figure 2. In this case the open condition C, and any other suspended loop predecessors would be re-enabled.

Theorem 1: A partial plan can be pruned (without compromising completeness) if all open conditions in the plan are suspended, and all threats have been resolved. ■

Sketch of Proof: Consider a plan in which the remaining open conditions $C_1,\ldots,C_n$ are suspended. Suppose that there is some totally-ordered completion of this plan. We argue that this completion will be non-minimal (see (9)) because at least one of the loops can be removed.

Let $S_1,\ldots,S_n$ be the steps used in the completion to establish the suspended condition $C_1,\ldots,C_n$. Let S_i be the step in $S_1,\ldots,S_n$ that occurs latest in the (totally ordered) completion of the plan. Since S_i is used to establish C_i, all of the steps in the causal-link path from C_i to the goal must follow S_i in the total order. This means that all of these steps must follow $S_1,\ldots,S_n$, since S_i is the latest. As a result, none of the $S_1,\ldots,S_n$ can threaten any of the $C_1,\ldots,C_n$, and there cannot be a causal link path from any step after C_i to any of the $S_1,\ldots,S_n$. As a result, the steps between C_i and its root links can be removed from the plan, so it is not minimal. ■

For the example in Figure 1, the plan can be pruned by this theorem if no links (as in Figure 2) are made to steps not in the loop, and no loop threats (as in Figure 3) occur.

Detection and Early Pruning

Although the method described in the previous section can be used as is, there is considerable overhead involved in recognizing recursion. For each new step added to the plan, the planner would need to search each causal link path to the goal, and look to see if any of the newly added open conditions are present on the path. While this is inexpensive for shallow plans, the cost grows significantly with plan depth and complexity. In (16), we introduced a structure called an *operator graph* that captures the interaction between operators relevant to a goal and set of initial conditions. In that paper, operator graphs were used for analyzing possible conflicts between operators relevant to a planning problem. However, the operator graph can be used for many other purposes, including recognizing:

1. Open conditions that have the potential to be recursive
2. Open conditions in a partial plan that will never result in links to steps in a suspended loop
3. Open conditions that will never lead to threats to a suspended condition

To explain these further, we first need to review the basics of operator graphs.

An operator graph is a directed bipartite graph containing one operator node for each operator relevant to a goal and one precondition node for each precondition of each such operator. The graph is constructed recursively by working backwards from the goal, according to the following rules:

1. There is an operator node for the Finish operator.
2. If an operator node is in the graph, there is a precondition node in the graph for each precondition of the operator, and a directed edge from the precondition node to the operator node.
3. If a precondition node is in the graph, there is an operator node in the graph for every operator with an effect that unifies with the precondition, and there is a directed edge from the operator node to the precondition node.

Figure 4 shows an operator graph for a simple travel problem where the goal is to be someplace with gas, and there is one operator:

Drive(x,y)

Preconditions:	At(x), Road(x,y)
Effects	At(y), ¬At(x)

(Lower case letters are used for variables. Constants and relations are capitalized.)

There are two arcs into the goal condition At(y) indicating that there are two possible ways of achieving it: one by linking to the initial conditions and the other by using the Drive operator. Similarly, there are two arcs into the At precondition of Drive. It too can potentially be satisfied by binding to the initial conditions, or by using another Drive operation. The other precondition of Drive, Road(x,y), can only be satisfied by the initial conditions, but there are several possible ways of doing so, indicated by the bundle of arcs from Start to Road(x,y). Likewise, the other goal condition Gas(y) can only be satisfied by the initial conditions, but again there are several possible ways of doing so.

