Making Forward Chaining Relevant

Fahiem Bacchus
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario
Canada, N2L 3G1
fbacchus@logos.uwaterloo.ca

Yee Whye Teh
Dept. of Computer Science
University of Toronto
Toronto, Ontario
Canada, M5S 1A4
ywteh@logos.uwaterloo.ca

Abstract

Planning by forward chaining through the world space has long been dismissed as being "obviously" infeasible. Nevertheless, this approach to planning has many advantages. Most importantly forward chaining planners maintain complete descriptions of the intermediate states that arise during the course of the plan's execution. These states can be utilized to provide highly effective search control. Another advantage is that such planners can support richer planning representations that can model, e.g., resources and resource consumption. Forward chaining planners are still plagued however by their traditional weaknesses: a lack of goal direction, and the fact that they search totally ordered action sequences. In this paper we address the issue of goal direction. We present two algorithms that provide a forward chaining planner with more information about the goal, and allow it to avoid certain types of irrelevant state information and actions.

Introduction

In this paper we present two ways of improving the efficiency of a forward chaining planner. Such planners search in the space of worlds generated by applying all possible (totally ordered) action sequences to the initial state.[2] The two mechanisms correspond to ways of making these planners more goal directed by allowing them to ignore actions that are irrelevant to the goal. The first method utilizes a static analysis of the domain actions. It runs in polynomial time and is performed on individual planning problems prior to plan search. The analysis allows the planner to ignore some of the domain actions during planning while still retaining completeness. The second method is a dynamic control mechanism that operates during search. It prunes from the search space action sequences containing actions made irrelevant by other actions in the sequence. This has the effect of pruning from the search space certain redundant paths. The two mechanisms can be used together to achieve even greater improvements.

At this point the reader may wonder why we are interested in forward chaining planners, since this approach to planning has long been dismissed in the planning community in favor of more sophisticated approaches. In our opinion, however, of all the approaches to AI planning that have been developed, including recent innovations like Graphplan [BF97] and Satplan [KS96], forward chaining has the most promise. This is a controversial opinion, and although we hope to accumulate more evidence to support it, we know that it is an opinion that the reader might not share. In this paper we can only offer a brief defense of forward chaining and why further development of this approach to planning is worthwhile.

Forward chaining planners have two particularly useful properties. First, they maintain complete information about the intermediate states generated by a potential plan. This information can be utilized to provide highly effective search control, both domain independent heuristic control [McD96], and even more effective domain dependent control. For example, with domain specific information in the blocks world domain, the TLPLAN system [Bac95] developed in [BK96b] can generate solutions to problems involving 100 blocks in under 10 seconds, where as the fastest domain independent planners, Graphplan and Satplan, both take over 1000 seconds to solve problems involving 11 blocks. Furthermore, TLPLAN can solve problems in a range of other domains orders of magnitude faster than any other planning system, and the intermediate states can also be used to ensure that the plan satisfies a range of temporally extended conditions, of which maintenance and safety conditions are just simple instances [BK96a]. The second advantage of forward chaining planners is they can support rich planning languages. The TLPLAN system for example, supports the full ADL language, including functions and numeric calculations. Numbers and functions are essential for modeling many features of real planning domains, particularly resources and resource consumption.

Nevertheless, forward chaining planners have a number of well known deficiencies that are at the root of their dismissal by the planning community. One of the most serious deficiencies is that such planners are not goal directed, and thus can end up pursuing action sequences irrelevant to the

[1]Copyright ©1998, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

[2]There is a range of terminology used to refer to such planners. Weld [Wel94] calls this kind of planner a progressive world-state planner, McDermott [McD96] calls it searching in the space of plan prefixes, and Russell and Norvig [RN95] it call a progressive totally ordered situation space planner. The term "forward chaining" is more succinct and it also has a long history.

current goal. This is an especially serious problem when we consider scaling up such planners. Useful intelligent agents will probably have to deal with a range of different problems, and will have at their disposal a range of different actions. Only a small subset of these actions are likely to be relevant to any particular task. If a forward chaining planner has to explore all possible actions irrespective of the goal, then it is ultimately doomed to failure. The algorithms we develop here address this problem, and doing so we demonstrate that this particular argument against forward chaining planners can be countered.

Of course there are other arguments against forward chaining planners that still remain. Most importantly among these is the issue of totally vs. partially ordered action sequences. The fact that forward chaining planners explore totally ordered action sequences remains an area of difficulty for such planners. We do not address this issue here, but we are currently exploring some mechanisms for dealing with this problem also.

