Handling of Conditional Effects and Negative Goals in IPP

Jana Koehler

Institute for Computer Science
Albert Ludwigs University
Am Flughafen 17
79110 Freiburg, Germany
koehler@informatik.uni-freiburg.de

Abstract

This report describes an extension of planning graphs to handle conditional effects and true negation in the system IPP. Starting from the planning formalism ADL, a formal semantics for parallel plans containing actions with conditional effects is defined. We review the formalism of planning graphs, which was developed in the Graphplan system and show how the approach can be extended to deal with the full expressivity of ADL. In contrast to Graphplan, IPP does not generate the planning graph explicitely, but builds only one fact and action layer. Furthermore, actions and states are represetended using a bitvector representation, which allows for the time- and memory-efficient implementation of the sound and complete search algorithm.

Technical Report No. 128

Contents

1 Technical Preliminaries

We begin straightforward with the definition of the technical framework, which will be used in this report. Important notions such as a planning problem, a plan, an operator, an action, and a state are defined. Two planning languages are considered: PDDL, which is a syntactic variant of the ADL formalism [Ped91] and BRL, IPP's experimental language to investigate resource-constrained planning problems [Koe99]. Both languages allow to formally investigate two important extensions of STRIPS:

- conditional effects, which extend the logical expressivity of a planning language,
- resource effects, which add non-logical constructs to a planning language.

The approach in IPP treats both extensions individually. But the algorithms can be easily combined to handle conditional resource effects as it is shown in [Koe99].

1.1 States, Planning Problems, Operators

Starting with version 4.0, explicit negation was introduced in IPP. This means, states are no longer described by sets of logical atoms, but now sets of ground literals (sometimes called *facts*) are admitted.

Definition 1 *The set of all ground literals is denoted with P. A* state *$S \in 2^P$ is a set of ground literals.*

A state is complete and consistent if for each ground literal $p \in P$ either p or $\neg p$ is true and contained in the state. If both p and $\neg p$ are true or if both are false and contained in the state, the state is inconsistent. Alternatively, if incomplete states are admitted one can interpret the falsehood of both p and $\neg p$ as an incomplete state description in which nothing is known about the truth of p.[1]

The inconsistent state, in which at least one ground atom and its negation are contained, is denoted with $\perp$ representing logical falsehood. Logical truth is represented with $\top$. It is used to represent the empty precondition or effect condition that is satisfied in every state.

Definition 2 *A* planning problem *$(\mathcal{O}, D, I, \mathcal{G})$ is a 4-tuple where $\mathcal{O}$ is the set of operators, D is the domain of discourse (a finite set of typed constants denoting objects), and I (the initial state) and $\mathcal{G}$ (the goals) are described by a logical formula.*

An operator is a more complex construct. It has a name, a parameter list, preconditions, and effects. The explicit use of a name is not really necessary from a theoretical point of view, but it is a widely agreed convention in the planning community.

Definition 3 *An* operator *is a 4-tuple consisting of*

1. *a* name*, which is a string,*
2. *a* parameter list *of typed variables,*

[1] IPP works on complete state descriptions.

3. the preconditions,

4. the effects.

Depending on the specific planning language a system accepts, the syntactic structure of preconditions and effects has to satisfy various syntactic restrictions.

Definition 4 *In a* STRIPS operator*, the preconditions and the effects are restricted to sets of literals. All variables have to occur as parameters of the operator.*

Definition 5 *A STRIPS action is an instantiation of the parameters of a STRIPS operator with constants of the corresponding type, where all literals have to be ground.*

A typical example is the **stack** action that puts a block named a on top of a block with name b if b is clear and a robot arm is holding a. As an effect, the robot arm becomes empty, a is clear, it is on b, but b is no longer clear and the robot arm is no longer holding a.[2]

stack(a,b):
:precondition *clear(b), holding(a)*
:effect *arm-empty, clear(a), on(a,b),* ¬ *clear(b),* ¬ *holding(a).*

In the following, simple STRIPS actions will be written without keywords, but as implications where the antecedent shows the preconditions and the consequent lists the unconditional effects:

stack(a,b): *clear(b), holding(a)*
⇒ *arm-empty, clear(a), on(a,b),* ¬ *clear(b),* ¬ *holding(a).*

Definition 6 *A* plan *is a partially ordered sequence of actions.*

1.2 Conditional Effects

The possibility to formulate conditional effects (sometimes called context-sensitive effects) significantly contributes to the expressivity of a planning formalism and is one of the important features of the ADL planning language [Ped91]. Depending on the context, in which an action is executed, it can now have different effects.

Definition 7 *An effect is* conditional *iff it has the form* $pre_i \Rightarrow eff_i$ *where* pre_i *and* eff_i *are sets of literals.*

Definition 8 *A conditional effect is* universally quantified *iff it has the form* $\forall\, x\ pre_i \Rightarrow eff_i$*. All variables occurring in the effect must either be universally quantified or must be parameters of the operator.*

[2] In this report, operator names are written with bold-face letters, and variable names are prefixed with a question mark. Keywords, which are prefixed with a colon, indicate the separate parts of an operator such as preconditions and effects.

Given $\mathcal{O}$, now possibly containing conditional effects, and D, the domain, one obtains the set of all *actions* O as the set of all possible ground instances of all operators in $\mathcal{O}$. Given an operator, it is generating the following set of actions: First, all possible type conforming instantiations of the declared parameters are systematically enumerated. For each instantiation, an action is generated in which all occurrences of variables are replaced by the corresponding constants in all parts of the operator. Universally quantified conditional effects are expanded into sets of ground conditional effects within each action.

Definition 9 *An* ADL action *o is a ground instance of an operator and has the form:*[3]

$$\begin{aligned} o :\ & \mathit{pre}_0 \\ & \phi_0 : \mathit{eff}_0 \\ & \phi_1 : \mathit{pre}_1 \Rightarrow \mathit{eff}_1 \\ & \vdots \\ & \phi_n : \mathit{pre}_n \Rightarrow \mathit{eff}_n \end{aligned}$$

The pre_i, eff_i *are sets of ground literals.* ϕ_0 *is denoting the preconditions of* o*,* eff_0 *are the unconditional effects, and* ϕ_1 *to* ϕ_n *are the conditional effects. The set of all effects* ϕ_0 *to* ϕ_n *of an action* o *is denoted with* $\Phi(o)$*.*

2 Planning with Planning Graphs

Planning with planning graphs [BF95] has received considerable attention because of the spectacular runtime behavior reported for the GRAPHPLAN system. While previous approaches to planning showed an explosion of runtime on very small planning problems, GRAPHPLAN scales to orders of magnitude beyond these, see Figure 1.

The underlying approach splits the planning process into two phases: A forward-search phase builds a compact data structure (a planning graph) that represents all states that are reachable from the initial state until the goals are achieved. The subsequent backward-search phase extracts a subgraph out of the planning graph that represents a valid plan.

The impressive performance and in particular the theoretical properties such as soundness, completeness, generation of shortest plans, and termination on unsolvable problems motivated the use of planning graphs as the foundation for the IPP planner.

But GRAPHPLAN also has its limitations. First, its performance can decrease dramatically if planning graphs become large, which can among others be caused by too much irrelevant information in the specification of a planning task, see [NDK97] for a detailed analysis. Second, its simple representation language is restricted to pure STRIPS operators – neither negation or equality, nor conditional or universally quantified effects are allowed and it was unclear whether the underlying planning algorithm could be extended to more expressive formalisms [BF97, GK97, McD96].

[3] The ground instance of a universally quantified effect $\forall x\ [\mathrm{pre}_i(x) \Rightarrow \mathrm{eff}_i(x)]$ is the conjunction of ground instances $\bigwedge_{k=0}^{n}[\mathrm{pre}_i([x/a_k]) \Rightarrow \mathrm{eff}_i([x/a_k])]$ if the domain $DOM(x)$ of the typed quantified variable x is $\{a_0, a_1, \ldots, a_n\}$. $[x/a_k]$ denotes the instantiation of the variable x with the object constant a_k.

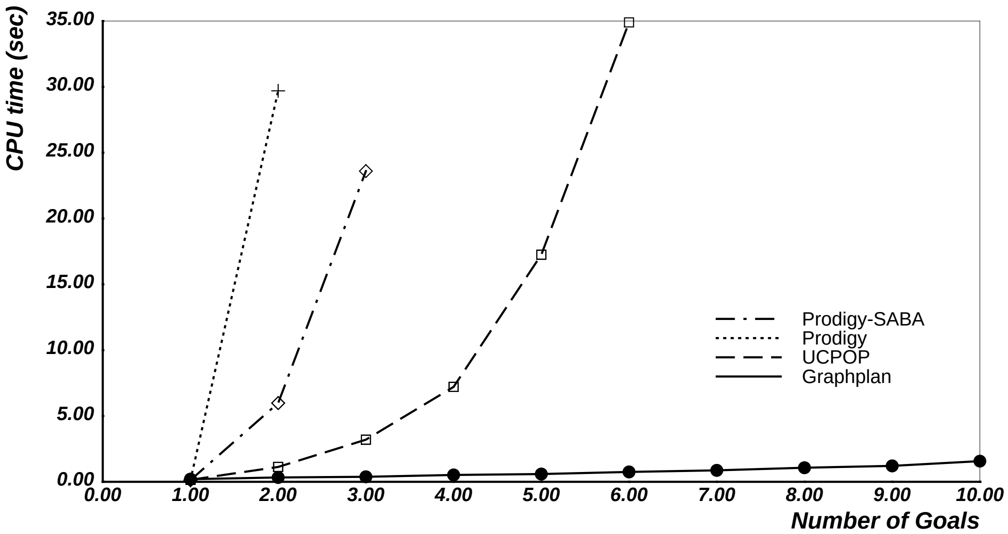

Figure 1: Spectacular improvement of scaling behavior in the GRAPHPLAN system. Cited with friendly permission from [BF95].

2.1 The Graphplan System

In the following, I review the main ideas of the GRAPHPLAN system and define the underlying concepts for standard STRIPS operators, in which the preconditions and effects are restricted to be sets of atomic formulas, i.e., no atomic negation is allowed. It is represented by introducing an additional predicate *not-p(x)* if $\neg p(x)$ is needed. To express that an atomic formula is made true or false by an action, so-called *add* and *delete* lists are distinguished in the effect representation. For example, instead of writing

stack(a,b): *clear(b), holding(a)*
$\Rightarrow$ *arm-empty, clear(a), on(a,b), $\neg$ clear(b), $\neg$ holding(a)*

one writes

stack(a,b): *clear(b), holding(a)*
$\Rightarrow$ *ADD arm-empty, clear(a), on(a,b) DEL clear(b), holding(a)*

and defines the predicates for the *add* and *delete* lists correspondingly. If an action is executed, all atoms in the add list are added to the state description and all atoms in the delete list are deleted from it in order to obtain the resulting state. The following definition captures the slightly different notion of actions in GRAPHPLAN.