Detecting Recursion

Our first observation is that loops within the operator graph indicate possible recursion during planning. More precisely:

Theorem 2: Let C be an open condition, and let P be its corresponding precondition node within the operator graph. (That is, the corresponding precondition node for the operator used in the step that gave rise to C.) If C is a recursive open condition, P must be part of a strongly-connected component (SCC)[1] within the operator graph. (The converse is not always true.) ■

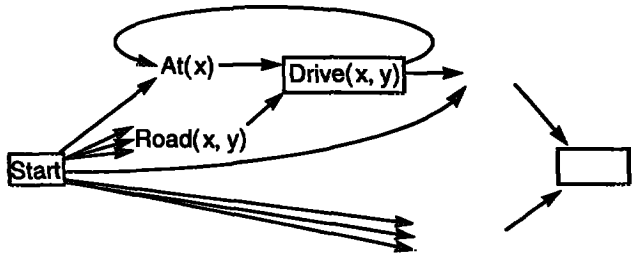

Figure 4: Operator graph for a simple travel problem.

The impact of this theorem is that the planner does not have to examine every open condition for possible recursion, only those that correspond to precondition nodes in the operator graph contained in SCCs.

In the graph above, there is one SCC that contains the Drive operator and its precondition At(x). As a result, the only open conditions that need to be checked for possible recursion are At open conditions that result from adding a Drive step to the plan.

SCCs can also be used to limit the amount of search involved in deciding whether a candidate open condition is recursive.

Theorem 3: Suppose C is an open condition that is a possible candidate for recursion, and suppose that the corresponding precondition P in the operator graph is in a SCC K. Let S be a step along a causal link path from C to the goal. If the operator node in the operator graph corresponding to S is not in the SCC K, no causal link between S and the goal can be a root of the recursion. ■

The import of Theorem 3 is that the planner can stop looking along the causal link path from an open condition C to the goal as soon as it finds a step not in the SCC. (For the example in Figure 4, this theorem has no impact, because the first step that is outside the SCC is Finish.)

Detecting Potential Links and Threats

It would be nice if we could tell which open conditions in a plan have the potential to add a link from a step in a suspended loop. Likewise, it would be nice if we could tell

[1] A strongly connected component is a maximal set of nodes such that there is a directed path from any node in the set to any other node in the set. Any loop in a directed graph (like the one in Figure 4) will be part of a SCC. SCCs and algorithms for finding them are described in algorithms texts such as (4).

which open conditions could potentially lead to loop threats. We can do this by looking at the nodes preceding the corresponding precondition node in the operator graph.

Definition 4: An operator R is said to be *relevant* to an open condition C if there is a directed path from R to the precondition node corresponding to C in the operator graph. ■

Given a suspension, the set of relevant operators for the remaining open conditions in a plan can help us decide whether the plan must be kept or can be discarded.

Theorem 4: Let P be a partial plan with a suspended open condition C and let L be the set of root links for the recursion. Let U be the remaining open conditions (including any other suspended open conditions), and let R be the set of operators relevant to the conditions in U. The partial plan P can be pruned (without compromising completeness) if all of the following conditions hold:

1. No operator in R appears in the SCC of C.
2. No operator in R threatens any root link $l \in L$.
3. P contains no unresolved threats to any root link $l \in L$. ■

The first of these conditions means that no other operator can link into the suspended loop. The second and third conditions mean that no threats can arise that would force re-enablement of the suspended condition. Every possible completion of this plan will therefore satisfy the conditions of Theorem 1, so the plan can be pruned.

As an example of the application of his theorem, consider the partial plan shown in Figure 5. We've added the goal condition Rested and the operator Sleep to the problem description. Sleep has no preconditions and can be used to achieve Rested. Currently, the open condition At(B) is suspended because it exactly matches the condition in the causal link Drive(A, B) $\xrightarrow{\text{At(B)}}$ Finish. The relevant operators for Gas(B) and Rested are Start and Sleep. Neither of these appear in the SCC for the open condition At(B). As a result, this plan can be pruned.

Instance Recursion

In the previous sections we considered only exact recursion. Thus, an open condition P(x,A) would be recursive if all of its paths to the goal contained a causal link with clause P(x,A), but not if the links only contained clauses P(y,A), P(x,B), or P(x,y). In other words, for exact recursion, the variables and constants in the recursive open condition must be identical to the variables and constants in the root links. In the example of Figure 5, the condition At(A) is not recursive, but the open condition At(B) is.

