Conditional Effects in Graphplan *

Corin R. Anderson David E. Smith Daniel S. Weld
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
corin@cs.washington.edu, de2smith@ptolemy.arc.nasa.gov, weld@cs.washington.edu

Abstract

Graphplan has attracted considerable interest because of its extremely high performance, but the algorithm's inability to handle action representations more expressive than STRIPS is a major limitation. In particular, extending Graphplan to handle conditional effects is a surprisingly subtle enterprise. In this paper, we describe the space of possible alternatives, and then concentrate on one particular approach we call *factored expansion*. Factored expansion splits an action with conditional effects into several new actions called *components*, one for each conditional effect. Because these action components are not independent, factored expansion complicates both the mutual exclusion and backward chaining phases of Graphplan. As compensation, factored expansion often produces dramatically smaller domain models than does the more obvious full-expansion into exclusive STRIPS actions. We present experimental results showing that factored expansion dominates full expansion on large problems.

Introduction

Since Graphplan (Blum & Furst 1995) appears to outperform all known STRIPS[1] planners, attention is now focusing on extending Graphplan to handle more expressive action languages. For example, (Gazen & Knoblock 1997; Koehler *et al.* 1997a; 1997b) describe Graphplan-derivative planners that handle disjunction, quantification, and conditional effects. This endeavor is important because the expressive power of ADL (Pednault 1989) provides a much more convenient way to model complex worlds. In this paper, we describe a new method for handling conditional effects in Graphplan, and compare this method to previous work (Gazen & Knoblock 1997; Koehler *et al.* 1997a; 1997b).

Many of ADL's expressive features are easy to implement in Graphplan, but handling conditional effects is surprisingly tricky. Conditional effects allow the description of a single action with context-dependent effects. The basic idea is simple: we allow a special `when` clause in the syntax of action effects. `When` takes two arguments, an *antecedent* and a *consequent*; execution of the action will have the consequent's effect just in the case that the antecedent is true immediately before execution (*i.e.*, much like the action's precondition determines if execution itself is legal — for this reason the antecedent is sometimes referred to as a secondary precondition (Pednault 1989)). Note also that, like an action precondition, the antecedent part refers to the world *before* the action is executed while the consequent refers to the world *after* execution. In this paper, we restrict the consequent to be a conjunction of positive or negative literals. Figure 1 illustrates how conditional effects allow one to define a single action schema that accounts for moving a briefcase that may possibly contain a paycheck and/or keys.

The Full Expansion Approach

One possible way of dealing with conditional effects in Graphplan, and the way adopted by (Gazen & Knoblock 1997), is essentially to expand such actions

---

*David Smith's current address is Nasa Ames Research Center, Mail Stop 269-2, Moffett Field, CA 94035. We thank Mark Peot who provided an initial Lisp implementation of basic Graphplan. Our paper was improved by discussions with Marc Friedman, Keith Golden, Steve Hanks, and Todd Millstein. This research was funded by Office of Naval Research Grants N00014-94-1-0060 and N00014-98-1-0147, by National Science Foundation Grant IRI-9303461, by ARPA / Rome Labs grant F30602-95-1-0024, and by a gift from Rockwell International Palo Alto Research Lab.

[1] Another promising approach is compilation to SAT (Kautz & Selman 1996), but the only times reported to be competitive with Graphplan were from hand-generated SAT problems run with a stochastic solver whose noise parameters were carefully tuned to the problem at hand. The state of the art in automatic generation of SAT formulae from STRIPS planning problems is not yet close to Graphplan performance (Kautz, McAllester, & Selman 1996; Ernst, Millstein, & Weld 1997).

```
move-briefcase (?loc ?new)
    :prec   (and (at briefcase ?loc) (location ?new)
                 (not (= ?loc ?new)))
    :effect (and (at briefcase ?new) (not (at briefcase ?loc))
                 (when (in paycheck briefcase)
                       (and (at paycheck ?new)
                            (not (at paycheck ?loc))))
                 (when (in keys briefcase)
                       (and (at keys ?new)
                            (not (at keys ?loc)))))
```

Figure 1: Conditional effects allow the same `move-briefcase` operator to be used when the briefcase is empty or contains keys and/or paycheck.

into several independent STRIPS operators. As an example, the action schema in Figure 1 could be broken up into four separate STRIPS schemata (as shown in Figure 2): one for the empty briefcase, one for the briefcase with paycheck, one for the briefcase with keys, and one for the briefcase with both paycheck and keys.

```
move-briefcase-empty (?loc ?new)
    :prec   (and (at briefcase ?loc) (location ?new)
                 (not (= ?loc ?new))
                 (not (in paycheck briefcase))
                 (not (in keys briefcase)))
    :effect (and (at briefcase ?new) (not (at briefcase ?loc)))

move-briefcase-paycheck (?loc ?new)
    :prec   (and (at briefcase ?loc) (location ?new)
                 (not (= ?loc ?new))
                 (in paycheck briefcase)
                 (not (in keys briefcase)))
    :effect (and (at briefcase ?new) (not (at briefcase ?loc))
                 (at paycheck ?new) (not (at paycheck ?loc)))

move-briefcase-keys (?loc ?new)
    :prec   (and (at briefcase ?loc) (location ?new)
                 (not (= ?loc ?new))
                 (not (in paycheck briefcase))
                 (in keys briefcase))
    :effect (and (at briefcase ?new) (not (at briefcase ?loc))
                 (at keys ?new) (not (at keys ?loc)))

move-briefcase-both (?loc ?new)
    :prec   (and (at briefcase ?loc) (location ?new)
                 (not (= ?loc ?new))
                 (in paycheck briefcase)
                 (in keys briefcase))
    :effect (and (at briefcase ?new) (not (at briefcase ?loc))
                 (at paycheck ?new) (not (at paycheck ?loc))
                 (at keys ?new) (not (at keys ?loc)))
```

Figure 2: The four STRIPS operators for moving the briefcase.