The overall aim of this and other work we are pursuing is to try to improve the baseline performance of forward chaining planners. Search control still remains an essential component in making such planners perform effectively,[3] but effective control information, especially domain specific information, is often quite expensive to acquire. By improving the baseline performance we hope to require less control information and to make the information we do have more effective.

In the sequel we present the two methods we have developed for adding goal direction to forward chaining planners and give some empirical results showing their effects.

Static Relevance

The first algorithm we describe is one that operates prior to searching for a plan. Hence, we call it "static" relevance. First an example. Say that we have the following actions in the domain:

1. $pre(a_1) = \{P\}$, $effects(a_1) = \{Q, S\}$,
2. $pre(a_2) = \{Q\}$, $effects(a_2) = \{R\}$, and
3. $pre(a_3) = \{P\}$, $effects(a_3) = \{T\}$,

where the actions are specified using the STRIPS representation with pre being the set of preconditions, $effects$ being the set of effects which can be positive literals (adds) or negative literals (deletes).

In the initial state $I = \{P\}$, both actions a_1 and a_3 can be executed. If the goal $G = \{R\}$, then it is easy to see that there is no reason to execute action a_3: it does not yield a goal literal nor does it yield a precondition that can ultimately be used to produce a goal literal. Action a_1 on the other hand produces Q which can be used by action a_2 to produce a goal literal. However, the other literal it produces, S does not facilitate the execution of any relevant actions. In sum, for this initial state and goal the action a_3 is irrelevant as is the literal S produced by action a_1.

[3]In fact, effective search control is essential for the success of any planning architecture.

Inputs: The initial state I and the goal state G, both specified as a collection of ground literals, and a set of ground action instances $Acts$ specified using the STRIPS representation.
Output: A set of literals, $RelvLits$, and actions, $RelvActs$, that are potentially relevant to the planning problem of transforming I to G.

Procedure $\mathbf{Static}(I, G, Acts)$

1. $ReacLits := I$; $RelvLits := G$;
   $RelvActs := ReacActs := \emptyset$;
2. $ReacActs := \{a : pre(a) \subseteq ReacLits\}$;
3. **if** $ReacActs$ was changed in step 2 **then**: $ReacLits := I \cup \bigcup_{a \in ReacActs} \{\ell : \ell \in effects(a)\}$;
4. **if** $ReacLits$ was changed in step 3 **then**: **goto** step 2;
5. **if** there exists $\ell \in G$ such that $\ell \notin ReacLits$ **then**: **return**(Failure);
6. $RelvActs := \{a : a \in ReacActs \wedge (effects(a) \cap RelvLits) \neq \emptyset\}$;
7. **if** $RelvActs$ was changed in step 6 **then**: $RelvLits := G \cup \bigcup_{a \in RelvActs} \{\ell : \ell \in pre(a)\}$;
8. **if** $RelvLits$ was changed in step 7 **then goto** step 6;
9. $\mathbf{return}(RelvLits, RelvActs)$;

Table 1: Static Relevance Algorithm

Our static relevance algorithm is designed to detect these kinds of irrelevance, and as we explain below it can be used to simplify a planning problem and to provide forward chaining with a degree of goal direction. The algorithm for computing the set of statically relevant actions and literals for a specific planning problem is given in Table 1. The algorithm takes as input a fully ground set of actions. This set can be computed from a set of parameterized operators by instantiating the operators in all possible ways with the constants contained in the initial state I.[4] First the algorithm performs a forward pass to detect the set of potentially reachable literals and actions. A literal is reachable if it is present in the initial state or if it is the effect of some reachable action. An action is reachable if all of the literals in its preconditions are reachable. Note that marking a literal or an action as reachable does not mean that it is actually reachable from the initial state. In particular, an action may have all of its preconditions marked as being reachable, but the conjunction of these preconditions might in fact not be reachable. The loop in steps 2–4 computes the reachable sets, $ReacLits$ and $ReacActs$.

Then the algorithm performs a backwards pass to detect the set of potentially relevant literals and actions. A literal is relevant if it is reachable and it appears in the goal or in the precondition of a relevant action. An action is relevant if it is reachable and it produces a relevant literal. The loop in step 6–8 computes the relevant sets, $RelvLits$ and $RelvActs$.

[4]We have given the algorithm in terms of ground actions, but our implementation in fact works directly with the parameterized operators. It generates the ground actions only on an as needed basis.

1. Remove all irrelevant literals from I to form a new initial state $I' = I \cap RelvLits$.
2. Remove all irrelevant actions from $Acts$ to form a new set of actions $Acts' = Acts \cap RelvActs$.
3. Modify every action $a \in Acts'$ by removing from $effects(a)$ all literals ℓ such that both ℓ and $\neg\ell$ are irrelevant (i.e., not in $RelvLits$).