Definition 10 *An* action *o in* GRAPHPLAN *is a ground instance of an operator and has the form*

$$o : pre_0$$
$$\alpha_0, \delta_0$$

where the preconditions pre_0, the add *list α_0, and the* delete *list δ_0 are restricted to sets of ground atoms.*

Definition 11 *A* planning graph $\Pi(N, E)$ *is a directed leveled graph with the set of nodes* $N = N_O \cup N_F$ *where* N_O *and* N_F *contain the sets of action and fact nodes, respectively. The set of edges* $E = E_P \cup E_A \cup E_D$ *is split into the three disjoint sets* E_P*, the precondition edges linking fact nodes to action nodes and* E_A*,* E_D *the effect edges linking action nodes to Add effects or Del effects, respectively.*

In a leveled graph, action and fact nodes are arranged in alternating levels and each level is associated with a time step (a natural number). The smallest level $N_F(0)$ comprises fact nodes (one per atom from the initial state I) and is followed by a level of action nodes $N_O(0)$, followed by $N_F(1), N_O(1), N_F(2), \ldots, N_F(max)$, i.e., the graph starts and ends with a fact level. Edges are restricted to link only nodes from two adjacent levels.

When building planning graphs, GRAPHPLAN starts with the initial facts to form the first fact layer in the graph, then adds an action node for each applicable action and draws the corresponding precondition, add, and delete edges. Each action level $N_O(n)$ contains two kinds of action nodes: so-called no-ops (one per atom in $N_F(n)$) and "ordinary" action nodes (one per action that is applicable in $N_F(n)$). No-ops are GRAPHPLAN's solution to the frame problem. For each fact $f \in N_F(n)$, a no-op nop_f is added to $N_O(n)$ with precondition $\mathrm{pre}_0(nop_f) = \{f\}$ and the only ADD effect $\alpha_0(nop_f) = \{f\}$. The planning graph construction terminates when all goal atoms occur in $N_F(max)$ or if the goals are unreachable, see Theorem 1 below.

One of the key insights of the GRAPHPLAN developers was the propagation of *exclusion* constraints (also called *mutex* relations) among action and fact nodes in the graph. To formally define mutually exclusiveness of actions, one begins with the notion of *interference.*

Definition 12 *Two actions* o_1 *and* o_2 interfere *iff* $\Big(\alpha_0(o_1) \cup pre_0(o_1)\Big) \cap \delta_0(o_2) \neq \emptyset$ *or* $\Big(\alpha_0(o_2) \cup pre_0(o_2)\Big) \cap \delta_0(o_1) \neq \emptyset$.

Interference constitutes the base case in the following recursive definition of the exclusiveness of actions. The general case requires to additionally define what it means for two actions to have *competing needs.*

Definition 13 *Two actions are* mutually exclusive *of each other*

- *at time step 0: iff they interfere;*
- *at time step* $n \geq 1$*: iff they interfere or the actions have competing needs.*

For simple STRIPS actions, being mutual exclusive means that there is no possible state of the world where both actions could be executed.

Definition 14 *Two actions* o_1 *and* o_2 *at an action level* $N_O(n \geq 1)$ *have* competing needs *iff there exist facts* $f_1 \in pre_0(o_1)$ *and* $f_2 \in pre_0(o_2)$ *that are mutually exclusive.*

Facts are mutually exclusive if no possible state of the world can make both true.

Definition 15 *Two facts* $f_1, f_2 \in N_F(n \geq 1)$ *are* mutually exclusive *iff there is no non-exclusive pair of actions* $o_1, o_2 \in N_O(n-1)$ *and no single action* $o \in N_O(n-1)$ *that adds* f_1 *and* f_2.

During graph construction, all action and fact nodes are tested for mutual exclusivity and the information is stored in the graph. Finally, one can define when an action must be added to the graph.

Definition 16 *An action o is* applicable in a fact level $N_F(n)$ *iff $pre_0(o) \subseteq N_F(n)$ and all $f \in pre_0(o)$ are non-exclusive of each other.*

Planning graphs possess several interesting properties. First, their size is polynomially restricted in their depths, the number of actions, and the number of facts in the initial state. Second, fact and action levels grow monotonically, i.e., if a fact or action is contained in a level n it will also be contained in all levels m with $m > n$, which follows immediately from the use of no-ops. While actions and fact levels monotonically increase, mutex relations can only monotonically decrease.

Lemma 1 *If two facts f_1 and f_2 are non-exclusive at a level $N_F(n)$ then they will remain non-exclusive at all future levels $N_F(m)$ with $m > n$.*

Proof: Two facts f_1 and f_2 are copied from a level to subsequent levels with the help of two corresponding no-ops. These no-ops are non-exclusive because their preconditions f_1 and f_2 are non-exclusive (assumption) and no-ops never interfere (Definition 12). Thus there is at least one way of achieving both facts with non-exclusive actions and therefore they remain non-exclusive. ■

The observation of monotonicity in planning graphs forms an important basis for an effective termination test.

Definition 17 *Let $\mathsf{Mutex}(n)$ be the set of all exclusive pairs at fact level n. A planning graph has* leveled off *at level n if and only if $|N_F(n)| = |N_F(n+1)|$ and $|\mathsf{Mutex}(n)| = |\mathsf{Mutex}(n+1)|$.*

Lemma 2 *Every planning graph levels off after a finite number of expansions.*

Proof: We observe first that any action has a finite number of literals as its effects.[4] Thus, the number of facts that can be achieved with a fixed set of actions is finite. Second, fact levels grow monotonically in the number of facts and mutual exclusion relationships can only decrease for identical fact levels. Because of that, each planning graph must converge to a fixpoint. ■

If the graph has leveled off at time step n then the set of applicable actions has reached a fixed point, i.e., $\forall\, m \geq n+1 : N_O(m) = N_O(n)$.

Theorem 1 *[Blum & Furst 95] A planning problem $\mathcal{P}(\mathcal{O}, D, I, G)$ has no solution if its planning graph has leveled off at time step n and either*

1. *one atomic goal is not contained in $N_F(n)$ or*
2. *at least two goal atoms are marked as mutually exclusive.*

[4]This will also remain valid when universally quantified effects are allowed in actions because of the finite domain assumption.

This theorem formulates a very efficient method to determine unsolvability in many cases. One grows the planning graph until the fixpoint is reached and then tests if all goals occur in the last fact level and are non-exclusive. If this is not the case, the planning problem must be unsolvable. Unfortunately, the test only provides a sufficient criterion. A necessary and sufficient test requires to perform search and to store unsolvable goal sets at each fact layer in the graph.

When GRAPHPLAN searches for a plan, it starts at the *max* level of the graph with the original goals as input. Then *add* edges are followed that connect to these goal nodes and that can yield a non-exclusive choice of actions. The preconditions of the selected actions form the new goals at the next time step and, of course, they must be non-exclusive to be solvable at all. If they are exclusive or the search algorithm cannot find a plan to achieve them, backtracking occurs to find a new choice of actions. The search terminates with a plan when the initial fact level is reached.

At each layer, the unsolvable goal sets G are memoized and can be used to

1. cut-off search when a superset of an unsolvable goal set is constructed

2. terminate on unsolvable planning problems.

Theorem 2 *[Blum & Furst 95] If the graph has leveled off at some level n and a stage t has passed in which $|G_n^{t-1}| = |G_n^t|$, then the planning problem is unsolvable.*

In [KNHD97] it has been shown that this test remains valid, even under the so-called *subset memoization* strategy. During subset memoization, the unsolvable goal sets at each fact layer get memoized in a separate data structure, see [HK99]. Before the planner starts searching for a set of actions to achieve a new goal set at a layer n, it checks if this set contains a previously generated goal set that has been memoized as unsolvable. In this case, no search is necessary as any superset of an unsolvable goal set must be unsolvable, too. Under the subset memoization strategy, these supersets of unsolvable goal sets do not get memoized, i.e., $|G_n^{t-1}|$ does not count all unsolvable goal sets. In the original GRAPHPLAN system, the termination test was therefore switched off when doing subset memoization. But as has been proven, this is an unnecessary measure of precaution as the termination test remains valid.

2.2 Other Graphplan Descendants

The GRAPHPLAN system has stimulated a whole bunch of research projects that are still ongoing. IPP was the first system that extended GRAPHPLAN's representation language to conditional effects and that was made available to the research community.

- The **UCPOP-GP Preprocessor** described in [GK97] was the first attempt of indirectly extending GRAPHPLAN to more expressive planning languages. It takes as input a UCPOP planning domain [PW92] and translates it into an equivalent set of GRAPHPLAN STRIPS operators and a fact file. Some of the preprocessing techniques are also used in IPP, cf. Section 1.

- **Blackbox** (Kautz/Selman 1998) uses GRAPHPLAN as a front end to generate an encoding of STRIPS planning problems in boolean formulas in conjunctive normal form [KS98]. Then a stochastic satisfiability testing algorithm is used to solve the planning problem [KS92, KS96].

- **SGP** (Anderson/Weld 1998) is an extension of GRAPHPLAN to the PDDL language that is very similar to IPP, but handles mutual exclusion relationships between conditional effects differently [AW98]. See Section 3 for a more detailed discussion.

- **STAN** (Fox/Long 1998) is a very efficient implementation [LF98b] of GRAPHPLAN that combines planning graphs with static analysis techniques to reduce the search space [FL97, LF98a].

- **BSR-Graphplan** is a kind of inverted GRAPHPLAN that also handles conditional effects. The system builds the graph backwards from the goals to the initial state and then searches forward to extract a plan [KLP97]. A more detailed evaluation of this approach is not possible because the system has not been released yet.

The source code of Blackbox, SGP, and STAN is available. These three systems also competed in the 1998 planning systems competition.

Besides system development, several research projects focus on the exploration of how planning graphs relate to other planning approaches. A first investigation of local search techniques for planning graphs is described in [SG98], while [Lot98] investigates how planning graphs can be used to speed up hierarchical task network planning. Both approaches have been implemented using IPP as the underlying planning system.

2.3 Implementational Details in IPP

GRAPHPLAN's code implements the idea of planning graphs in a straightforward way and explicitly built the graph. This means, nodes for the initial state are generated, all applicable actions are stored, and the next level is initialized by copying the previous level. Obviously, such an implementation is time- and in particular very memory-consuming.