The trouble with this approach is that it can result in an explosion of the number of actions. If a book could also be in the briefcase, eight action schemata would be required. If a pen could be in the briefcase, sixteen action schemata are required, and so forth. More generally, if an action has n conditional effects, each with m conjuncts in its antecedent, then the number of independent STRIPS actions required in the worst case[2] is 2^{nm}. This explosion frequently occurs with quantified conditional effects. For the briefcase we really want to quantify over all items in the briefcase as shown in Figure 3. In essence, this operator has one conditional effect for each item in the briefcase. If there were twenty items that could be in the briefcase, full expansion would yield over a million STRIPS operators.

```
move-briefcase (?loc ?new)
    :prec   (and (at briefcase ?loc) (location ?new)
                 (not (= ?loc ?new)))
    :effect (and (at briefcase ?new) (not (at briefcase ?loc))
                 (forall ?i (when (in ?i briefcase)
                                  (and (at ?i ?new)
                                       (not (at ?i ?loc))))
```

Figure 3: A quantified conditional operator for moving the briefcase.

The Factored Expansion Approach

A second possibility for dealing with actions with conditional effects, and the one we concentrate on in this paper, is to consider the conditional effects themselves as the primitive elements handled by Graphplan. In essence, this makes *all* effects be conditional. For example, the action schema in Figure 1 would be interpreted as shown in Figure 4.

```
move-briefcase (?loc ?new)
    :effect  (when (and (at briefcase ?loc) (location ?new)
                        (not (= ?loc ?new)))
                   (and (at briefcase ?new)
                        (not (at briefcase ?loc))))
             (when (and (at briefcase ?loc) (location ?new)
                        (not (= ?loc ?new))
                        (in paycheck briefcase))
                   (and (at paycheck ?new)
                        (not (at paycheck ?loc))))
             (when (and (at briefcase ?loc) (location ?new)
                        (not (= ?loc ?new))
                        (in keys briefcase))
                   (and (at keys ?new) (not (at keys ?loc))))
```

Figure 4: Fully conditionalized schema for moving a briefcase.

The advantage of this "factored expansion" is an increase in performance. By avoiding the need to expand actions containing conditional effects into an exponential number of plain STRIPS actions, factored

[2]The number of antecedent conjuncts, m, participates in the exponent because the behavior of the action varies as a function of *each* conjunct — if any one conjunct is false then the corresponding effect is inactive. The worst case comes about if all nm propositions are distinct in which case all combinations must be enumerated. The 2^{nm} number can actually be reduced to n^m as explained in (Gazen & Knoblock 1997), however this is still very large.

expansion yields dramatic speedup. But this increased performance comes at the expense of complexity:

- Because factored expansion reasons about individual effects of actions (instead of complete actions), more complex rules are required in order to define the necessary mutual exclusion constraints during planning graph construction. The most tricky extension stems from the case when one conditional effect is *induced* by another — *i.e.*, when it is impossible to execute one effect without causing the other to happen as well.

- Factored expansion also complicates the backward chaining search for a working plan in the planning graph because of the need to perform the analog of *confrontation* (Penberthy & Weld 1992; Weld 1994), *i.e.*, subgoal on the negated preconditions of undesirable conditional effects.

The IP^2 Approach

A third possible method for handling conditional effects is employed by the IP^2 planner (Koehler *et al.* 1997b). The IP^2 system sits halfway between the full and factored expansion methods, using techniques similar to both. A more thorough discussion of the IP^2 system is made in the empirical results section, after the requisite Graphplan background is discussed.

Overview

In the next section we briefly review the basic Graphplan algorithm and extend it to handle negated preconditions and disjunction; these extensions are necessary for handling conditional effects. Following the Graphplan background, we give a detailed development of our factored expansion approach with illustrative examples. Next, we present empirical evidence that factored expansion yields dramatic performance improvement over full expansion. Finally, we offer some discussion of related issues and work and give concluding remarks.

Graphplan Background

We briefly summarize the basic operation of the Graphplan algorithm as introduced in (Blum & Furst 1995; 1997). Graphplan accepts action schemata in the STRIPS representation — preconditions are conjunctions of positive literals and effects are a conjunction of positive or negative literals (*i.e.*, composing the add and delete lists). Graphplan alternates between two phases: *graph expansion* and *solution extraction.* The graph expansion phase extends a *planning graph* until it has achieved a necessary (but insufficient) condition for plan existence. The solution extraction phase performs a backward-chaining search for an actual solution; if no solution is found, the cycle repeats.