Table 2: Reduced planning Space Algorithm

There are a number of points to be made about the algorithm. First, the algorithm operates on literals, i.e., positive or negative atomic facts. Hence if the actions have negative literals as preconditions the algorithm continues to function properly. The only caveat is that under the standard closed world assumption used by most planners, I contains many implicit negative facts. The algorithm does not place these negative facts in the set $ReacLits$ even though they are in fact reachable literals (the algorithm does place negative facts produced by actions into $ReacLits$). Instead, whenever we test a negative literal for membership in $ReacLits$ (in steps 2, 6, and 7) the implementation also tests to see if the literal is implicitly in I. The end result is that the final set $RelvLits$ contains all relevant literals, both positive and negative.

Second, although we have specified the two loops as recomputing the sets from scratch, it is not difficult to see that these computations can be performed incrementally. Our implementation does the computation incrementally.

And finally, we have found that in practice it is more efficient to do an initial backwards pass from G marking all of the predicate and action *names* that are potentially relevant. In particular, this initial pass ignores the arguments to the actions and literals. Once the names have been marked we can restrict the forward pass to only consider literals and actions whose names have been marked as being potentially relevant.

Utilizing Static Relevance

Once we have the set of relevant literals and actions generated by the above algorithm, we can use them to construct a smaller planning space. The smaller space can be viewed as being a quotient space where the states of the original planning space have been reduced into a smaller set of equivalent classes. Search for a plan can then be conducted in this smaller space.

In particular, the original planning space is specified by the initial state I, the goal G, and the set of actions $Acts$. The algorithm given in Table 2 constructs the reduced space $(I', Acts', G)$. The algorithm is specified as using and generating a set of ground actions. However, in our implementation we do not explicitly store this set. Instead, we use the original set of parameterized operators, and check the ground actions and effects generated at plan time to ensure that they are relevant.

The reduced planning space preserves completeness.

Theorem 1 *There exists a sequence of actions from $Acts$ that can solve the planning problem of transforming I to a state satisfying G* iff *there exists a sequence of actions from $Acts'$ that can solve the planning problem of transforming I' to a state satisfying G.*

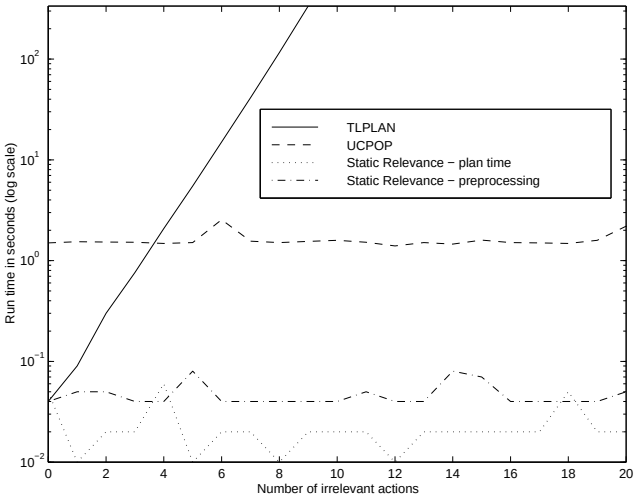

Figure 1: Decreased branching factor using static relevance

And it is not that hard to compute.

Theorem 2 *The complexity of computing the reduced planning space is $O(o^k)$, where o is the number of domain objects mentioned in the initial state I, and k is a constant equal to the maximum arity of the domain operators and predicates. This complexity is polynomial in the size of I. More precisely, let n be the number of distinct operators (the parameterized operators from which the ground actions are generated), let m be the number of domain predicates. Both of these are constants. Then the complexity is bounded above by $(n + m)o^k$.*

The reduced planning space offers two advantages for forward chaining planners. First, there is the obvious advantage that by removing irrelevant actions we reduce the branching factor of the search space they explore. The second advantage is a bit more subtle. Any search engine can profitably employ cycle checking. Breadth-first based search offers the most opportunity for cycle checking, however, even when utilizing depth-first based search the search engine can still check for state cycles along the current path being explored. By removing irrelevant effects from the actions and the initial state in the reduced planning space, states become equivalent that would not have been equivalent in the original space, and cycle checking can play a greater role.

Empirical Results

To test static relevance we conducted three tests. The first two are designed to show the claimed properties of static relevance: that irrelevant actions can be ignored which reduces the branching factor, and that eliminating irrelevant literals facilitates greater cycle-detection.