Let us consider the following very simple example to illustrate how planning graphs can be efficiently implemented. The example describes a small office delivery scenario where a robot has to deliver objects from origins to destinations in various locations. The IPP *fact file* contains the following declaration of objects, the initial state, and the goals:

type/object declaration:	*loc:office1, office2; object:letter*
initial state:	*origin(letter, office1) in(office1) dest(letter, office2)*
goal state:	*delivered(letter)*

This states the problem to deliver one letter from *office1*, where the robot and the letter are, to *office2*. The IPP *ops file* contains the following three planning operators:

go*(?loc1, ?loc2:loc): in(?loc1)*
$\Rightarrow$ *in(?loc2),* $\neg$ *in(?loc1)*

drop*(?b:object, ?loc:loc): have(?b), dest(?b, ?loc), in(?loc)*
$\Rightarrow$ *delivered(?b)* $\neg$ *have(?b)*

get*(?b:object, ?loc:loc): origin(?b, ?loc), in(?loc)*
$\Rightarrow$ *have(?b)*

After parsing the two input files, the system instantiates the operators to obtain the set of possible actions. As a first step, the *inertia* are removed, i.e., all those facts from the initial state that are never made true or false by an action. They can be eliminated from all preconditions and effect conditions in the actions and from the conditions of conditional goals. This can lead to identical effect conditions of different conditional effects or make a conditional effect unconditional if the effect condition only comprises inertia. In such a case, IPP reorganizes the effects of the action, i.e., conditional effects with empty effect conditions are merged into the unconditional effect and conditional effects with the same effect condition are merged into a single conditional effect. Conditional effects with unsatisfiable inertia conditions disappear completely. This can lead to a much more compact representation of actions and also allows the system to deduce more interference relationships between actions if conditional effects become unconditional, which in turn can reduce the search space.

In the example, neither the origin nor the destination of a letter can be changed, i.e., the two facts *origin(letter, office1)* and *dest(letter, office2)* are removed from the state representations and the action descriptions. Furthermore, only those actions are generated that satisfy inertia, i.e., no **get(letter, office2)** action results from the instantiation process because no origin of an object in office2 has been specified. Similarly, for the **drop(letter, office1)** action. The result is the following set of actions

get(letter, office1): *in(office1)*
$\Rightarrow$ *have(letter)*

drop(letter, office2): *have(letter), in(office2)*
$\Rightarrow$ *delivered(letter),* $\neg$ *have(letter)*

go(office2, office1): *in(office2)*
$\Rightarrow$ *in(office1),* $\neg$ *in(office2)*

go(office1, office2): *in(office1)*
$\Rightarrow$ *in(office2),* $\neg$ *in(office1)*

In the GRAPHPLAN system, the planning graph is explicitly built. This means, the first fact layer contained the initial facts, then all applicable actions plus no-ops for the initial facts built the first action layer. Their effects built the second fact layer. Each fact is represented by a string and a node in the graph structure. Precondition links connected facts to actions, and *add* and *delete links* connected actions to their positive and negative effects. This implementation was preserved in IPP until version 3.3 of the system and is shown in Figure 2.[5]

The plan that solves the delivery task comprises the three totally-ordered actions **get(letter, office1)**, **go(office1, office2)**, and **drop(letter, office2)**.

The explicit copying of fact and action layers caused significant memory consumption in GRAPHPLAN and IPP, though IPP made several improvements to store exclusivity information more effectively and to exclude inertia from the graph.

[5] Negation of the facts is not shown, but instead the facts are linked by a *delete edge* (dashed lines) as in GRAPHPLAN.

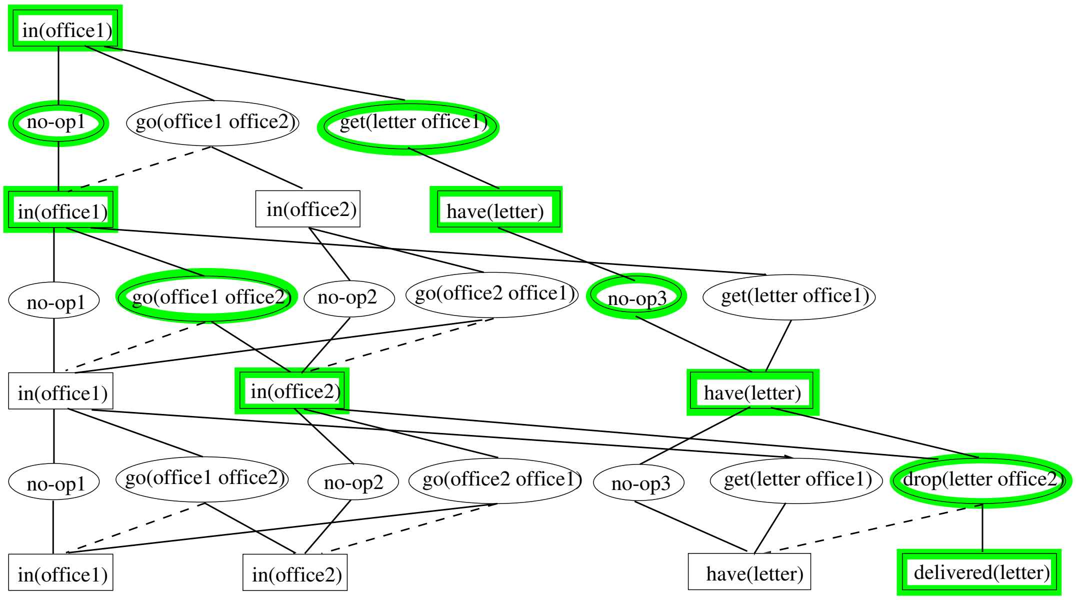

Figure 2: Implementation of a planning graph until IPP 3.3. Note the monotonic growth of the layers and the repeating patterns in the link structure between the single layers. A valid plan is a subgraph of the planning graph shown in light gray. No-ops that are selected to guarantee the validity of preconditions for other selected actions are also marked, but not listed when printing the plan.

A more compact representation was developed starting with IPP 4.0. A first major step encoded actions using bitvectors.[6] The basis for this encoding is to explicitly determine the set P of all atoms, which can be easily done by analyzing actions and state descriptions. Then two bitvectors are created in which each position corresponds to one atom $p \in P$. One bitvector encodes the positive occurrences of p and the other encodes the negative occurrences of p. Consequently, a state is represented by a pair of such bitvectors as in Figure 3. Again, inertia can be removed from the set P.

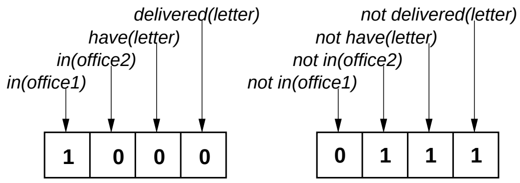

Figure 3: Representation of the example's initial state under CWA where a "1" means the atom is true, and a "0" means the atom is false. Technically, all truth values of positive literals that are listed in the initial state are set to "1" when parsing the state description. Then CWA sets all remaining positive literals to "0". The negative literals obtain the opposite value of the positive literal. One can then easily test if the initial state representation is consistent. The inertia *origin(letter, office1)* and *dest(letter, office2)* are not encoded.

Depending on whether a planner makes the closed-world assumption (CWA), two

[6] This encoding was proposed by Jürgen Eckerle in an experimental iterative-deepening STRIPS planning algorithm [Eck98] and a similar encoding is used in the STAN planner [LF98b]. For IPP, the encoding was extended to support explicit negation.

different computations need to be performed in order to obtain the initial state encoding. Under CWA, only those facts need to be stated that are known to be true, all other facts are automatically inferred to be false. This implies, that for each atom its exact truth value can be determined and no unknown values can occur. If CWA is not made, only those atoms are known to be true or false that are explicitly listed in the initial state, for all other atoms the truth values are unknown. In IPP, CWA is made because it is very common in planning systems, saves writing effort because only the true facts need to be stated, and it yields a unique model for the initial state in which each atom is either true or false. This still allows to say that $\neg p$ is true, but one can also simply say nothing about p and the truth of $\neg p$ will be inferred automatically. The explicit representation of negative and positive occurrences of atoms in the initial state is in fact redundant under CWA, but has been chosen to allow easier extension to non-CWA if necessary in the future and it also makes it easier to match action effects and preconditions to states and to test for consistency.

Since actions are nothing else than state transitions, they can now be represented by several of these bitvector pairs, see Figure 4.[7]

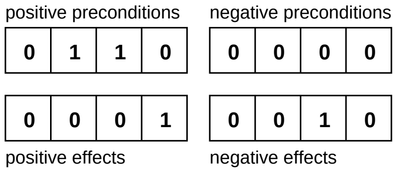

Figure 4: Representation of the **drop(letter, office2)** action where a separate bitvector pair is used to encode preconditions and effects. In the effect representation, a "1" means the action explicitly sets the atom as true, "0" means the atom remains unchanged. In the precondition representation, "1" means the atom must be true for the action to be applicable, "0" means the truth value of the atom is not of importance.

Accordingly, one can now also represent the planning graph based on bitvector pairs because each fact layer represents a set of states $S \subseteq 2^P$. Going even further, the planning graph is no longer explicitly expanded, but just one layer of facts and actions is shown with a marker that tells at which layer the fact or action appears, which becomes possible because of the monotonic growth of the layers.[8] One could also encode each action(name) by a single bit in the graph, i.e., keep a bitvector for the action layer, but this will not further reduce memory consumption because the action names must be kept somewhere in the system anyway. Since exclusivity relations between pairs of actions or facts can only disappear, it is sufficient to store these exclusive pairs once together with a marker that tells at which level they disappear or if they hold infinitely.[9] Figure 5 shows the compact encoding of the planning graph for the delivery example without exclusivity information. To extract the GRAPHPLAN fact layers, one just needs to consider all facts whose marker is lower or equal the time step number of the desired fact layer. The graph has leveled off

[7] If an action contains a conditional effect, then the effect condition and the effect itself are encoded by two pairs of bitvectors. An action with several conditional effects has therefore several bitvector pairs as its effect representation.

[8] The idea of an implicit representation of planning graphs using only one action and fact layer was devised by David Smith and Daniel Weld at the AIPS-98 Workshop on Combinatorial Search.

[9] The implementation of exclusivity information in IPP 4.0 extends the idea of spikes [FL98].

if no new markers can be set in action or fact nodes and the number of mutex relations remains constant.