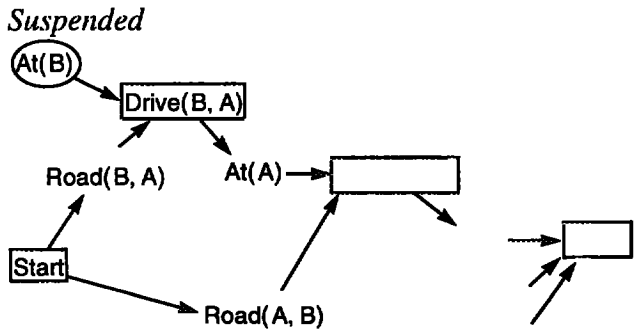

Figure 5: Partial plan with a suspended recursive open condition.

While the results of the previous sections were sufficient to prune the plan in Figure 5, in general, they are not sufficient to stop all recursion for planning problems in which the operators contain variables. Consider what would happen in our Drive example if the planner did not choose a good order for open condition expansion. The planner could generate an infinite sequence of partial plans like the one shown in Figure 6. While this open condition ordering may

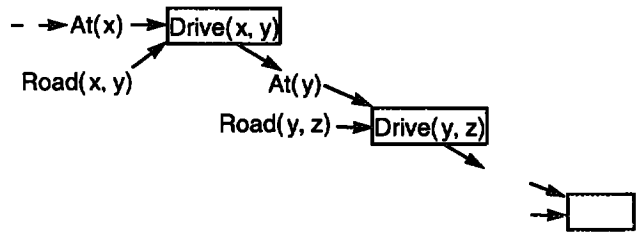

Figure 6: Recursion resulting from unbound variables.

look silly, it is exactly the one that would be chosen by such "smart" (and generally effective) strategies as the Least-commitment (LC) strategy used in (13) and Joslin's LCFR strategy (8). These strategies choose to expand the open condition with the fewest options. They get into trouble because there are many roads and many gas stations, but only two ways of achieving a condition like At(x): adding a Drive step or binding to the single At fact in the initial conditions. If there is no solution to the above problem (because you start on an island and all gas stations are on the mainland), planners with the LC or LCFR strategy would never terminate.

One obvious possibility is to avoid working on conditions such as At(x) and At(y) until the variables x and y are bound. As it turns out, this isn't quite right.[2] Instead, we want to avoid working on At(x) and At(y) until the variable z is bound in At(z). More precisely, we want to avoid At(x) and At(y) as long as they remain an *instance* of the root link condition At(z).

The reason for this strategy is that until z is bound, the planner doesn't know what it is trying to achieve, which, in a recursive situation like this, can lead to lots of irrelevant plans. Another way of thinking about it is this: suppose there is some plan for achieving At(x), for some binding of x. As long as At(x) remains an instance of At(z), that same plan would be directly applicable to At(z). Until the planner has reason to believe that z=B won't work, there's no reason this shorter plan shouldn't be preferred. In the extreme case where z is never bound (gas is available everywhere), any plan involving recursion would be non-minimal.

As we did for exact recursion, we now provide a formal definition of instance recursion and give a suspension strategy.

Definition 5: An open condition C' is said to be *instance recursive* if there is a set of variable bindings V such that:

1. for every causal link path from C' to the goal, there is a causal link $S_j \xrightarrow{C} S_i$ in the path and $C' = C|_V$ (where $C|_V$ refers to C instantiated with the variable bindings V).[3]
2. None of the S_i are ordered with respect to each other.

As before, we refer to the $S_j \xrightarrow{C} S_i$ as the root links of the recursion. ■

In the above example At(x) and At(y) are both instance recursions of the root link Drive(y, z) $\xrightarrow{At(z)}$ Finish, with z bound to x and y respectively.

Instance Recursion Strategy: Let C' be an instance recursive open condition and suppose there is no loop threat for C'. The condition C' is suspended until either:

1. A causal link is added from a step between C' and one of its root links to an open condition outside the loop.
2. A loop threat for C' appears.