The first test was to run a simple blocks world problem with 4 blocks in the initial world state, and 4 blocks

in the goal state. The test consisted of adding n new actions $a_1, \ldots, a_n$ where a_i is the action $pre(a_i) = \{P_i\}$, and $effects(a_i) = \{G_i\}$. We set the initial state to contain all of the P_i so that all of these extra actions are executable, but the goal does not contain any G_i so in fact they are all irrelevant. A standard forward-chaining planner (we use the TLPLAN system for all of our tests [Bac95]), will have an increasingly higher branching factor as n increases, but static relevance will detect that these actions are irrelevant and hence will not be affected by their presence. Figure 1 shows the results. We ran TLPLAN using breadth first search. In the test we also ran UCPOP. UCPOP uses goal regression so it also is unaffected by these extra actions. The results show that with static relevance TLPLAN, like UCPOP, is able to achieve a runtime unaffected by n, without it its complexity climbs rapidly.

The next test is designed to show that static relevance facilitates cycle detection. Again we use the block world domain, but this time we make multiple copies of each of the operators. In particular, we make three copies of each of the operators *pickup*, *putdown*, *stack*, and *unstack*. For the i-th copy ($i = 1, 2, 3$), of *pickup* and *unstack* we add the effects *extra1*(i) and $\neg$*extra2*(i), and for *putdown* and *stack* we add the effects *extra2*(i) and $\neg$*extra1*(i).

Neither of the predicates *extra1* or *extra2* appear in the goal, and thus they are irrelevant. Without static relevance, TLPLAN is unable to utilize cycle-checking properly as even though a cycle might be present in the standard blocks world component of the world, the various changes to the extra literals *extra1* and *extra2* makes the world different (most of the time). When static relevance is used, however, we see that this extended domain reduces to the original blocks domain (with three copies of each operator).

Our test consisted of running 10 random 3 blocks problems (to convert a random initial state to a random goal state). Without the irrelevant literals TLPLAN took 0.061 seconds in total run time to execute the tests but when they were present its run time rose to 13.92 seconds. When static relevance is used this dropped back down to 1.98 seconds.[5]

The final test was designed to address the classical criticism of forward chaining planners, that they cannot scale up in the face of increasing numbers of available actions. In this test we used a blocks world problem containing 5 blocks that takes TLPLAN 1.4 seconds to solve using breadth-first search exploring 741 worlds.

Then we added in a number of additional (non-interacting) domains, running TLPLAN with the union of the domain actions and with an initial state that contained literals from the other domains (so that the additional actions were executable). The domains we added were standard test domains: the monkey and bananas, rocket, logistics, and tires domains, as well as an artificial domain containing 20 actions. After adding in only one extra domain TLPLAN was unable to solve the original problem after searching 5000 worlds. With static relevance however its runtime returned to the previous 1.4 seconds (approximately) and stayed there as we kept increasing the number of additional domains.

In all cases, the static relevance algorithm took approximately 0.03 seconds to execute: its run time remained constant as we increased the number of extra domains. This was due to the initial backwards pass (mentioned briefly in the text above) that marks action and predicate names prior to the forward reachability pass. In this test the initial backwards pass is able to eliminate all of the actions from the extra domains from any further processing. Also the planner's run time did not increase because the static relevance algorithm is able to remove the other domains' operators from the list of operators prior to planning: since no instance of any of these operators appears in the set of relevant actions, the planner does not need to consider them at all.[6]

Related Work

One way of understanding static relevance is to view it as a mechanism that allows a forward chaining planner to realize some of the benefits of partial order planners (in particular, planners based on the SNLP algorithm [MR91]). When such planners add actions to the partial plans they are searching, they only consider actions that achieve open conditions. Such conditions can only be generated in the plan by the goal or by the preconditions of an action added to achieve some prior open condition. It is not hard to see that static relevance is essentially computing the set of all the actions that could potentially achieve an open condition in some plan. The key difference is that it also takes into account the action bindings that are reachable from the initial state.

Gerevini and Schubert [GS96] have developed an algorithm for computing action bindings that are reachable from the initial state. They then use this information in an SNLP-style planner (UCPOP [PW92]), to help it avoid exploring actions that are relevant but not reachable. Their algorithm is closely related to ours. The key difference is that they use their algorithm to compute reachability information, where as we use ours to compute relevance information. Also they work with sets of bindings instead of fully ground literals and actions. Fully ground literals provide stronger information than sets of binding (i.e., the relevance sets computed are smaller), and we have found that there is hardly any computational time penalty over working with sets of bindings. We also deal with negative literals. We do not, however, deal with ADL actions where as Gerevini and Schubert's algorithm can handle the **when** clauses of ADL actions. It would not be difficult, however, to extend our algorithm to handle this case as well.

[5] The difference between this and the original run time arises from the extra time required to run the static relevance algorithm as well as some plan time overhead required by our implementation. This latter overhead could be reduced by a more sophisticated implementation.