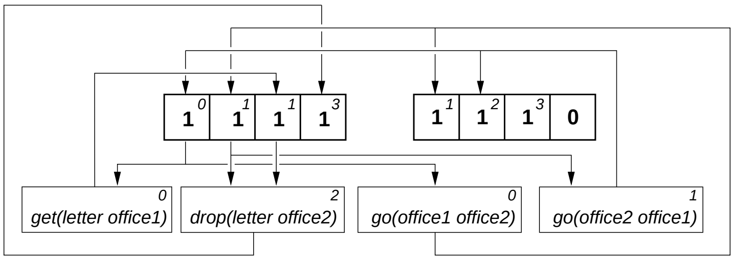

Figure 5: Compact representation of the planning graph in IPP 4.0.

Based on this encoding, many tests that have to be performed very frequently during planning become simple bitvector operations. In the following definitions, pv denotes the bitvector of positive facts and nv denotes the bitvector of negative facts.

Definition 18 *A state S is inconsistent iff* $pv(S)$ AND $nv(S) > 0$.

Definition 19 *An action o is applicable in a state S iff* $pv(pre_0(o))$ AND $pv(S) = pv(pre_0(o))$ *and* $nv(pre_0(o))$ AND $nv(S) = nv(pre_0(o))$ *and all precondition literals are pairwise exclusive of each other.*

Definition 20 *Two actions o_i and o_j interfere with each other iff*

$$\big(nv(pre_0(o_i)) \text{ OR } nv(\mathit{eff}_0(o_i))\big) \text{ AND } \big(pv(pre_0(o_j)) \text{ OR } pv(\mathit{eff}_0(o_j))\big) \neq 0$$

or

$$\big(nv(pre_0(o_j)) \text{ OR } nv(\mathit{eff}_0(o_j))\big) \text{ AND } \big(pv(pre_0(o_i)) \text{ OR } pv(\mathit{eff}_0(o_i))\big) \neq 0$$

hold.

Intuitively, two actions interfere if they assign different truth values to the same atom or have inconsistent preconditions. In the above representation, this means that a “1” is contained in the same bitfield in the positive and negative bitvector encoding the preconditions and unconditional effects. This means, trying to compute the logical AND over the two bitvectors must return a value different from 0 in this case.

3 Going Beyond STRIPS: ADL

Although all ADL language features can be handled by a preprocessing approach that translates expressive operators into simple STRIPS operators [GK97], a “transformational solution” remains unsatisfying in the case of universally quantified conditional effects. To illustrate the problems, let us consider the *briefcase* domain containing a **move** operator with such a more complicated effect. This operator specifies that all objects which are inside a briefcase move whenever the briefcase moves. As a precondition, the briefcase must be at a specific location $?l_1$ expressed with the *at-b* predicate. As an effect, the briefcase will be at the new location l_2 and no longer at l_1. The universally quantified conditional effect specifies what happens to all objects $?o$ for which $in(?o)$ is true in the state in which **move** will be executed.

move(l_1, l_2:*location*)
:precondition *at-b*(l_1)
:effect *at-b*(l_2), $\neg$ *at-b*(l_1)
$\forall x$:*object in*(x) $\Rightarrow$ *at*(x,l_2), $\neg$ *at*(x,l_1)

In this representation, the exact location of each object in each state of the world can be directly read off the state representation because it is updated whenever the **move** operator is applied to a state. In principle, sets of STRIPS operators can be used to encode such conditional effects. The example operator can be equivalently translated into a set of operators—one operator for each possible subset of objects, i.e., moving the empty briefcase, moving the briefcase with one object inside, with two etc. But such an encoding leads to exponentially more actions, which can make even small planning problems practically intractable. More precisely, instantiation of the ADL **move** operator as given above will yield $|loc| \times (|loc| - 1)$ actions, where $|loc|$ is the number of different locations. The translation of this operator into STRIPS actions has to consider all subsets of objects, i.e., $(|loc| \times (|loc| - 1)) \times 2^{|obj|}$ actions result. No better, i.e., more compact, encoding is possible [Neb98].

An example is shown in Figure 6 in which GRAPHPLAN had to solve a simple roundtrip problem in the briefcase domain where an increasing number of objects at different locations has to be brought to a new destination. The translation leads to a strongly increasing number of operators (because all possible subsets of objects have to be considered for transportation), which in turn generate lots of actions. Runtime jumps from marginal in the case of 3 objects to almost four and a half hours for 5 objects.

These observations motivated a direct embedding of operators with conditional and universally quantified effects into planning graphs, while other features of ADL such as existential quantification, domain axioms, or disjunctive preconditions are handled successfully in IPP using the preprocessing approach described in [GK97], see Section 2.

#objects	#locations	#operators	#actions	plan length	cpu time in s
1	2	4	12	3	0.02
2	3	6	48	5	0.11
3	4	10	152	7	1.14
4	5	18	440	9	173.48
5	6	34	1212	11	15656.58

Figure 6: Dramatic performance decrease of GRAPHPLAN on briefcase roundtrip problems caused by the translation of conditional effects into sets of STRIPS operators.

3.1 A Semantics for Parallel ADL Plans

One of the distinguishing features of GRAPHPLAN is its ability to produce *shortest* plans in the sense that it exploits maximal parallelism of actions in the plan. For standard STRIPS operators, it is relatively easy to define when two actions can be executed in parallel and what the result of the execution is, cf. [BF97]. In the case of actions with universally quantified and conditional effects, several semantics are possible that define the result of applying a parallel action set to a state in different ways. Two of them are discussed in the following.

Let us consider the following ground instance of the **move** operator in a problem with the two possible objects *letter, toy*, and the two possible locations *office, home* yielding the following action with one set of unconditional effects and two different conditional effects:

move*(office, home)*
:precondition *at-b(office)*
:effect *at-b(home)*, ¬ *at-b(office)*
in(letter) ⇒ *at(letter, home)*, ¬ *at(letter, office)*
in(toy) ⇒ *at(toy, home)*, ¬ *at(toy, office)*

The result of applying a single action to a given state can be easily defined. If the action is applicable, all action's effects whose effect condition hold in the state are added to the state description and all atoms that have been reverted by the action's effects are removed from it in order to maintain a consistent state. Trying to apply an action which is not applicable leads to an undefined state.[10]

Definition 21 *Let O be the set of ADL actions with conditional effects, i.e., all ground instances of the operators in $\mathcal{O}$ given in the normalform as defined in Definition 9*

$$\begin{aligned} o\ :\ & pre_0 \\ & \phi_0 : \mathit{eff}_0 \\ & \phi_1 : pre_1 \Rightarrow \mathit{eff}_1 \\ & \vdots \\ & \phi_n : pre_n \Rightarrow \mathit{eff}_n \end{aligned}$$

O^* be all sequences over O, and Res be a function from states and sequences of actions to states.

$$Res : 2^P \times O^* \longrightarrow 2^P$$

The result of applying a sequence containing the single action o to a state S_P is defined as

$$Res(S_P, \langle o \rangle) = \begin{cases} \big(S_P \cup \Phi(S_P, o)\big) \setminus \bar{\Phi}(S_P, o) & \text{if } S_P \models pre_0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

with

$$\Phi(S_P, o) = \bigcup_{S_P \models pre_i, i \geq 0} \mathit{eff}_i \qquad \text{and} \qquad \bar{\Phi}(S_P, o) = \{\neg p \mid p \in \Phi(S_P, o)\}$$

The result of applying a sequence of more than one action to a state is recursively defined as

$$Res(S_P, \langle o_1, \ldots, o_n \rangle) = Res(Res(S_P, \langle o_1, \ldots, o_{n-1} \rangle), o_n).$$

[10] An alternative definition could be that the state remains the same.

For a planning language of actions with only unconditional effects, the definition of *Res* can be extended to a set of parallel actions in a straightforward way such that the resulting state is uniquely defined [BF97]. However, when conditional effects are allowed, it is very difficult and even very restrictive to guarantee the uniqueness property of *Res*. In order to guarantee that a unique state results from the execution of a parallel set of ADL actions, one could come up with the following definition $Res^\star$:

Definition 22 *Let $Res^\star$ be a function from states and sequences of sets of actions to states*

$$Res^\star : 2^P \times (2^O)^* \longrightarrow 2^P$$

The result of applying the empty sequence of actions to a state is defined as $Res^\star(S_P, \langle\rangle) = S_P$. In the case of a sequence containing one non-empty set we obtain

$$Res^\star(S_P, \langle\{o_1, \ldots, o_n\}\rangle) = \begin{cases} \left(S_P \cup \bigcup_{i=1}^n \Phi(S_P, o_i)\right) \setminus \bigcup_{i=1}^n \bar{\Phi}(S_P, o_i) & \textit{if (1) to (4) hold} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

(1) $\forall\, o_i : S_P \models pre_0(o_i)$

(2) $\forall\, o_i, o_j, i \neq j : \Phi(S_P, o_i) \cup \Phi(S_P, o_j) \not\models \bot.$

(3) $\forall\, o_k, o_j, k \neq j \;\forall\, pre_i(o_k)$ *with* $S_P \models pre_i(o_k) \;: Res(S_P, o_j) \models pre_i(o_k)$

(4) $\forall\, o_k, o_j, k \neq j \;\forall pre_i(o_k)$ *with* $S_P \not\models pre_i(o_k) \;: Res(S_P, o_j) \not\models pre_i(o_k)$

The recursive case of a sequence $Q = \langle Q_1, \ldots, Q_n\rangle$ is defined as

$$Res^\star(S_P, Q) = \begin{cases} Res^\star(Res^\star(S_P, \langle Q_1, \ldots, Q_{n-1}\rangle), \langle Q_n\rangle) & \textit{if } Res^\star(S_P, \langle Q_1, \ldots, Q_{n-1}\rangle) \\ & \textit{is defined} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Condition (1) requires that all preconditions of all actions in the set need to be satisfied in the state S_P. Condition (2) makes sure that all actions are independent of each other in the sense that for each pair of actions the unconditional and the conditional effects with satisfied effect conditions are consistent. Condition (3) requires that no precondition or effect condition that was valid in the state S_P is made invalid by an unconditional or conditional effect, while Condition (4) requires that no effect condition that was invalid in the state S_P is made valid by the effects of an action.