Variables bindings are added to the plan so that C' is no longer an instance of one or more of its root links. ■

The instance recursion strategy is weaker than the exact recursion strategy. For exact recursion we suspended all loop predecessors of the recursion but here we suspend only the recursive condition itself. For instance recursion we cannot suspend other loop predecessors because they may bind variables in the root link.

We can now generalize Theorem 1 to the cases involving instance recursion.

Theorem 5: A partial plan can be pruned if all remaining open conditions are suspended (either because of exact or instance recursion) and all threats have been resolved. ■

As before, the argument is that every possible completion of this partial plan will be non-minimal; at least one of the loops can always be removed.

As before, we would like to be able to prune such plans earlier. In the last section, we showed how the operator graph can help recognize when outstanding open conditions can no longer link into or threaten an exact recursion. We can do the same here. However, we also need to make sure that the outstanding open conditions cannot bind any of the variables in the root links. We generalize Theorem 4 as follows.

Theorem 6: Let P be a partial plan with a suspended instance recursive open condition C' and let L be the set of root links for the recursion. Let U be the remaining open conditions (including any other suspended open conditions), and let R be the set of operators relevant to the conditions in U. The partial plan P can be pruned (without compromising completeness) if all of the following conditions hold:

1. No operator in R appears in the SCC of C'.
2. No operator in R threatens the condition of any root link $l \in L$.
3. P contains no unresolved threats to any root link $l \in L$.
4. No open condition in U contains a variable in any of the root link conditions $l \in L$. ■

Although the suspension strategy and pruning theorems for instance recursion look very similar to those for exact recursion, there is a subtle difference. The final condition in Theorem 6 makes it much weaker than Theorem 4, so fewer instance recursive plans can actually be pruned. It is worth noting that instance recursions sometimes turn into exact recursions (e.g. Figure 5) and are then subject to the stronger Theorem 4.

[2] One reason is that all open conditions containing the variables of interest may be recursive, so the variables will never get bound without expanding a recursive open condition. The simplest example of this is a Drive operator that has no Road precondition (the vehicle might be a Humvee or tank). In this case, there would be no other open condition to bind the x in At(x).

[3] This definition is asymmetric on purpose. It is not enough to require only that C and C' unify. Suppose C' is At(x) and C is At(A). In this case we do not want to suspend or prune the condition At(x). The reason is that At(x) is easier to establish than At(A) because any location for x will work (provided it satisfies other intermediate preconditions). If C and C' unify, but C' is not an instance of C, exact or instance recursion will usually result after one more time around the loop.

Experiments

The exact and instance suspension and pruning strategies were implemented in our ONLP planner (Operator Graph-based partial order planner). We tested the recursion algorithms on a number of problems drawn from several domains including: Russell's Tire World domain(13), Bulldozer (13), towers of Hanoi, Monkey and Bananas, Weld's Refrigerator domain, a noncombatant evacuation operation (NEO) domain, and blocks world. All of these domains were adapted from the domain library in UCPOP (12).

ONLP implements a number of strategies for operator graph and plan analysis. For our tests, we used the following settings:

- Open Condition Ordering: Least Commitment (13) – select the open condition that can be satisfied in the fewest ways.
- Threat Delay Strategy: DMIN (17) – a least commitment threat resolution strategy similar to DUNF (13).
- Variable Analysis: substitute the binding for each variable in the operator schemas if there is only one possible binding for that variable.
- Cost Function:

  # Steps + # Unsuspended Open Conds
  + K (# Suspended Open Conds)

  The constant K penalizes partial plans that have suspended open conditions. The values used in our tests were K = 1 (no penalty), K = 2 and K = 4.
- Search Algorithm: Stable best-first--A best first search algorithm that always breaks ties in the order that the plans were added to the search queue (LIFO).

The results of these tests are shown below. In these charts, N_{normal} and $N_{suspend}$ denote the number of plans generated by the planner with and without recursion suspension and pruning. (These counts do *not* include the number of plans explored during threat resolution, only the plans generated using add-link or add-step. See (13) and (17) for explanation.)