The planning graph contains two types of nodes, proposition nodes and action nodes, arranged into levels. Even-numbered levels contain proposition nodes, and the zeroth level consists precisely of the propositions that are true in the initial state of the planning problem. Nodes in odd-numbered levels correspond to action instances; there is an odd-numbered node for each action instance whose preconditions are present and are mutually consistent at the previous level. Directed edges connect proposition nodes to the action instances at the next level whose preconditions mention those propositions. And directed edges connect action nodes to subsequent propositions made true by the action's effects.

The most interesting aspect of Graphplan is its use of local consistency methods during graph creation — this appears to yield a dramatic speedup during the backward chaining search. Graphplan defines a binary mutual exclusion relation ("mutex") between nodes in the same level as follows:

- Two action instances at level i are mutex if either
  - *Interference / Inconsistent Effects:* one action deletes a precondition or effect of another, or
  - *Competing needs:* the actions have preconditions that are mutually exclusive at level $i-1$.

- Two propositions at level j are mutex if all ways of achieving the propositions (*i.e.*, actions at level $j-1$) are mutex.

Suppose that Graphplan is trying to generate a plan for a goal with n conjuncts, and it has finally extended the planning graph to an even level, i, in which all goal propositions are present and none are pairwise mutex. Graphplan now searches for a solution plan by considering each of the n goals in turn. For each such proposition at level i, Graphplan chooses an action a at level $i-1$ that achieves the goal. This is a backtracking choice — all possible actions must be considered to guarantee completeness. If a is consistent (non-mutex) with all actions that have been chosen so far at this level, then Graphplan proceeds to the next goal, otherwise if no such choice is available, Graphplan backtracks. After Graphplan has found a consistent set of actions at level $i-1$ it recursively tries to find a plan for the set of all the preconditions of those actions at level $i-2$. The base case for the recursion is level zero — if the propositions are present there, then Graphplan has found a solution. If, on the other hand, Graphplan fails to find a consistent set of actions at some level and backtracking is unsuccessful, then it continues to alternate between growing the planning graph and searching for a solution (until it reaches a set limit or the graph levels off).

Negated and Disjunctive Preconditions

Although methods for handling negated and disjunctive preconditions were not presented in (Blum & Furst 1995), they are both straightforward and essential prerequisites for handling conditional effects. Clearly proposition p and $\neg p$ are mutually exclusive in any given level. Whenever an action instance deletes a proposition (*i.e.* has a negated literal as an effect), one must add that negative literal to the subsequent proposition level in the planning graph.

Disjunctive preconditions are also relatively easy. Conceptually, the precondition (which may contain nested ands and ors) is converted to disjunctive normal form (DNF). Now when the planning graph is extended with an action containing multiple disjuncts, an action instance may be added if *any* disjunct has all of its conjuncts present (non-mutex) in the previous level. During the backchaining phase, if the planner at level i considers an action with disjunctive preconditions, then it must consider all possible precondition disjuncts at level $i-1$ to ensure completeness.

Conditional-Effects Graphplan

The central concept of the factored expansion approach to handling conditional effects is that of an action *component.* Formally, a component is a pair consisting of a consequent (conjunction of literals) and an antecedent (disjunction or conjunction is allowed). An action has one component per *effect* (where effect is defined in (Penberthy & Weld 1992, Section 2.3)). A component's antecedent is simply the action's primary precondition conjoined with the antecedent of the corresponding conditional effect; a component's consequent is simply the consequent of the corresponding conditional effect.[3]

Using this approach, every ordinary STRIPS action would have only one component. However, actions with conditional effects would have one component for the unconditional effects, and one component for each conditional effect. As an example suppose that action A has precondition p and three effects e, `(when` q $(f \wedge \neg g)$`)`, and `(when` $(r \wedge s)$ $\neg q$`)`. This action would have three components:

[3] Just as it is useful to consider lifted action schemata in addition to ground actions, we will consider lifted component schemata as well as ground components. Unless there is some potential confusion we shall call both the lifted and ground versions *components.*

1. C_1 has antecedent p and consequent e.
2. C_2 has antecedent $p \wedge q$ and consequent $f \wedge \neg g$.
3. C_3 has antecedent $p \wedge r \wedge s$ and consequent $\neg q$.

Revised Mutex Constraints

For the most part, graph expansion works in the same manner as in the case of STRIPS actions, except at odd-numbered levels, we add instances of components instead of action instances. An instance of component C_i is added when its antecedents are all present and pairwise non-mutex at the previous proposition level. For example, if p and q are the only literals present in level $i-1$, then level i would contain an instance of C_1 and C_2 (but not C_3). When a component is added to an odd-numbered level, then its consequent is added to the subsequent level in the obvious way. Thus, in our example, e and f and $\neg g$ would all be added to level $i+1$. See Figure 6.

So far, this is straightforward, but the handling of mutex constraints is actually rather subtle. Recall that mutex constraints are defined recursively in terms of the constraints present at the previous level. The definition for proposition levels is unchanged from vanilla Graphplan:

- Two propositions p and q at level j are mutex if all ways of achieving p (*i.e.* all level $j-1$ components whose consequents include p as a positive literal) are pairwise mutex with all ways of achieving q.