On one hand, this definition of $Res^\star$ leads to a unique resulting state similarly to what one obtains for simple STRIPS operators. On the other hand, this definition is far too restrictive and unnecessarily complicates the planning process when trying to satisfy conditions (1) to (4) during action selection. In particular, to satisfy (3) and (4) one has to proceed over all effect conditions and decide if it has to be made true or false. As a simple example for the restrictiveness of this semantics, consider the two actions with empty preconditions and one conditional effect:

$$\mathbf{o_1} : \top \qquad\qquad \mathbf{o_2} : \top$$
$$a \Rightarrow y \;,\; \neg c \qquad\qquad b \Rightarrow x \;,\; c$$

together with the goal $\{x, y\}$ and the state $S_P = \{a, b\}$. Trying to apply the action set to this state leads to an undefined state because c and $\neg c$ are both made true. But since we are not interested in the value of c (it is not a goal), we would rather be satisfied with obtaining a set of states as the result, one in which c holds and another one in which $\neg c$ holds. In fact, GRAPHPLAN also is too restrictive when defining interference of actions because the same phenomenon can already be observed for simple STRIPS actions [Kno94]. In the example by Knoblock, one action paints a table and the floor in red color. A second action paints a chair and the floor in blue color. The goal is to have a red table and a blue chair, but the color of the floor is irrelevant. GRAPHPLAN would mark **paint(blue)** and **paint(red)** as exclusive because the two actions interfere since one destroys the floor color the other has created. Thus, it would never consider them for parallel execution, i.e., the system only generates shortest parallel plans wrt. its own definition of interference. However, since the color of the floor is irrelevant, both actions are independent relative to the current goals. They can be executed in parallel, but a different state would result from each linearization. The only solution to this problem one can imagine, is to come up with a goal-dependent definition of interference, but it would change the basic planning algorithm significantly because mutual exclusion relationships between facts and actions had to be computed during search as they would now depend on the goal sets. Interference because of conflicting effects would become goal-dependent. It is not really clear if this effort is justified.

But for conditional effects, it is relatively easy to allow multiple states as the result of applying a parallel set of ADL actions. Being less restrictive wrt. the resulting states also avoids the problem that the planning algorithm has to decide for each effect condition if it has to be made true or false. This can be accomplished by the following definition:

Definition 23 *Let $\mathcal{S}_\mathcal{P}$ be a set of states and R be a function from sets of states and sequences of sets of actions to sets of states:*

$$R : 2^{2^P} \times (2^O)^* \longrightarrow 2^{2^P}$$

The result of applying the empty sequence is defined as

$$R(\mathcal{S}_\mathcal{P}, \langle\rangle) = \mathcal{S}_\mathcal{P}$$

For the sequence containing exactly one set set of parallel actions $Q = \{q_1, \ldots, q_n\}$ we define

$$R(\mathcal{S}_\mathcal{P}, \langle Q\rangle) = \begin{cases} \{T \in 2^P \mid S_P \in \mathcal{S}_\mathcal{P}, q \in Seq(Q), T = Res(S_P, q)\} & \textit{if } Res(S_P, q) \textit{ is defined} \\ & \forall\, S_P \in \mathcal{S}_\mathcal{P}, q \in Seq(Q) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

with $Seq(Q)$ denoting the set of all linearizations of the action set $Q = \{q_1, \ldots, q_n\}$. The result of applying a sequence of several sets of parallel actions is defined as

$$R(\mathcal{S}_\mathcal{P}, \langle Q_1, \ldots, Q_n\rangle) = \begin{cases} R(R(\mathcal{S}_\mathcal{P}, \langle Q_1, \ldots, Q_{n-1}\rangle), \langle Q_n\rangle) & \textit{if } R(\mathcal{S}_\mathcal{P}, \langle Q_1, \ldots, Q_{n-1}\rangle) \\ & \textit{is defined} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Definition 23 will be the formal basis to prove the planner sound and complete in Section 3.6.

3.2 Interference of Conditional Effects

Considering the normal form representation for actions as defined in Definition 9, a slight change needs to be made in the definition of interference as it is used in IPP because explicit negation is now allowed, but *delete* effects do no longer exist as in GRAPHPLAN.

Definition 24 *Two actions o_i, o_j interfere iff their unconditional effects and/or preconditions are inconsistent, i.e.,* $\mathit{eff}_0(o_i) \cup \mathit{eff}_0(o_j) \models \perp$ *or* $\mathit{eff}_0(o_i) \cup \mathit{pre}_0(o_j) \models \perp$ *or* $\mathit{eff}_0(o_j) \cup \mathit{pre}_0(o_i) \models \perp$ *or* $\mathit{pre}_0(o_i) \cup \mathit{pre}_0(o_j) \models \perp$.[11]

This definition of interference ignores conditional effects because whether two actions interfere based on their conditional effects cannot be decided in advance, but depends on the specific state in which the actions are to be executed.

The definition of mutual exclusion remains unchanged. For simple STRIPS actions, being mutual exclusive means that there is no possible state of the world where both actions could be executed. Under conditional effects, the two notions do not exactly coincide anymore. If two actions are mutually exclusive then there is no state in which both actions could be executed. The opposite, though, does not need to hold. Actions can be non-exclusive, but their conditional effects can lead to unresolvable conflicts.

Based on the previous definitions, IPP is *optimistic* when building planning graphs, i.e., analysis and resolution of conflicts between actions caused by conditional effects are deferred to the planning phase. A more pessimistic approach with a different notion of interference was also explored, but did only seem to complicate the algorithms and was therefore abandoned in a very early stage of the project.

Although actions do not become exclusive if potential conflicts between their conditional effects occur, one can sometimes propagate exclusivity information from preconditions and effect conditions to conditional effects.

Definition 25 *Two sets of facts F_1, F_2 are mutually exclusive (written* $\mathsf{mutex}(F_1, F_2)$*) iff they contain facts $f_1 \in F_1$ and $f_2 \in F_2$ such that f_1 and f_2 are mutually exclusive in the sense of Definition 15.*

Definition 26 *Given a planning graph, two facts f_1 and f_2 are mutually exclusive at fact level $N_F(n+1)$ if for all possible pairs of actions (o_1, o_2) in level $N_O(n)$ with $f_1 \in \mathit{eff}_i(o_1)$ and $f_2 \in \mathit{eff}_j(o_2)$ (with $i, j \geq 1$) one of the following conditions holds*

1. *o_1 and o_2 are mutually exclusive according to Definition 13 (including the case that the actions have competing needs, i.e.,* $\mathsf{mutex}(\mathit{pre}_0(o_1), \mathit{pre}_0(o_2))$ *holds).*
2. $\mathsf{mutex}(\mathit{pre}_i(o_1), \mathit{pre}_j(o_2))$*, i.e., the conditional effects have competing effect conditions.*[12]

[11] Testing $\mathrm{pre}_0(o_i) \cup \mathrm{pre}_0(o_j) \models \perp$ is more efficient in an implementation because the two actions are exclusive because of inconsistent preconditions, which is a state-independent property and needs to be tested only once. Alternatively, the two actions are exclusive because of competing needs, but this is a state-dependent property and needs to be tested at each time step of the graph.

[12] This property was independently discovered in [KLP97].

3. $\mathsf{mutex}(pre_i(o_1), pre_0(o_2))$ *or* $\mathsf{mutex}(pre_0(o_1), pre_j(o_2))$, *i.e., the effect conditions of one action compete with the precondition of the other.*[13]

Notice that the actions $o1$ *and* $o2$ *are not marked as exclusive.*

One can easily see that such a restricted propagation of exclusiveness relations over conditional effects does not exclude any solutions from the graph. In case 1, two conditional effects are exclusive because they can only be achieved with exclusive actions. For Case 2, note that a pair of mutually exclusive effect conditions means that there is no state in which both effect conditions can hold simultaneously, i.e., their corresponding effects cannot both be made true when applying the action to a state. For Case 3, the argumentation is similar, but here effect conditions of one action and preconditions of the other action cannot be made true simultaneously.

The above definition allows the planner to propagate exclusivity relations over fact nodes, but there is no way in IPP for propagating exclusivity relations over action nodes based on an analysis of the interference between conditional effects. This has led to some criticism in [AHEK94] where such a propagation of mutex relations over actions is proposed.

The approach in the SGP system described in [AHEK94] relies on the so-called "factored expansion" of actions. It means that each action is split into its separate conditional effects that are having the action's preconditions and their own effect conditions as effects. All facts are made conditional, i.e., unconditional effects have an empty effect condition. This is almost identical to the representation of effects in IPP where each effect points to its effect condition, while the unconditional effect points to the empty effect condition. However, actions in IPP are still treated as atomic units, while SGP creates "components" similar to GRAPHPLAN's STRIPS actions that represent each conditional effect in the graph. This opens the possibility to mark component nodes as exclusive, which is claimed to lead to much better performance of a planner.[14]

A careful analysis of the SGP *induced mutex relations* that are calculated between components reveals that quite often they do not seem to lead to any search space reduction when compared to the approach that IPP is using. Let us consider the following example taken from [AHEK94] with the two actions

$$\begin{array}{llcll} \mathbf{o_1}: & \top & \qquad\qquad & \mathbf{o_2}: & t \\ & p \Rightarrow x & & & \top \Rightarrow \neg x \\ & p, q \Rightarrow e\ ,\ f & & & \end{array}$$

Figure 7 shows the different representations in both planners. IPP keeps a single action node o_1 that has three facts as conditional effects, but no precondition. The effect nodes point to their effect conditions. SGP splits o_1 into two component nodes o_{1a} and o_{1b}. Each of them has preconditions and unconditional effects and can therefore be considered as a STRIPS action.

In IPP, the actions are non-exclusive because only a conditional effect of o_1 interferes with o_2. Consequently, the planner would consider them as a valid choice to reach a goal set.

[13] When adopting a more strict semantics such as the one in Definition 22, one can also mark two effects as exclusive if for all possible ways of achieving these effects, it holds that the actions have inconsistent conditional or unconditional effects.

[14] A comparison of SGP and IPP in the planning competition 1998 showed a different picture.

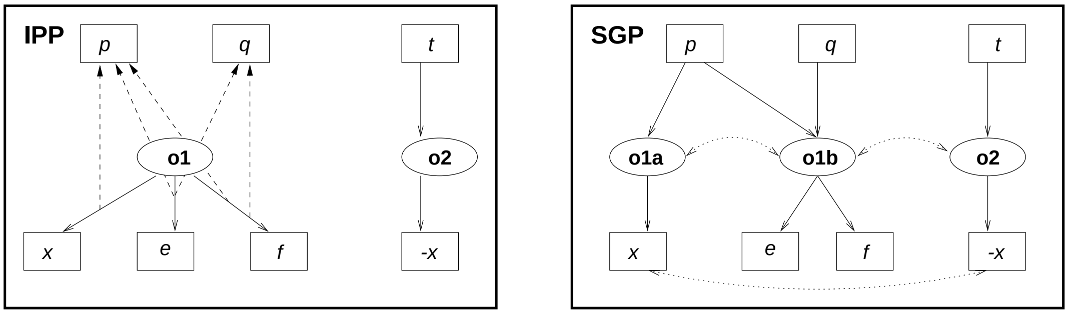

Figure 7: Representation of conditional effects in IPP compared to SGP.