Figure 7 plots the reduction in search space size realized by using recursion suspension and pruning ($N_{normal}/N_{suspend}$) against the size of the search space when no suspension and pruning is used (N_{normal}). Two data series are plotted in this chart: one for K=1 and one for K=4.

Almost all of the points in Figure 7 are above the X-Axis, which indicates that there was some degree of search space reduction achieved for most problems. The improvement

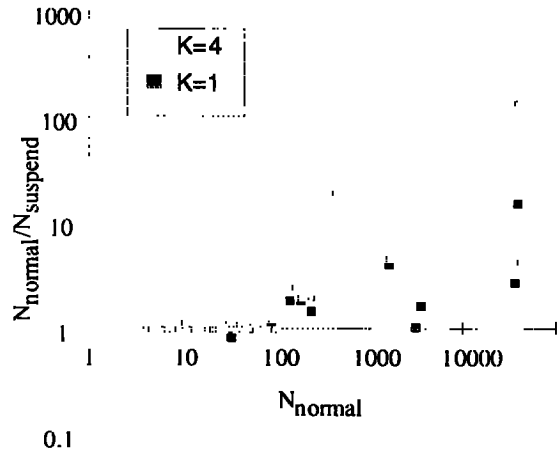

Figure 7: Search space improvement when using recursion suspension and pruning.

ranges from none to as much as 139 times on the tire changing problem. Generally, the improvement is greater for the larger domains and harder problems. One reason is that these plans tend to be longer, and the odds of a plan bei ng non-minimal increases with the length of the plan (more opportunities for recursion). Note that better performance is obtained using the penalty K=4 for suspended open conditions. This reflects the high probability that the best plan does not contain recursion.

Figure 8 plots the improvement in search time using recursion suspension and pruning ($T_{normal}/T_{suspend}$) against the time (in seconds) required for solving each problem with no suspension and pruning (T_{normal}).

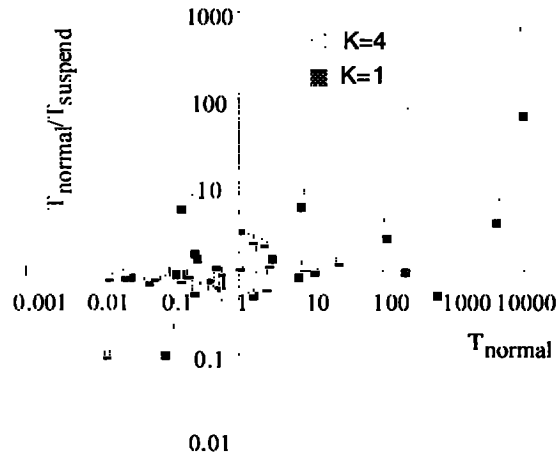

Figure 8: Time improvement when using recursion suspension and pruning

As with search space size, all points are either close to, or above the X-axis. This indicates that the overhead of recursion checking and suspension is sufficiently small that it has little negative impact on planner performance, even when no search space reduction is realized by the strategy. The overhead of building the operator graph, and finding strongly connected components was negligible. For small problems it was typically less than 0.1 seconds, and for all problems it was less than 1 second.

It is interesting to note that a few of the time improvements are very much larger than the search space size improvements indicated in Figure 7. This would seem to indicate that the normal planner is spending much of its time dealing with lengthy and complex plans avoided by the suspension strategy.

Discussion

Non-causal-link Planning

Kambhampati (9) has shown that it is much easier to detect and prune non-minimal plans for non-causal-link (NCL) planners such as Tweak (3). If a loop is necessary for solving a problem, it will get reintroduced by the white-knight mechanism. As a result, many of our complex conditions for suspension and pruning do not appear to be necessary for NCL planning. Baioletti, et. al. (1) have developed a set of conditions that prune recursive open conditions for a Tweak-based planner.