[6] In some cases an operator may have some relevant instances and some irrelevant instances. Since the planner works with operators not actions (operator instances), it must consider the operator instances generated at plan time to determine whether or not it is relevant. This adds a constant time overhead at plan time. However, when no instance is relevant, we can eliminate that overhead by removing the operator entirely.

Our static relevance algorithm is also somewhat related to the planning graph construction of Graphplan: both can be viewed as being a type of reachability analysis. One difference is that we do not compute exclusivity sets. To do so, however, Graphplan's planning graph must grow with the length of the plan. An interesting question is whether or not some exclusivity information can be gained (which would allow further reduction of the relevance sets) without paying the plan length factor.

Nebel et al. [NDK97] point out that the size of Graphplan's planning graph can be a serious issue in its performance, and they have developed a collection of heuristics to detect irrelevant literals. Their heuristics are able to detect irrelevances beyond what our algorithm can detect. However, in doing so they lose completeness: their heuristics can remove relevant information thus rendering the planning problem unsolvable. Nevertheless, it may be possible to utilize some of their techniques to extend our approach.

Dynamic Relevance

Static relevance is a useful idea but it is relatively weak. It is particularly problematic when testing with the standard suite of planning test domains. Invariably these test domains are designed to generate plans for one particular purpose, and often all of the actions in the domain end up being statically relevant for the planning problem at hand (although not all of the effects do).

In this section we describe another algorithm that keeps track of relevance dynamically. Again we can motivate the idea with an example. Consider the standard blocks world with four operators *pickup*, *putdown*, *stack* and *unstack*. Say that in the initial world we have $\{$*ontable*(a), *ontable*(b), *ontable*(c), *ontable*$(d)\}$. Now consider the action sequence *pickup*(a), *stack*(a, b), *pickup*(c), *stack*(c, d), *unstack*(a, b), *putdown*(a). It is clear that there was never any need to move block a, and that the shorter action sequence *pickup*(c), *stack*(c, d) would have achieved the same final state. Unfortunately, unless we are doing blind breadth-first search there is no guarantee that the planner would have seen and remembered the shorter plan before it visited the longer sequence. Dynamic relevance is designed to prune such sequences from the search space. We thus avoid having to search all of the descendants of the pruned sequence as well, which means that such pruning has can potentially yield exponential savings during search.

Dynamic relevance is based on checking to see if an action sequence has a (not necessarily contiguous) subsequence of actions that are irrelevant. Consider an action sequence $\langle a_0, a_1, a_2, a_3, a_4, a_5\rangle$. We can split such a sequence into two subsequences, $R = \{a_0, a_3, a_4\}$ and $\overline{R} = \{a_1, a_2, a_5\}$. The question is "When are the actions in $\overline{R}$ irrelevant?" There are probably many different answers to this question, but an obvious one is the following:

Definition 3 A subsequence $\overline{R}$ of an action sequence is irrelevant when

1. R, the complement of $\overline{R}$ is an executable sequence (from the initial state), and
2. when R is executed it yields the same final state as the entire action sequence.

Intuitively, the definition says that R, the complement of $\overline{R}$, is equivalent to the entire sequence, and hence $\overline{R}$ is irrelevant. It should be clear that completeness is preserved when we prune actions sequences containing irrelevant subsequences from the forward chaining search space. It should also be clear that this definition covers the example given above. In particular, the subsequence *pickup*(a), *stack*(a, b), *unstack*(a, b), *putdown*(a), is irrelevant.

Our definition does not cover all intuitively irrelevant cases, however. Consider the sequence of actions *pickup*(a), *stack*(a, b), *pickup*(c), *stack*(c, d), *unstack*(a, b), *stack*(a, c). The shorter sequence *pickup*(c), *stack*(c, d), *pickup*(a), *stack*(a, c) would have achieved the same final state. However, the first sequence contains no irrelevant subsequences. To detect cases like this we would need a mechanism that can realize that the actions *stack*(a, b) and *unstack*(a, b) can be removed and then the remaining actions reordered so that *pickup*(a) comes just prior to *stack*(a, c). Future work may be able to find some additional cases that can be detected efficiently.

It is possible to give syntactic tests that given a sequence and a subsequence can test if the subsequence is irrelevant. That is, we have developed syntactic versions of the above semantic definition. However, for our forward chaining planner, we have found that it is most efficient to implement the test directly by simply executing the complement to determine if it is in fact executable and yields an identical final state.