There are two changes to the definition of mutex constraints for components. The interference condition (originally defined in the Graphplan Background section) has an extra clause (so it implies fewer mutex relations), and there is also a new way of deriving a mutex relation, the *induced component* condition:

- Two components C_n and C_m at level i are mutex if either:
  - *Interference / Inconsistent Effects:* components C_n and C_m come from *different* action instances *and* the consequent of component C_n deletes either an antecedent or consequent of C_m (or vice versa), or
  - *Competing Needs:* C_n and C_m have antecedents that are mutex at level $i-1$, or
  - *Induced component:* There exists a third component C_k that is mutex with C_m *and* C_k is *induced* by C_n at level i (see definition below and Figure 5).

Intuitively, component C_n *induces* C_k *at level* i if it is impossible to execute C_n without executing C_k; more formally, we require that:

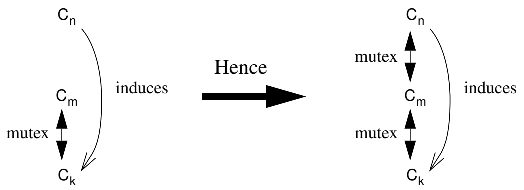

Figure 5: If C_n induces C_k and C_k is mutex with C_m, then C_n is also mutex with C_m.

1. C_n and C_k derive from the same action schema with the same variable bindings, and
2. C_k is non-mutex with C_n, and
3. the *negation* of C_k's antecedent *cannot* be satisfied at level $i-1$. In other words, suppose that the antecedent of C_k is $p_1 \wedge \ldots \wedge p_s$, then we require that, for all j, either $\neg p_j$ is absent from level $i-1$ *or* $\neg p_j$ is mutex with a conjunct of C_n's antecedent.

We now discuss in more detail the two differences between our definition of mutual exclusion and the definition used by vanilla Graphplan.

Interference Our first change to the mutex definition reduces the number of mutexes by adding a conjunct to the interference clause. This is because actions often clobber their own preconditions. For example, one can't put one block on another unless the destination is clear, yet the act of putting the block down makes the destination be not clear. This self-clobbering behavior doesn't bother Graphplan (because a STRIPS action's effects are never compared to its preconditions), and our modification of the interference condition is simply a small generalization to ensure that there is no problem (*i.e.* no mutex constraint generated) when one conditional effect clobbers the antecedent of another conditional effect *of the same action.* (Note that this generalization is only necessary for factored expansion — if full expansion is used, the vanilla definition of interference is fine.)

Induced components Our second change to the mutex definition is an optimization that increases the number of mutexes through the notion of induction. A few examples will make the definition of induced component more intuitive, show why the notion of component induction is level-dependent, and explain how it fits into the mutex picture. First, consider a simple case: component C_2 (from the example earlier) induces C_1 because both components come from action A and the antecedent of C_2 is $p \wedge q$, which entails p. Thus, any plan that relies on the effects of C_2 had better count on the effects of C_1

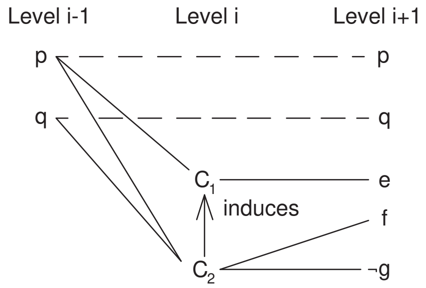

Figure 6: An example of an induced component.

as well — there is no way to avoid them using the equivalent of *confrontation* (Penberthy & Weld 1992; Weld 1994). Essentially, we are recording the impossibility of executing the conditional part of an action (C_2) without executing the unconditional part (C_1) as well. Hence we say that C_1 is induced by C_2 (Figure 6). Thus if C_1 is mutex with some component C_m from a *different* action, then C_2 should be considered mutex with C_m as well because execution of C_2 induces execution of C_1 that precludes execution of C_m.

While C_2 induces C_1 at *every* level, there are cases where the set of induced components is level dependent. Suppose that proposition level $i-1$ contains q but does not contain $\neg q$. Can the negation of C_2's antecedent be made true at level $i-1$? Since $\neg q$ is not present at level $i-1$, the only way to avoid C_2 is to require $\neg p$ at level $i-1$. However, $\neg p$ is mutex with p, which is the antecedent of C_1. Hence at level i, component C_1 induces C_2 as well — there is no way to execute C_1 (at this level) without executing C_2 as well.

Suppose that action B has spawned component C_m, which has no antecedent and has g as consequent. Furthermore, suppose that the goal is to achieve $g \wedge e$ and that precisely three components C_1, C_2, and C_m are present in level i. Since C_1 (which produces e) induces C_2 (at level i) which deletes g, C_1 is mutex with C_m at level i. Thus g and e are mutex at $i+1$, which correctly reflects the impossibility of achieving both goals (Figure 7).

On the other hand, suppose that $\neg q$ was present at $i-1$. Then the negation of C_2's antecedent could be made true, and C_1 would not induce C_2 at level i. Hence, C_1 would not be mutex with C_m. This correctly reflects the possibility of confrontation, and below we show how to modify the backward chaining search to ensure that $\neg q$ is raised as a goal at level $i-1$ if C_1 is chosen to support e and C_m is chosen to support g.