Minimality

Although our methods are quite powerful, they still allow the generation of non-minimal plans in some circumstances. For example, consider the partial plan in Figure 2. According to our strategy, the open condition C would be re-enabled when the causal link $S_k \xrightarrow{D} S_l$ is added to the plan. However, suppose that the step S_k does not clobber the condition C. In this case the steps in the causal-link chain after S_k up to S_j can still be removed from the plan. Thus, our strategy, as stated, allows a non-minimal plan to be generated in this case. We have found other, more complex examples of non-minimality.

It is possible to strengthen several of the theorems in this paper to eliminate some or all of these non-minimal plans. However, the conditions become more difficult to understand, and more difficult to implement.

Conclusions

The recursion suspension and pruning strategy significantly reduces search time and space for many problems and, in particular, they reduce search time significantly for large recursive problems. Using the operator graph to help detect recursion keeps the overhead associated with this technique to a minimum. Finally, our test results indicate that penalizing plans with large numbers of suspended open conditions can further improve performance.

Acknowledgments

We first implemented the suspension method in early 1993. Dan Weld encouraged us to finally write it up. Dan, Mike Williamson and Oren Etzioni provided detailed comments on an earlier draft. Thanks to Tony Barrett for collecting and making many of the test domains available. This work was supported by DARPA contract F30602-91-C-0031 and by Rockwell.

References

1. Baioletti, M., Marcugini, S., Milani, A. and Rivoira, S., A nonlinear planner with loop avoidance. *Fourth International Conference on Industrial & Engineering Application of Artificial Intelligence & Expert Systems*, pages 650–658, 1991.
2. Barrett, A. and Weld, D., Partial order planning: evaluating possible efficiency gains. *Artificial Intelligence*, vol 67(1), pages 71–112, 1994.
3. Chapman, D., Planning for conjunctive goals. *Artificial Intelligence*, vol 32(3), pages 333–377, 1987.
4. Cormen, T., Leiserson, C., and Rivest, R., *Introduction to Algorithms*. McGraw Hill, 1991.
5. Feldman, R. and Morris, P., Admissible criteria for loop control in planning. In *Proc. AAAI-90*, pages 151–157, 1990.
6. Etzioni, O., Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, vol 62(2), pages 255–301, 1993.
7. Fikes, R., Hart, P., and Nilsson, N., Learning and executing generalized robot plans. *Artificial Intelligence*, vol 3(4), pages 251–288, 1972.
8. Joslin, D. and Pollack, M., Least cost flaw repair: a plan refinement strategy for partial-order planning. In *Proc. AAAI-94*, pages 1004–1009, 1994.

9. Kambhampati, S., Admissable pruning strategies based on plan minimality for plan-space planning. In *Proc. IJCAI-95*, pages 1627–1633, 1995.

10. McAllester, D. and Rosenblitt, D., Systematic nonlinear planning. In *Proc. AAAI-91*, pages 634–639, Anaheim, CA, 1991.

11. Minton, S., Knoblock, C., Kuokka, D., Gil, Y., Joseph, R., and Carbonell, J., Prodigy 2.0: the manual and tutorial, Tech. Report CMU-CS-89-146, Carnegie Mellon University. Pittsburgh, PA, 1989.

12. Penberthy, J. and Weld, D., UCPOP: A sound, complete, partial order planner for ADL. In *Proc. KR-92*, pages 189–197. 1992.

13. Peot, M. and Smith, D., Threat removal strategies for partial-order planning. In *Proc. AAAI-93*, pages 492–499, 1993.

14. Rich, E. and Knight, K., *Artificial Intelligence*. Second edition, McGraw Hill, 1991.

15. Smith, D., Genesereth, M., and Ginsberg, M., Controlling recursive inference. *Artificial Intelligence*, vol 30(3), pages 343–389, 1986.

16. Smith, D. and Peot, M., Postponing threats in partial order planning. In *Proc. AAAI-93*, pages 500–506, 1993.

17. Smith, D. and Peot, M., *A Note on the DMIN strategy*. Technical Memo, Rockwell Palo Alto Laboratory, 1993. Available on-line from http://www.rpal.rockwell.com:80/~de2smith/publications.html

18. Tate, A., Generating project networks. In *Proc. IJCAI-77*, pages 888-893, 1977.