Our definition provides a fairly efficient test for whether or not a particular subsequence is irrelevant. However a given sequence contains an exponential number of subsequences. Detecting whether one of them is irrelevant seems to be hard (we suspect that this is NP-hard). So the question becomes how to test action sequences relatively efficiently and still detect a useful number of ones that contain irrelevant subsequences.

To address this problem we have developed a greedy algorithm that has complexity linear in the length of the action sequence. Thus it imposes an O(depth of the node) overhead on each node expanded in the search space. In the domains we tested the algorithm is able prune away sufficient nodes in the search space to more than make up for this overhead. In future work we intend to analyze the tradeoff between the algorithm's overhead and the reduction in the search space it yields in more detail.

The algorithm examines a sequence of actions and tries to greedily construct a relevant subsequence, thus possibly detecting that the sequence has an irrelevant subsequence (the complement of the greedily constructed relevant subsequence). For each action a_i in the sequence it places all of the previous actions a_j, $j < i$, into a subsequence R and a_i into the subsequence $\overline{R}$. Then for each subsequent action a_ℓ, $i < \ell$, it greedily tries to place a_ℓ into R by checking to see if a_ℓ is executable given the current contents of R. R is a subsequence of actions whose first omitted action is a_i and whose other omitted actions are those that depended on a condition

Inputs: An action sequence $P = \langle a_1, \ldots, a_n \rangle$, a new ground action a, W the world generated by executing the sequence P in the initial state, a list of alternate worlds *AltWorld*, such that *AltWorld*(a_i) is the alternate world associated with action a_i. *AltWorld*(a_i) stores the world generated by the greedy subsequence whose first omitted action is a_i. If *AltWorld*(a_i) is the same world as that produced by P we know that we have detected a subsequence of P that has the same effects as P (hence P contains an irrelevant subsequence). It is assumed that a is executable in W.
Output: *Fail* if we detect that a generates an irrelevant subsequence, else the extended action sequence $P + a$, the new final world $W' = a(W)$, and a new list of alternate worlds, *AltWorld*$'$ one for every action in the extended sequence.

Procedure $\mathbf{Dynamic}(P, a, W, \textit{AltWorld})$

1. $W' := a(W)$;
2. **for** $i := 1$ **to** n
   (a) **if** a is executable in *AltWorld*(a_i) **then**: *AltWorld*$'(a_i) := a($*AltWorld*$(a_i))$ (greedily add it to the subsequence whose first omitted action is a_i);
   else *AltWorld*$'(a_i) :=$ *AltWorld*(a_i);
   (b) **if** *AltWorld*$'(a_i) = W$ **then**: **return**(*Fail*) (we have detected a subsequence whose complement is irrelevant);
3. *AltWorld*$'(a) = W$.
4. **return**($P + a$, W', *AltWorld*$'$).

Table 3: Dynamic Relevance Algorithm

produced by a_i. Finally, it checks to see if the complement of *R* is irrelevant by checking to see if the actions in *R* yield the same final state as the entire sequence. A more efficient incremental implementation is given in Table 3. This algorithm would be called whenever we try to grow an action sequence by adding a new action a. It returns either the incremented action sequence or rejects a as being an illegal extension to the sequence (i.e., it has determined that a generates an irrelevant subsequence). It should be noted that, in the algorithm, if the current plan P is empty then $n = 0$ and the loop of step 2 is never executed.

For example, say the planner examines the action sequence *pickup*(a), *stack*(a, b), *pickup*(c), *stack*(c, d), *unstack*(a, b), *putdown*(a), given above. When it examines the greedy subsequence starting at *pickup*(a), it will place *pickup*(a) in $\overline{R}$, *stack*(a, b) in $\overline{R}$, *pickup*(c) in *R*, *stack*(c, d) in *R*, *unstack*(a, b) in $\overline{R}$, and *putdown*(a) in $\overline{R}$. Then it can detect that $R = \langle pickup(c),$ *stack*$(c, d)\rangle$ yields the same final state as the entire sequence.

As indicated above our greedy algorithm does not detect all irrelevant subsets. One simple example is the action sequence a_1, a_2, a_3 and a_4, where $\neg P_1$, $\neg P_2$, and $\neg P_3$ hold in the initial state. If a_1 adds P_1, a_2 adds P_2, a_3 adds P_3, and a_4 deletes P_1 and P_2, then the subsequence a_1, a_2, and a_4 are irrelevant. However, our algorithm will not detect this. In this example, there are two "roots" in the irrelevant subsequence, a_1 and a_2. Our algorithm can only detect singly rooted irrelevant subsequences. These ideas can be formalized and the algorithm extended to detect k rooted irrelevant subsequences. When $k \geq$ the length of the action sequence, all irrelevant subsequences can be detected. Unfortunately, the complexity of the algorithm grows exponentially with k, and since irrelevance detection must take place during search we doubt that the $k > 1$ versions would be of practical importance. As our empirical results will demonstrate, the greedy $k = 1$ algorithm works well in practice.