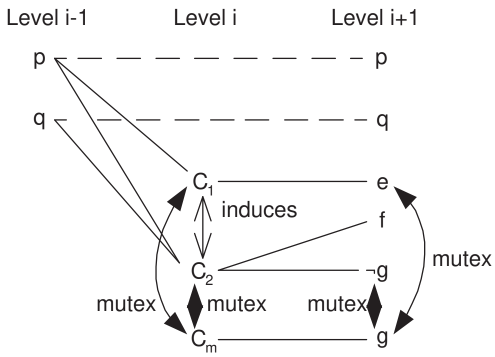

Figure 7: Since induction makes C_1 mutex with C_m, additional mutex relations are added at proposition level $i+1$.

Revised Backchaining Method

As the discussion of induced components suggested, factored expansion of context-dependent actions into components also complicates the backchaining search for solution plans.

Here's a simple example that illustrates the issues. Action D has precondition p and two effects e, and the conditional effect `(when` q f`)`. Factored expansion yields two components:

1. C_4 has antecedent p and consequent e.
2. C_5 has antecedent $p \wedge q$ and consequent f.

First, let's consider how the backward chaining phase of Graphplan normally works. Suppose that level $i-1$ has p, q and $\neg f$ present. Thus level i has components C_4 and C_5 present. Suppose that no other components are present except for the no-op components that carry $\neg f$ *etc.* forward. If the goal is f then one way to achieve the goal at $i+1$ is via component C_5 and so the backward chainer will subgoal on $p \wedge q$ at level $i-1$.

Now consider the more interesting case where the goal is $e \wedge \neg f$ (Figure 8). The planner must consider using C_4 to achieve e and using a no-op to maintain $\neg f$, but it must ensure that component C_5 doesn't clobber $\neg f$. Thus the planner must do the equivalent of *confrontation* — subgoaling on the negation of C_5's antecedent. Thus the goal for level $i-1$ is $p \wedge \neg(p \wedge q)$, which simplifies to $p \wedge \neg q$.

As a final example, consider the following world (Figure 9). Action A has no preconditions and has two effects: g and `(when` r $\neg h$`)`; action B also has no preconditions and has effects h and `(when` z r`)`. Factored expansion yields four components from these actions: A_1, A_2, B_1, and B_2. Suppose that the propositions

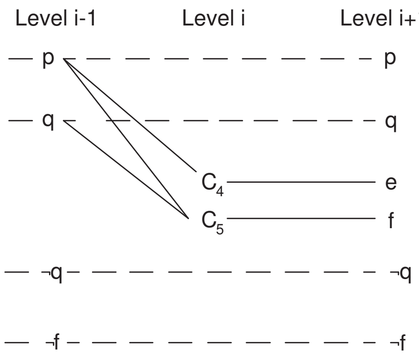

Figure 8: Planning graph when backchaining occurs; in order to prevent C_5 from clobbering $\neg f$ the planner must use confrontation to subgoal on $\neg q$ at level $i-1$.

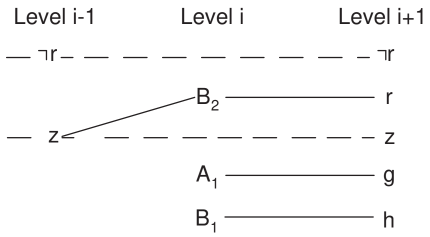

Figure 9: Even though A_2 does not appear in the planning graph, the component may fire if action B is taken before action A.

at level $i-1$ are $\neg r \wedge z$ and the goals are $g \wedge h$. In this case three of the components are present, but A_2 with effect $\neg h$ is absent because its precondition r is not present at $i-1$.

Because the goals are present at level $i+1$, supported by the non-mutex components A_1 and B_1, it would seem that a plan that executes actions A and B, in either order, would be correct. However, in reality, the order in which these actions are executed does matter. If action B were executed first, then both B_1 and B_2 would fire (all of B_2's preconditions hold at level $i-1$). B_2 establishes the r proposition, and thus, when action A is executed, the preconditions of component A_2 are true. Hence, A_2 also fires, and clobbers the goal h — *but A_2 wasn't even in the planning graph at this level.* The observation to be made from this example is that, even though a component may not occur in the planning graph at a particular level, it may be necessary to confront that component at that level anyway.

With these examples in mind, we are nearly ready to present the general algorithm for backchaining search. To make the presentation simpler, we define one more term. We say that component C_m is *possibly in-*

duced by component C_s if C_m and C_s are both derived from the same action with the same variable bindings. Note that C_s possibly induces C_s. The algorithm for backchaining can now be expressed as follows:

1. Let $\mathcal{SC}_i \leftarrow \{\}$. $\mathcal{SC}_i$ is the set of components selected at level i to support the goals at level $i+1$.

2. For each goal g at level $i+1$, choose (backtrack point) a supporting component C_s at level i that has g as an effect and that is non-mutex with the components in $\mathcal{SC}_i$.

2.1 If C_s is not already in $\mathcal{SC}_i$, then add C_s to $\mathcal{SC}_i$ and add the preconditions of C_s to the goals for level $i-1$.