In SGP, the components o_{1a} and o_2 are mutually exclusive because of their inconsistent effects, which are no longer conditional. In order to be able to propagate mutex relations over components, SGP tests whether one component *induces* another component. In the example, both components o_{1a} and o_{1b} *induce* each other: The precondition of o_{1b} implies the precondition of o_{1a}, therefore o_{1b} induces o_{1a}. The precondition q of o_{1b} cannot be prevented in the graph because $\neg q$ is not contained in the fact level, therefore o_{1a} induces o_{1b}. This means, whenever one component is selected to achieve a goal, the other must be executed as well. Because of the *induced* relation between the two components, o_2 is also made exclusive to o_{1b}, i.e., an *induced mutex relation* is additionally established by SGP.

Now let us assume that the goals are $\neg x, e, f$. SGP fails immediately because no non-exclusive set of components can be found to achieve the goals. IPP selects the two action nodes and constructs the new goal set t, p, q from the preconditions and the effect conditions of selected conditional effects. Then it checks for conflicts of conditional effects and finds that o_1 has a conditional effect x that is inconsistent with the goal $\neg x$ and therefore this effect has to be prevented in the state where the action is executed. The only way of doing this, is to add $\neg p$ to the goal set,i.e., to do the *confrontation* of the corresponding effect condition.[15] But this makes the goal set inconsistent and therefore IPP fails on this selection before it enters any search at all. The *movie* domain from the planning competition has been designed to exhibit the property of induced mutex relations. However, in this domain IPP 3.3 was able to detect that the effect condition of the single conditional effect that is contained in the actions in this domain comprises inertia only. Consequently, the inertia are removed and the conditional effect becomes unconditional. Thus, IPP derives additional mutex information and outperformed SGP in the competition. A joint discussion with Weld and Anderson led to a revised *movie* domain in which no inertia can be removed by IPP. With this revised action set, IPP is quite slow because almost no mutex relations can be derived and subset memoization fails completely in reducing search. However, using RIFO as a preprocessor to filter out all irrelevant information, IPP outperforms SGP again.

It is an open question which of the techniques is better than the other one. In general one can say that induced mutex relations seem to be very rare, e.g., there is no other domain in all benchmark sets except *movie* where these relations seem to occur or offer an important advantage. Besides this, the movie domain has been solely designed with the purpose to prove the usefulness of SGP's action component representation and does not appear to be very natural. On the other hand, removal of inertia and irrelevants seem to be more natural techniques and both phenomena occur quite frequently in planning

[15] IPP's planning algorithm is described in detail in Section 3.4.

domains.

3.3 Conditional Effects in Planning Graphs

The algorithm to construct planning graphs for a given planning problem in the ADL subset of operators differs not very much from the original algorithm described in [BF97]. The following description does not make use of the bit representation, but speaks more general and implementation-independent of nodes and edges. The only differences affect

- the representation of effect conditions in conditional effects,
- which conditional effects have to be added to the planning graph,
- the absence of delete edges because no *delete* effects exist anymore in actions.

Again, two kinds of nodes are distinguished to represent facts and actions. Precondition edges connect fact nodes and action nodes, and effect edges (instead of separate *add* and *delete* edges) connect action nodes to their effects. Effect edges are now written as triples $N_O \times N_F \times 2^{N_F}$ where an edge is drawn between an action node $o \in N_O$ and its effect (a fact node) $f \in N_F$ that is labeled with a set of fact nodes $F' \subseteq 2^{N_F}$, which represents the (conjunctive) effect condition under which o achieves f. Figure 7 above showed IPP's representation of conditional effects with pointers from effect nodes to precondition nodes that represent the effect conditions.

Technically, planning graphs are constructed in the following way: Given a planning problem, the set of actions is determined as all possible ground instances of all operators. The facts from the initial state form the first fact level $N_F(0)$. As in GRAPHPLAN, each action level $N_O(n)$ contains two kinds of action nodes: so-called no-ops (one per fact in $N_F(n)$) and "ordinary" action nodes (one per action that is applicable in $N_F(n)$). For each action node o, precondition edges between each fact in $N_F(n)$ that is a precondition of o and the action node are established. IPP planning graphs differ from GRAPHPLAN only wrt. the effect edges and how the next fact level $N_F(n+1)$ is built. Given a conditional effect $\text{pre}_i(o) \Rightarrow \text{eff}_i(o)$, IPP proceeds over the individual facts $f \in \text{eff}_i(o)$. A fact is added to $N_F(i+1)$ iff the following conditions are satisfied:[16]

1. $\text{pre}_i(o) \subseteq N_F(n)$
2. all facts in $\text{pre}_i(o)$ are non-exclusive of each other in $N_F(n)$
3. all facts in $\text{pre}_i(o)$ are non-exclusive of the facts in $\text{pre}_0(o)$ in $N_F(n)$

The first two conditions imply that the effect is applicable, similarly to the notion of action applicability as defined in Definition 16. Condition 3 tests if precondition and effect condition can possibly hold in the same state. Only if all three conditions are satisfied, the effect f can possibly be made true in the next fact level. If f is not already contained in $N_F(n+1)$, a new fact node is generated.[17] Otherwise, only an effect edge is established that is augmented with a pointer to all facts in $N_F(n)$ that occur in $\text{pre}_i(o)$. Since one

[16] Note that they are tested only once for the conditional effect.

[17] Under the implicit graph representation using bitvectors, there is already a bitfield reserved for each fact and only the marker of a fact is set when it occurs for the first time as an action effect.

action can achieve the same atomic effect f under different effect conditions, more than one edge can exist between an action node and a fact node in the next level.

Now the planner has to enter a fixpoint computation to check if conditional effects that have not been added to the current action level could possibly be enabled by another action in the same level. To illustrate this problem, let us consider the following example actions:

$$\begin{array}{llcll} \mathbf{o_1}: & p & \qquad\qquad & \mathbf{o_2}: & \top \\ & q \Rightarrow \neg x & & & \top \Rightarrow e \\ & r \Rightarrow e & & & \neg x \Rightarrow x \end{array}$$

In this example, the current fact level contains the facts q, p, r, but not $\neg x$. For action o_2, only its unconditional effect e would be added to the fact level, because the effect condition $\neg x$ of the conditional effect x is not satisfied. The problem is now that actions can be selected for parallel execution, but a plan has to be valid for all possible linearizations. One possible linearization of the two actions would schedule o_1 before o_2 and therefore $\neg x$ would hold when o_2 is executed, i.e., the action would additionally cause the side-effect of reverting the truth value of x. This could lead to potential conflicts later in the plan the planner must be aware of.

Therefore, it has to be tested if an action at the same action level can establish effect conditions of another action through its conditional effects. Such effect conditions and effects are added to the graph. These nodes, which are shown in Figure 8 in dashed-dotted lines are considered when the search algorithm enters its conflict resolution stage, but they are never considered as a choice to achieve a goal, because they become true only in some linearization, but not in all. In the example, o_2 cannot be selected to achieve x because the required effect condition $\neg x$ only holds if o_1 is executed *before* o_2. The other linearization for the parallel choice of actions would establish $\neg x$ "too late", i.e., not form a valid plan to reach this goal.

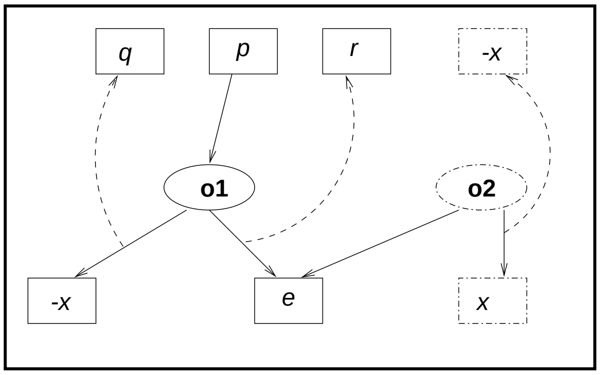

Figure 8: Completion of conditional effects in IPP.

After each level is completed, the mutual exclusive pairs of facts in $N_F(n+1)$ and actions in $N_O(n)$ are determined. Here, a significant amount of work can also be saved by inheriting mutex information from the previous action and fact level [LF98b]. For example, actions that are mutually exclusive because of interference, will be so independent of the fact level. Only exclusivity of actions because of competing needs can disappear. Furthermore, only newly added actions have to be compared with each other and with the actions that are already contained in the graph.[18]

The planning graph construction terminates when all goal facts occur in $N_F(max)$ or if the goals are unreachable, see Theorem 1. The following algorithm summarizes the process of graph expansion.

input:	a planning problem $\mathcal{P}(\mathcal{O}, D, I, G)$	
output:	$\Pi(N, E)$ `/* the planning graph or ⊥ */`	
	max `/* the number of fact levels generated */`	
	$\mathsf{Mutex}(0) \ldots \mathsf{Mutex}(max)$ `/* exclusive pairs at all graph levels */`	
initialization:	n	$:= 0$
	$N_F(0)$	$:= I$ `/* initial state is level zero */`
	max	$:= 0$
	$N = N_F$	$:= N_F(0)$
	N_O	$:= \emptyset$ `/* no actions in the first invocation */`
	$E = E_P = E_E$	$:= \emptyset$ `/* no edges */`

0. termination condition 1: $G \subseteq N_F(n)$ and G is non-exclusive in $N_F(n)$
`/* positive case: search can start now */`
return $\Pi(N, E)$, max, $\mathsf{Mutex}(0) \ldots \mathsf{Mutex}(max)$.

termination condition 2: Theorem 1
`/* negative case: problem is unsolvable */`
`/* graph has leveled off, goals are unreachable*/`

1. Graph expansion:
build initial graph layer $N_F(0)$
loop
exit: termination condition 1 or 2
initialize a new action level $N_O(n) := \emptyset$
initialize a new fact level $N_F(n+1) := \emptyset$
call expand$(n+1)$
$n := n + 1$, $max := n$
endloop

<u>expand$(n+1)$</u>

1.1 add no-ops:
for all facts $f \in N_F(n)$
create a no-operator nop_f
$N_O(n) := N_O(n) \cup \{nop_f\}$ `/* add no-op to new action level */`
$E_P(n) := E_P(n) \cup \{(f, nop_f)\}$ `/* update set of precondition edges */`
$N_F(n+1) := N_F(n+1) \cup \{f\}$ `/* add fact to new fact level */`
$E_E(n) := E_E(n) \cup \{(nop_f, f, \emptyset)\}$ `/* update set of effect edges */`
endfor

1.2. action level expansion and update of precondition edges:
compute set $\Sigma(n)$ of all actions o that are applicable in level $N_F(n)$ (Definition 16)
$N_O(n) := N_O(n) \cup \Sigma(n)$ `/* update action nodes, one node per action in` $\Sigma(n)$ `*/`
for all $o \in \Sigma(n)$
for all preconditions $p \in \mathrm{pre}_0(o)$ `/*` $p \in N_F(n)$ `*/`
$E_P(n) := E_P(n) \cup \{(p, o)\}$ `/* update precondition edges */`
endfor
endfor

[18]This seems to be obvious, but GRAPHPLAN and earlier versions of IPP completely recomputed all mutex pairs at each level.