One way of viewing dynamic relevance is once again to compare it with what partial order planners do. In particular, SNLP style planners ensure that no causal link is ever violated. This means that every action in any plan explored must produce something "useful". In part this is what dynamic relevance detects: if all of an action's effects are superseded prior to being used that action will form an irrelevant subset. However, dynamic relevance goes further with its ability to detect that certain *subsets* of actions when considered together are redundant.

Combining Dynamic and Static Relevance: Unlike the static relevance algorithm, dynamic relevance does not depend on the syntactic form of the actions, and can work equally well with STRIPS or ADL actions. Furthermore, in those domains where we can use static relevance we can apply both types of relevance detection. In particular, we can apply dynamic relevance pruning when searching the reduced space produced by static relevance. The experiments we present in the next section demonstrate that there is considerable benefit to be gained from the combination.

Empirical Results

For our first experiment, we considered Russell's flat tire domain, where the general task is to change a flat tire using actions such as inflating a new tire, removing an old tire, etc. In the initial state, we have a flat tire on the hub, and a new, uninflated tire, as well as the required tools, in the trunk. The standard "fix a flat tire" goal for this domain (the *fixit* problem in the UCPOP distribution) contains 9 literals. In the experiment we always used the same initial state but generated random planning problems of size n (from 1–9) by setting the goal to be a randomly selected size n subset of the standard goal literals. Further, for each n we choose 15 random goals of size n (sampling with replacement since for some values of n there are less than 15 distinct candidate goals) and computed the average run time. We ran TLPLAN both with and without dynamic relevance detection, and UCPOP using one of its distributed search control mechanisms (which performs better than the default best first search). UCPOP's performance is dependent on the input order of the goal literals, so when testing UCPOP on a particular problem we ran it on 10 random permutations of the goal, taking the average (thus running UCPOP 150 times for each value of n). TLPLAN on the other hand is independent of the goal ordering so only one test was required for each problem. The results are shown in Figure 2.

Most problems with $n > 2$ proved to be too hard for both TLPLAN and UCPOP (within the given search bounds). However, with dynamic relevance detection, TLPLAN was

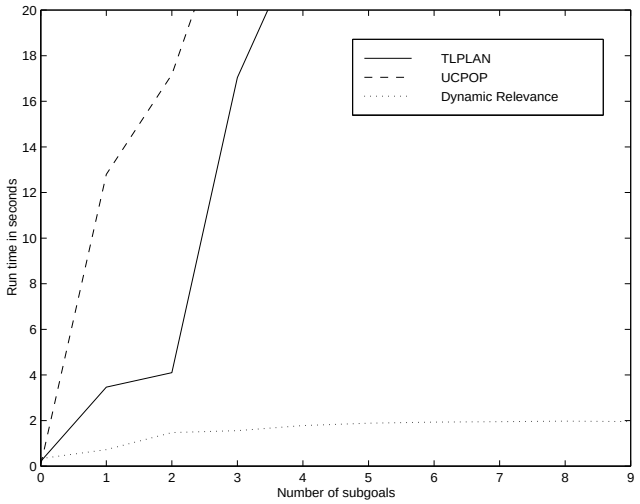

Figure 2: Performance in the flat tire domain.

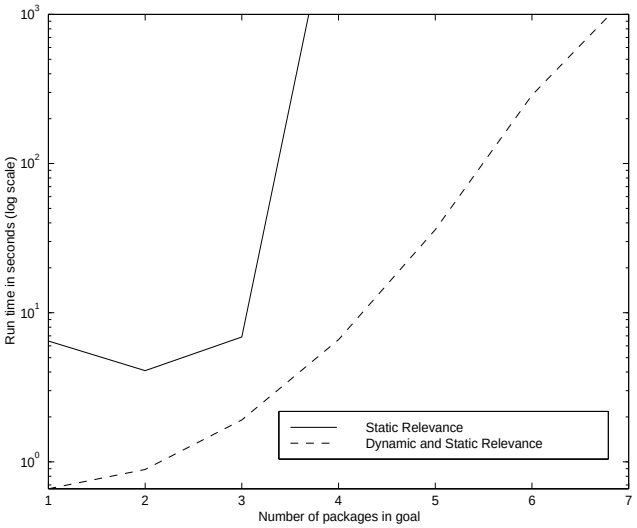

Figure 3: Performance in the simplified logistics domain.

able to solve all of the problems for all tested sizes. In this domain dynamic relevance is able to capture some natural and effective control information to help TLPLAN. For example, once actions were executed to remove the flat tire from the hub, actions putting it back on the hub were not allowed, as they would undermine the effects of the previous actions. Another example is that tools could not be put back into the trunk until they were used.