3. For each pair of components C_s, C_t in $\mathcal{SC}_i$, consider all pairs of components C_m, C_n, where C_s possibly induces C_m and C_t possibly induces C_n. If C_m and C_n are mutex, then we must choose (backtrack point) one of C_m or C_n to confront (see discussion of confrontation below).

4. If $i = 1$, then the completed plan is the set of actions from which the components in $\mathcal{SC}_1$, $\mathcal{SC}_3$, $\mathcal{SC}_5$, ... belong. Otherwise, reduce i by 2 and repeat.

The remaining detail in this algorithm is the handling of confrontation. Let's suppose that component C_n is to be confronted. Confrontation involves constraints at *two* time points. First, the antecedent of C_n must be prevented from being true at the previous proposition level, $i-1$. Second, the antecedent of C_n must not be allowed to *become* true under any ordering of the steps at level i. Otherwise, C_n might fire, if the action to which C_n belongs is executed after some other component establishes C_n's antecedent.

There are several ways to satisfy both of these requirements for confrontation. Perhaps the most straightforward, and the one that we implement, is to simply add one or more no-ops to $\mathcal{SC}_i$ that carry the negation of C_n's antecedent[4] from level $i-1$ to $i+1$. The preconditions of the no-ops satisfy the first requirement from above (*i.e.*, that C_n's antecedent not be true at level $i-1$). And, by selecting the no-ops, no other component that *makes* C_n's antecedent true can be selected at level i because it would violate mutex relations with a no-op.

Note that there are optimizations to this rule. If a particular no-op that we wish to select is not available at level i, then there is no way that the corresponding term of C_n's antecedent could be made false at this level. Thus, the planner can immediately disregard all no-op sets that include any no-ops not present at level i. Also, when negating the antecedent of C_n, some logic simplification may be possible, as was the case in the example above.

There are also several optimizations to the new backchaining search algorithm that one could adopt (and that we have adopted). One optimization is to roll steps 2 and 3 together, confronting possibly induced components as $\mathcal{SC}_i$ is being grown. Another optimization is in the choice between C_m and C_n in step 3. If C_m or C_n are already in $\mathcal{SC}_i$, then that component cannot be confronted, and confrontation on the other must be attempted. Similarly, if one of C_m or C_n has already been confronted, either successfully or unsuccessfully, a second attempt at confrontation won't prove otherwise.

Expansion time

Besides the choice of *expansion method* when implementing conditional effects, one must also make a choice of *expansion time*. With *compile-time* expansion, one creates new components before starting to construct the planning graph. With *run-time* expansion one performs this expansion as each level in the graph is built.

Both full and factored expansion allow operators to be broken up into components at either compile- or run-time. The advantage to compile-time expansion is that the expansion is performed only once. The disadvantage is that, for full-expansion, there may be exponentially many components to create at expansion time. This is not an issue for factored expansion, though, where the number of components is linear with the number of conditionals. Thus, compile-time expansion is an attractive choice when implementing factored expansion. Note, however, that even though the operators can be factored into their components at compile-time, if one chooses to use factored expansion, the induces relation must still be determined at run-time (because induction can be level dependent).

One really can't avoid the exponential blow-up problem with full expansion, although one can put it off for as long as possible. Using run-time expansion, each operator is expanded into only those components that are applicable at the current level. This method allows the planner to not create all the components at once, but rather create only those components that are valid at each level. The downside of this method is the added computational overhead of the expansion at each level. But this overhead is usually much less than that of creating exponentially many components initially. Hence, run-time expansion is a good complement to full-expansion.

[4] Note that there may be search required in this process. If the *negation* of C_n's antecedent is disjunctive, then we choose (backtrack point) one disjunct and add its no-op. If, on the other hand, the negated antecedent is conjunctive, then we must add no-ops for each conjunct.

Empirical Results

We conducted two experiments to evaluate methods of handling conditional effects in Graphplan. In the first experiment, we compared full expansion to factored expansion in our implementation of Graphplan. In the second experiment, we compared factored expansion to IP^2 (Koehler *et al.* 1997b).

Full Expansion vs. Factored Expansion

In our comparison of full vs. factored expansion, the full expansion was done at run-time while the factored expansion was performed at compile-time. Both methods are part of the same implementation, written in Common Lisp. Experiments were carried out on a 200 MHz PentiumPro workstation running Linux.

We ran both planners on a series of problems from several domains that use conditional effects. The general trend was that for "easy" problems (problems whose plan could be found within about a second), the full expansion method was faster than the factored expansion method. But when the planner required more than a few seconds to find a plan, the factored expansion method ran faster. Three experiments that highlight this trend were performed in the Briefcase World domain, the Truckworld domain and the Movie Watching domain.

The Briefcase World and the Truckworld experiments are a series of problems parameterized by the number of objects in the world. The Briefcase World experiment involves moving objects from home to school. The Truckworld problems require moving pieces of glass from one location to another, without breaking the glass.

The Movie Watching domain involves problems of preparing to watch a movie at home. Preparation includes rewinding the movie, resetting the time counter on the VCR, and fetching snacks to be eaten during the movie. The problems in this experiment are parameterized by the number of possibilities for each snackfood (for instance, there are 5, 6, or 7 bags of chips, and only bag is necessary to achieve the goal).