1.3. fact level expansion and update of effect edges:

```
for all o ∈ Σ(n) /* only actions other than no-ops */
    for all pre_i(o) ⇒ e
        if pre_i(o) ⊆ N_F(n) and all facts in pre_i(o) ⊆ N_F(n) are non-exclusive and
          pre_i(o) is non-exclusive of pre_0(o) /* test only for i ≠ 0 */
          then N_F(n + 1) := N_F(n + 1) ∪ {e}
                /* new nodes are only created for new facts */
                E_E(n) := E_E(n) ∪ {(o, e, pre_i(o))}
          else nothing /* ignore effects with unsatisfied conditions */
        endif
    endfor
endfor
for each action o ∈ N_O(n) compute Mutex(n)
for each fact f ∈ N_F(n + 1) compute Mutex(n + 1)
    loop /* do effect completion for conditional effects */
        for all effects pre_i ⇒ eff_i not yet added to the graph
          if pre_i ⊆ N_F(n + 1) and non-exclusive
          /* effect can be enabled by an action at the same action level */
                then add pre_i ⇒ eff_i to the graph
          endif
        endfor
        exit: N_F(n + 1) remains unchanged
        mark newly effects as dummy nodes at level n + 1
        mark newly added preconditions as dummy nodes at level n
        /* dummy nodes can never be achieved as goals */
    endloop
```

Given the following sets of actions $\mathbf{op_1}, \mathbf{op_2}, \mathbf{op_3}$

$\mathbf{op_1} : d1$	$\mathbf{op_2} : d2$	$\mathbf{op_3} : d3$
$\top \Rightarrow a\ ,\ \neg d1$	$\top \Rightarrow b\ ,\ \neg d2$	$\top \Rightarrow c$
	$y \Rightarrow x$	$y \Rightarrow x$
	$x \Rightarrow \neg a$	$z \Rightarrow y$

together with the initial state $I = \{d1, d2, d3, x, y, z\}$ and goals $G = \{a, b, c\}$, Figure 9 shows the planning graph that is generated until the goals are reached for the first time. No-ops are omitted and two separate fact layers are drawn to distinguish preconditions and effects more easily.

IPP planning graphs inherit all properties of the original planning graphs as described in [BF95]. Their size is polynomially restricted in their depths, the number of actions, and the number of facts in the initial state. Fact and action levels grow monotonically and mutual exclusivity relationships decrease. The level-off property and the termination test remain unchanged.

3.4 The Search Algorithm

Similar to GRAPHPLAN, the planning graph is built until the goals are reached for the first time or the graph has leveled off and it turns out that the goal state is not reachable. Note that the planning graph contains only the initial fact level $N_F(0)$ if $G \subseteq I$, i.e., if

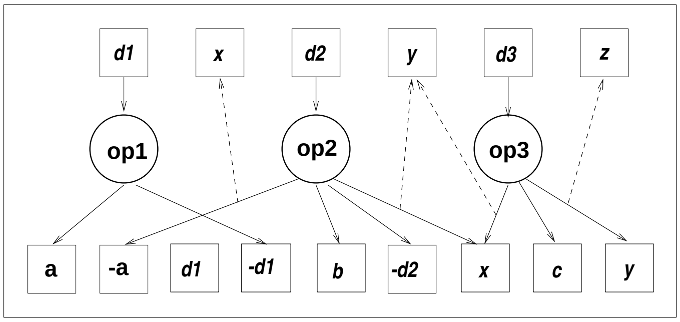

Figure 9: An example planning graph in explicit representation. Each conditional effect edge has a pointer to its effect condition. The three actions are non-exclusive because they interfere only over their conditional effects.

the goal state holds already in the initial state. If the goals are unreachable, "can't reach non-exclusive goals" is returned. If they are already satisfied in the initial state, the empty plan is returned.

A. initial creation of the planning graph

call the planning graph generation algorithm
returns $\Pi(N, E)$ `/* shortest graph containing goals */`
max `/* the number of proposition levels generated */`
Mutex(n)`/* all exclusive pairs at level` n `*/`
if $max = 0$ then return $\langle\rangle$ and stop planning.
if termination condition 2 return "can't reach non exclusive goals".

If none of the special cases applies, a recursive search algorithm search(n) over the planning graph is initialized that starts at the last fact level ($n = max$) and terminates when actions from level $N_O(n-1)$ have been successfully selected to achieve the goals at fact level $N_F(n)$ for all levels $1 \leq n \leq max$. The search algorithm differs in four points from its GRAPHPLAN predecessor:

1. the use of explicit negation which requires to test for consistency,
2. the selection procedure for actions at each level, which takes into consideration that an action can possibly achieve the same goal fact under different effect conditions,
3. the resolution of conflicts caused by conditional effects, and
4. the minimality test for an action set.

A goal can now have two different origins. First, goals can result from preconditions and effect conditions of actions that have to be selected at the action level $n-1$ to achieve the goals at level n. Second, goals can result from the confrontation of effect conditions in order to prevent harmful conditional effects among the selected actions.

B. level-guided expansion and search

loop

`/* initialization of search parameters /*`

n $:= max$ `/* current level */`

$\mathcal{G}_{max}$ $:= G$ `/* fixed goal set at max level */`

$\mathcal{G}_n := \emptyset, \forall\, n$ with $1 \leq n \leq max - 1$ `/* variable goal set at other levels */`

$\Delta_n := \emptyset, \forall\, n$ with $0 \leq n \leq max - 1$ `/* set of selected edges at each level */`

call search(n) `/* find plan in given planning graph */`

if *FAILURE*

  then if $\mathcal{G}_{max}$ is unsolvable (Theorem 2)

    then return "no solution" and stop planning

    else call expand($n+1$) `/* expand planning graph by one level */`

   endif

  else plan found: return set of selected actions at each level

endif

endloop

Given a set of goal facts $\mathcal{G}_n \subseteq N_F(n)$ action selection proceeds as follows: First, a set of effect edges $\Delta_{n-1} \subseteq E_E(n-1)$ is selected at action level $N_O(n-1)$ that connect to the facts in the goal set $\mathcal{G}_n$. In contrast to GRAPHPLAN that selects an action, IPP selects a particular effect edge that is linked to the goal, because the same action can achieve a goal fact under different types of effects (conditional or unconditional) or under different effect conditions. By selecting an edge, an action is chosen indirectly (marked as "used").

1. <u>choice point</u>: select set of effect edges Δ_{n-1}

for each goal $g \in \mathcal{G}_n$

  if a "used" action $o \in N_O(n-1)$ has an unconditional effect edge to g

   then $\mathcal{G}_n := \mathcal{G}_n \setminus g$

   `/* skip goals that are already unconditionally achieved */`

   else $\Delta_{n-1} := \Delta_{n-1} \cup (o, g, \mathrm{pre}_i(o))$ and $\mathcal{G}_{n-1} := \mathcal{G}_{n-1} \cup \mathrm{pre}_i(o) \cup \mathrm{pre}_0(o)$

    such that the following conditions are satisfied:

    a) o is non-exclusive to all actions in Δ_{n-1}

    b) $\mathcal{G}_{n-1}$ is non-exclusive and consistent

    c) $\mathcal{G}_{n-1} \cup \mathrm{eff}_0(o)$ is consistent for <u>all</u> o

    d) $\mathcal{G}_n \cup \mathrm{eff}_0(o)$ is consistent

  endif

endfor

The conditions a) to d) guarantee that a valid choice is made and no unnecessary search effort is spent on trying to satisfy goals for which it is very obvious that they can never be satisfied. First, the selected effect must belong to an action that is non-exclusive to all other selected actions. Second, the new goal set at fact level $n-1$ must remain consistent and only contain facts that are non-exclusive of each other, i.e., it is possibly satisfiable. Third, no unconditional effect of *every* selected action must be inconsistent with the new goals because otherwise there is a linearization of the parallel action set in which the precondition of an action or the effect condition of a selected effect will not be valid. Finally, the unconditional effects of the newly added action are consistent with the current goals at fact level n because otherwise there is a linearization, namely with this action at the end, where the old goals will not be valid. If no such choice of edges can be found, the planner backtracks to modify the current goal set $\mathcal{G}_n$.

Given the example in Figure 9 and the goals $\mathcal{G}_1 = \{a, b, c\}$, the planner has as its only choice the edges $(op_1, a, \emptyset)$, $(op_2, b, \emptyset)$, and $(op_3, c, \emptyset)$. All conditions are satisfied, i.e., the actions are non-exclusive and the new goal set is consistent. The new goals $\mathcal{G}_0$ are obtained as the preconditions $\{d1, d2, d3\}$ of the actions.

Now the planner has to repeat the following operations until a fixpoint is reached:

(1) Given $\mathcal{G}_{n-1}$ does it imply effect conditions of effects of actions in Δ_{n-1} other than the selected ones? This means that other conditional effects will also be made true and have to be considered for conflict resolution by the planner. All these edges are added to Δ_{n-1} and considered for the minimality test.

(2) A minimality test is executed because if the selected edge set is non-minimal the planner can backtrack immediately.

(3) The conflict resolution is activated and has to consider *all* conditional effects of *all* used actions. As soon as it detects a conflicting conditional effect, it will augment the goals $\mathcal{G}_{n-1}$ such that the effect conditions of this effect will not become valid. Then it returns to (1). and repeats the conflict resolution cycle by testing again if other conditional effects are now implied by the extended goal set. This process continues until the goal set reaches a fixpoint. If the action set is still minimal, search can proceed at the next level in the graph with $\mathcal{G}_{n-1}$ as input.

The minimality test proceeds as follows: For each used action, all unconditional and conditional effect edges are considered, whose effect conditions are implied by the new goals $\mathcal{G}_{n-1}$. The original test by Blum and Furst is applied to the used actions wrt. the collected set of edges: The action set is minimal if each action (which can also be a no-op) achieves at least one goal fact that is not achieved by any other action considering the collected edges. If the action set is non-minimal, the planner backtracks and the next choice of edges and actions is computed. The difference to GRAPHPLAN lies in the necessity to repeat the minimality test always when the goal set has been extended by the conflict resolution.