When $n = 9$ the problems generated were all the same as the standard goal. TLPLAN with dynamic relevance detection took around 1.97 seconds to solve this problem, while UCPOP was only able to solve 41 out of the 150 problems within the given search bounds (i.e., it only solved 41 out of 150 different random permutations of the goal).

The second experiment we conducted used a simplified version of the logistics domain. The logistics domain is a very difficult domain for totally ordered planners, because such planners cannot take advantage of the fact that the movements of the various packages and vehicles are independent of one another. Neither dynamic nor static relevance detection do anything about the issue of total-orderings. Hence, TLPLAN runs very slowly on this domain even with relevance detection. As a result, we used a simplified version of the domain, where there are only two cities, and we are required to send packages from one city to another. The problems we used in the experiment all contained an initial state in which 10 packages were located in one city, and the goal was to send i of the packages to the other city. We ran TLPLAN, with and without both dynamic and static relevance detection, using depth first search. The results are shown in Figure 3.

TLPLAN without relevance detection could not solve the problem even for one package. TLPLAN with only dynamic relevance had similar difficulties and was also unable to solve any of the problems. With static relevance, things improved and TLPLAN was able to solve up the 3 package problem. However with both types of relevance detection, we were able to boost the performance significantly—TLPLAN managed to solve the 6 package problem before the search limit was reached.

There are two distinct ways in which irrelevance occurs in this domain. The first one has to do with the packages that do not appear in the goal. Actions to move these packages greatly (and needlessly) increase the branching factor of the search space. The second one has to do with actions that undo the effects of an earlier action, for example an "(unload package1 truck1)" following a "(load package1 truck1)". Many action sequences contain such irrelevant subsequences. It is clear that static relevance is exactly what is needed to detect the first type of irrelevance, while dynamic relevance is exactly what is required to detect the second type of irrelevance. Either of these irrelevances serve to make TLPLAN very inefficient when solving problems, which is why both dynamic relevance and static relevance detection do not work very well by themselves. When both dynamic and static relevance detection are employed, TLPLAN is given a considerable performance boost.

Conclusions

Our results demonstrate that considerable gains can be achieved by adding the notions of relevance into forward chaining. A traditional argument has been that forward chaining cannot scale up because of the large number of actions a real agent can execute. However, if most of these actions are irrelevant for any particular task (as we would expect them to be), then our results show that fairly simple notions of irrelevance can be utilized to ignore them. Hence, our work shows that this particular argument against forward chaining is invalid.

Nevertheless, it is equally clear from our experiments that forward chaining planners are "still not ready for prime time," even when augmented with notions of relevance. As we discussed in the introduction, we feel that forward chaining has considerable potential, and it is for this reason that

we are pursuing this and other work that is aimed at improving the "base line" for forward chaining planners. That is, we want to enhance the performance of such planners prior to applying any domain independent or domain dependent heuristic control. Our experiments have shown that the next thing that needs to be addressed are the inefficiencies produced by using total orderings. We are currently working on approaches to this problem.

Acknowledgments

This work was supported in part by the Canadian Government through their NSERC and IRIS programs. Thanks also to the reviewers who made a number of useful comments.

References

[Bac95] Fahiem Bacchus. Tlplan (version 2.0) user's manual. Available via the URL ftp://logos.uwaterloo.ca:/pub/bacchus/tlplan-manual.ps.Z, 1995.

[BF97] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[BK96a] F. Bacchus and F. Kabanza. Planning for temporally extended goals. In *Proceedings of the AAAI National Conference*, 1996.

[BK96b] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 141–153. ISO Press, Amsterdam, 1996.

[GS96] A. Gerevini and L. Schubert. Accelerating partial-order planners: Some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research*, 5:95–137, 1996.

[KS96] H. Kautz and Bart S. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the AAAI National Conference*, pages 1194–1201, 1996.

[McD96] D. McDermott. A heuristic estimator for means-end analysis in planning. In *Proceedings of the Third International Conference on A.I. Planning Systems*, 1996.

[MR91] D. McAllister and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the AAAI National Conference*, pages 634–639, 1991.

[NDK97] B. Nebel, Y. Dimopoulos, and J. Koehler. Ignoring irrelevant facts and operators in plan generation. In *Proceedings of the 4th European Conference on Planning*. Springer Verlag, 1997.

[PW92] J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for adl. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114, 1992.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence A Modern Approach*. Prentice Hall, 1995.

[Wel94] D. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.