Figure 10 shows a performance comparison between the full and factored expansion methods. Each datapoint represents a problem from one of the three experiments, averaged over five trials. Standard deviation values range from 1% to 8% of the mean values. Points above the line are problems for which factored expansion Graphplan runs faster than full expansion Graphplan.

We can see from this figure that the larger problems in the Briefcase World and Truckworld domains cause great difficulty for full expansion. The performance decrease comes from the fact that these problems give rise to an exponential number of fully-expanded actions. Because factored expansion creates only a linear number of components with respect to the number of objects, factored expansion's performance doesn't suffer. We also note in this figure that both methods perform equally well in the Movie Watching domain. By using full expansion, the planner doesn't need to reason about induced mutexes explicitly, whereas induced mutexes are detected explicitly by the factored expansion algorithm.

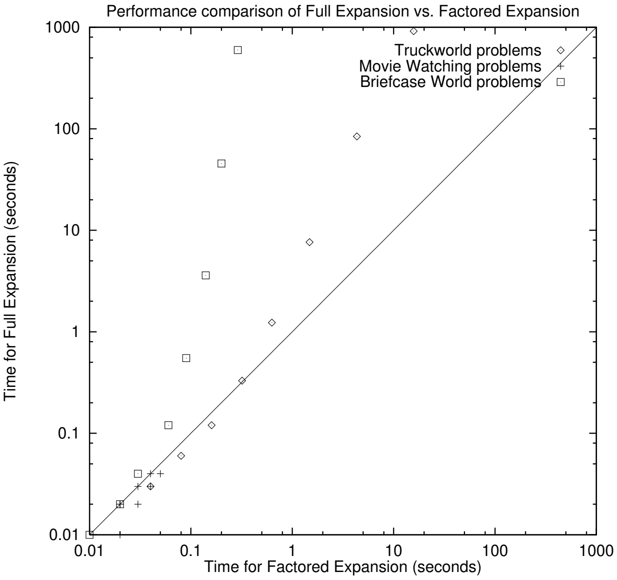

Figure 10: Performance comparison of factored expansion versus full expansion. Datapoints represent parameterized problems in the Briefcase World, Truckworld and Movie Watching domains. The line separates the problems for which factored expansion runs faster (datapoints above the line) from problems for which full expansion runs faster (datapoints below the line).

IP^2 vs. Factored Expansion

In the IP^2 system, conditional effects are handled in a way that is somewhat similar to our factored expansion. The primary differences are that (a) actions are considered as whole units with separate effect clauses for each conditional effect; (b) two actions are marked as mutex only if their unconditional effects and preconditions are in conflict, and (c) atomic negation is not handled explicitly by the algorithm. These differences allow IP^2 to use simpler mutex rules, but reduce the number of mutex constraints that will be found.

When planning problems don't involve mutexes between the conditional effects of several actions, the performance of IP^2 and factored expansion are similar. However, in the Movie Watching domain, IP^2 does not perform as well because it does not identify the induced mutex between the goals at level 2. Because of this, IP^2 has to conduct a great deal of search at level 2 before it can be sure that the goals are not satisfiable at this level.

In our comparison experiment between the C implementation of IP^2 and the Lisp implementation of factored expansion, we ran both systems on the three domains used in the full versus factored expansion experiment. A logscale plot of the results appears in Figure 11.

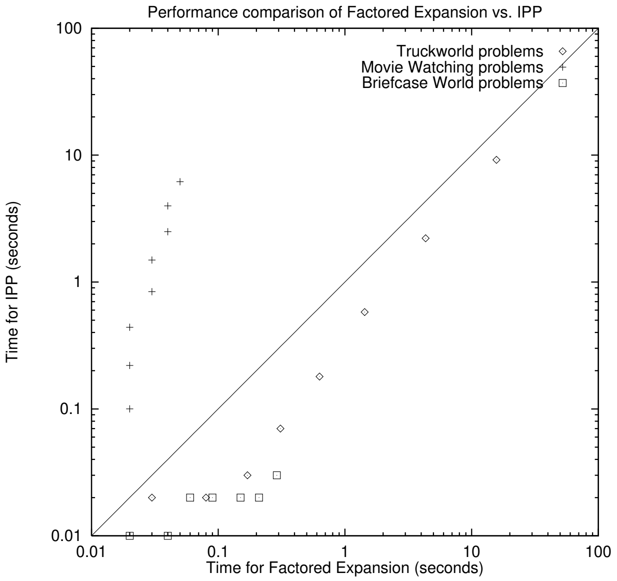

Figure 11: Performance comparison of IP^2 version 3.2 (C) vs. factored expansion (Lisp). Times taken on a 200 MHz PentiumPro Linux workstation with 64MB RAM. Datapoints represent problems from the Briefcase World, Truckworld and Movie Watching domains. The line separates the problems for which factored expansion ran faster (datapoints above the line) from problems for which IP^2 ran faster (datapoints below the line).

The datasets in this experiment tell us three things. First, the Movie Watching problems show that, when induced mutexes are not identified, the resulting unnecessary search is expensive. Factored expansion Graphplan doesn't search for a solution until the fourth level in the planning graph, while IP^2 begins its search after the second level is added. Second, the Briefcase World problems show that the time to extend the planning graph is sometimes as important, if not more so, than the time to search the planning graph.[5] In the Briefcase World problems, the time is heavily dominated by graph expansion. IP^2 is highly optimized for this task, while our factored expansion implementation doesn't have any such optimizations. Finally, the Truckworld problems show that there are domains for which IP^2 and Factored Expansion have similar performance.

Related Work

Since publication of the original Graphplan papers (Blum & Furst 1995; 1997), several researchers have investigated means for extending the algorithm to handle more expressive action languages. As discussed earlier, IP^2 (Koehler *et al.* 1997b) is a Graphplan derivative that uses a technique similar to factored expansion, although induced mutex relationships are not discovered. IP^2 also includes a number of optimizations that greatly improve its performance. One optimization precalculates all the ground instances of the operators in the input domain. By generating a complete set of legal instantiations of operators, no type checks or unifications are necessary during graph expansion. Another optimization removes facts and operators that are deemed irrelevant to the goal (Nebel, Dimopoulos, & Koehler 1997). A third optimization removes *inertia*, facts that are true at all graph levels. In domains that have static attributes associated with objects, removing inertia greatly improves performance – these static facts don't have to be considered when extending the planning graph or when searching backwards through the graph.

In (Gazen & Knoblock 1997), a preprocessor has been defined and implemented to convert UCPOP-style domains into simpler STRIPS style domains. This preprocessor allows Graphplan to solve problems whose definitions are given in the full expressiveness of ADL. In this work, conditional effects are handled by a compile-time full-expansion of each operator.

Kambhampati's group has considered several extensions to Graphplan. For example, (Kambhampati, Lambrecht, & Parker 1997) describes how to implement negated and disjunctive preconditions in Graphplan, and also sketches the full expansion strategy for handling conditional effects. Their paper also hints that an approach like our factored method might prove more efficient, but does not appear to recognize the need for either revisions to the mutex definition in order to account for induced components, or for confrontation during backchaining.

[5] See (Kambhampati, Lambrecht, & Parker 1997) for additional discussion on this matter plus an impressive regression-based focussing technique for optimizing graph expansion.

Conclusions

In this paper, we've introduced the factored expansion method for implementing conditional effects in Graphplan. The principle ideas behind factored expansion are 1) breaking the action into its *components*, 2) modifying the rules for mutual exclusion by adding the notion of mutexes from induced components, and 3) modifying the rules for backchaining to incorporate confrontation.

We compared factored expansion to full expansion (Gazen & Knoblock 1997) and the IP^2 method (Koehler *et al.* 1997a). There appear to be two potential forms of combinatorial explosion:

- **Instantiation Explosion**. As illustrated in Figure 2, the full expansion approach can compile an action containing conditional effects into an exponential number of STRIPS actions. Neither factored expansion nor the IP^2 method fall prey to this problem.
- **Unnecessary Backchaining**. Since the IP^2 method deduces a subset of the possible mutex relations, it will sometimes be fooled into thinking a solution exists and hence will waste time in exhaustive backchaining before it realizes its mistake. This was illustrated in the Movie Watching domain in Figure 11. Neither full expansion nor factored expansion have this problem.

We close by noting that efficient handling of conditional effects is just one aspect of a fast planning system. As noted earlier, IP^2 has several orthogonal optimizations that are also worthwhile for some of the domains we tested.

References

Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proc. 14th Int. Joint Conf. AI*, 1636–1642.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *J. Artificial Intelligence* 90(1–2):281–300.

Ernst, M.; Millstein, T.; and Weld, D. 1997. Automatic sat-compilation of planning problems. In *Proc. 15th Int. Joint Conf. AI.*

Gazen, B., and Knoblock, C. 1997. Combining the expressivity of UCPOP with the efficiency of Graphplan. In *Proc. 4th European Conference on Planning.*

Kambhampati, R.; Lambrecht, E.; and Parker, E. 1997. Understanding and extending graphplan. In *Proc. 4th European Conference on Planning.*

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. 13th Nat. Conf. AI*, 1194–1201.

Kautz, H.; McAllester, D.; and Selman, B. 1996. Encoding plans in propositional logic. In *Proc. 5th Int. Conf. Principles of Knowledge Representation and Reasoning.*

Koehler, J.; Nebel, B.; Hoffmann, J.; and Dimopoulos, Y. 1997a. Extending planning graphs to an ADL subset. In *Proc. 4th European Conference on Planning.*

Koehler, J.; Nebel, B.; Hoffmann, J.; and Dimopoulos, Y. 1997b. Extending planning graphs to an ADL subset. TR 88, Institute for Computer Science, University of Freiburg. See `http://www.informatik.uni-freiburg.de/~koehler/ipp.html`.

Nebel, B.; Dimopoulos, Y.; and Koehler, J. 1997. Ignoring irrelevant facts and operators in plan generation. In *Proc. 4th European Conference on Planning.*

Pednault, E. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. 1st Int. Conf. Principles of Knowledge Representation and Reasoning*, 324–332.

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. Principles of Knowledge Representation and Reasoning*, 103–114. See also `http://www.cs.washington.edu/research/projects/ai/www/ucpop.html`.

Weld, D. 1994. An introduction to least-commitment planning. *AI Magazine* 27–61. Available at `ftp://ftp.cs.washington.edu/pub/ai/`.