Now, the planner has to enter the conflict resolution stage. The selected actions at level $n-1$ are tested for interference caused by conditional effects. To prevent a harmful conditional effect, its effect condition is negated and added to the goal set to make sure that this condition does not hold in the state in which the action is executed. Since IPP does not know which specific linearization will be chosen for execution because it selects a set of "parallel" actions, it has to address two tasks:

1. By establishing a negated effect condition as a goal at level $n-1$ it guarantees that the undesired effect condition does not hold in the state in which the "parallel" action set is executed.

2. By testing that no action in the parallel set reverts the negated effect condition it guarantees that all linearizations lead to a valid execution sequence.

Two different possibilities for conditional interference have to be tested:

1. $\exists\, o\, \exists\, i : \text{eff}_i(o) \cup \mathcal{G}_{n-1}$ is inconsistent.
   In this situation, the conditional effect of o is inconsistent with the precondition of a selected action or the effect condition of a selected effect that are contained in $\mathcal{G}_{n-1}$. For example, if an edge was selected that establishes $p \Rightarrow q$, but another action is used that has the undesired effect $r \Rightarrow \neg p$ then $\neg p$ must be prevented by adding $\neg r$ to the new goal set. Otherwise, a non-executable action or conditional effect would occur in some linearization of the parallel action set.

2. $\exists\, o\, \exists\, i : \text{eff}_i(o) \cup \mathcal{G}_n$ is inconsistent.
   In this situation, there is an action whose conditional effect if added to $\mathcal{G}_n$ makes the goal set inconsistent. For example, if the goals at time step n are a, b, c and a selected action has the conditional effect $x \Rightarrow \neg a$ then it could happen that accidentally x is true when this action is executed. It would therefore make $\neg a$ true as an effect and since this action must also be valid as the last one in a linearization, the goal a cannot hold in the goal state. The main difference to unconditional effects causing inconsistency is that these cannot be prevented and cause backtracking immediately, while inconsistency caused by conditional effects can possibly be prevented.

If the planner discovers conditional interference, it checks if the effect conditions $\text{pre}_i(o)$ of a harmful conditional effect are contained in the set $\mathcal{G}_{n-1}$. In this case, backtracking to choose a new Δ_{n-1} is necessary, because $\text{pre}_i(o)$ is causally linked to the new goals $\mathcal{G}_{n-1}$, i.e., whenever they are achieved, the context for the harmful side effect is established as well. Otherwise, the set $\text{pre}_i(o) \setminus \mathcal{G}_{n-1}$ of effect conditions that are not in the new goal set is determined and negated. For example, if the planner has constructed the new goals $a \wedge b$ and the effect condition to prevent is $a \wedge c \wedge d$ then only $c \wedge d$ can probably be prevented as a must be made true. Negating this remaining condition leads to $\neg c \vee \neg d$ that is conjunctively joined to the goal set $a \wedge b$.

The planner has to determine the disjunctive normal form of this formula and obtains the disjunctive goal $(a \wedge b \wedge \neg c) \vee (a \wedge b \wedge \neg d)$, which leads to a choice point, i.e., each disjunct is tried before backtracking occurs. Obviously, a conditional effect is only achieved if the effect condition as a whole (set) is established, i.e., preventing one single literal per effect condition already avoids the undesired effect. Considering the set difference $\text{pre}_i(o) \setminus \mathcal{G}_{n-1}$ is a simple trick to avoid that the augmented goal set becomes inconsistent.

In the example of Figure 9, the planner finds out that op_2 has a conditional effect $\neg a$ with condition x that is inconsistent with the goal a, i.e., $\neg x$ is added to the goal set. Since the goal set has changed, the planner has to make sure that no other conditional effect is triggered. This is not the case in the example, but there is still the action op_3 that adds x under condition y, i.e., it could trigger the undesired effect. This means, if op_3 is executed before op_2 in a state in which y holds, the condition x becomes true and op_2 will make the goal a impossible. Therefore, $\neg y$ must be added to $\mathcal{G}_0$ as well. It does not matter that op_2 adds x under condition y, because the effect x of op_2 will always hold in the next state after the condition x for the effect $x \Rightarrow \neg a$ of op_2 has been evaluated. Again, the goal set has been extended and therefore, the tests are repeated. But again no new effects are triggered and only op_3 can add the effect condition y of its own harmful conditional effect and this does not matter. Thus, a fixpoint is reached and all effect conditions of effects that potentially could cause conflicts are already prevented by $\mathcal{G}_0$.[19]

[19] Since $\mathcal{G}_0$ contains goals $\neg x, \neg y$ that do not hold in the example's initial state, the planner would fail during completion and had to backtrack.

2. <u>Fixpoint Iteration:</u> conflict resolution

for all actions o marked as "used" in $N_O(n-1)$

  if exists an edge $(o, g, \mathrm{pre}_i(o))$ such that $\{g\} \cup \mathcal{G}_n$ or $\{g\} \cup \mathcal{G}_{n-1}$ are inconsistent

   `/* goal, precondition, or effect condition threatened by effect g */`

   then $\mathcal{G}_{n-1} := \mathcal{G}_{n-1} \cup \neg(\mathrm{pre}_i(o) \setminus \mathcal{G}_{n-1})$

    (backtrack if $\mathrm{pre}_i(o) \setminus \mathcal{G}_{n-1} = \emptyset$)

    determine disjunctive normal form of $\mathcal{G}_{n-1}$

    <u>choice point</u>: choose one of the disjuncts as new goals $\mathcal{G}_{n-1}$

   else invoke search$(n-1)$

  endif

endfor

If all possible choices of actions to achieve the goals at level n have failed, the planner has to backtrack to fact level $n+1$ to change the set $\mathcal{G}_n$ by selecting different actions at level n because it has proven these goals to be unsolvable. Unsolvable goal sets are memoized at each level,see [HK99] for a description of the IPP subset memoization algorithm.

Backtracking at the maximum level of the graph is not possible and leads to a failure of search(max) on the planning graph, i.e., no valid plan could be extracted. The planning graph is extended by another action and fact level and the planner searches again on the extended graph. This process of interleaved graph expansion and search terminates if either a set Δ_n was successfully selected at each level of the graph – the plan is the set of all actions that are marked as "used" – or the problem turns out to be unsolvable, cf. Theorem 4 below.

3.5 An Example

To illustrate the search algorithm in more detail let us consider the following example from the *briefcase* domain, which comprises the following three operators:

move*(?x, ?y:location)*
precondition: *at-b(?x)*
effects: *at-b(?y)* , ¬ *at-b(?x)*
ALL ?o object in(?o) ⇒ *at(?o, ?y)* , ¬ *at(?o, ?x)*

take-out*(?o:object, ?x:location)*
precondition: *in(?o)* , *at-b(?x)*
effects: ¬ *in(?o)*

put-in*(?o:object, ?x:location)*
precondition: ¬ *in(?o)*, *at(?o, ?x)* , *at-b(?x)*
effects: *in(?o)*

Let us assume that the initial state and the goals are given as below, i.e., an object o is inside the briefcase which is at location l. The briefcase has to be moved to location m, but the object should remain at location l.

initial state: *in(o)* , *at-b(l)*, *at(o, l)*
goals: *at(o, l)* , *at-b(m)*

Figure 11 at the end of this section shows the implicit representation of the planning graph that is generated for the example. In action level 0 of the graph, the two actions **take-out(o,l)** and **move(l,m)** are applicable. They are mutually exclusive because the effect $\neg$ *at-b(l)* of **move(l,m)** is inconsistent with the precondition *at-b(l)* of the **take-out** action. The action **put-in(o,m)** becomes applicable at action level 2, although its preconditions are already true at fact level 1, but they are exclusive. The **move(l,m)** action has a conditional effect that is inconsistent with the subgoal *at(o,l)* and that cannot be prevented because the object o is in the briefcase in the initial state, i.e., the two subgoals are exclusive at fact level 1. The graph has to be extended to the second level from which search starts.

Given the goals *at(o,l)* and *at-b(m)* at the second level, IPP has several choices of actions with marker values lower than 2 to achieve them. The first subgoal can be achieved using the the action **move(m,l)** or a no-op. The second subgoal can be achieved using either **move(l,m)** or a no-op. Let us only consider the choice that will lead to a valid plan, i.e., the no-op to achieve the goal *at(o,l)* and the action **move(l,m)** to achieve the goal *at-b(m)*, see Figure 10 for the relevant part of the planning graph.

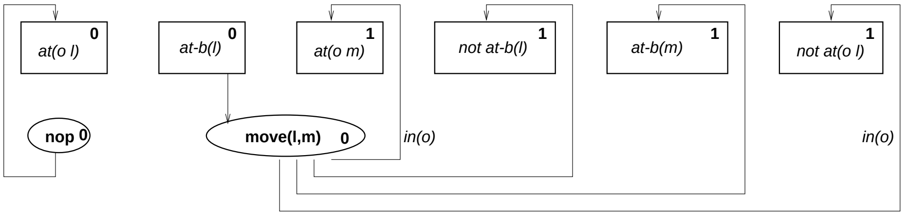

Figure 10: Relevant part of the planning graph to achieve the goals at fact level 2.

The selected action set is minimal and non-exclusive. The preconditions of both actions with *at(o,l)* from the no-op and *at-b(l)* from the **move** action form the new goals $\mathcal{G}_1$, which are consistent. The conflict resolution tests if one of the actions has a conditional effect that is inconsistent with the old or the new goals. Under condition *in(o)*, **move(l,m)** has the conditional effect $\neg$ *at(o,l)*, which is inconsistent with the current and the new goal set. Therefore, this effect condition is negated, yielding $\neg$ *in(o)*, and added to the new goal set, which remains consistent. Thus, at fact level 1 the planner has to achieve the goals $\neg$ *in(o)*, *at(o,l)*, and *at-b(l)*.

To achieve the goal *at-b(l)* at fact level 1, actions with a marker value of 0 need to be considered and thus, only a no-op can be used. The action **move(m,l)** also points to this goal, but its marker value is too large, i.e., 1 instead of 0. Similarly, for the goal *at(o,l)*, which is made true by a no-op or conditionally by the **move(m,l)** action, but again only the no-op has a marker value of 0. To achieve $\neg$ *in(o)*, only the action **take-out(o,l)** can be used because it has a marker value of 0. Such the planner selects two no-ops and the action **take-out(o,l)**. This choice forms the goals in the initial state: *at-b(l)*, *at(o,l)* *in(o)*, which are satisfied. Since this action set does not contain any conditional effects, no conflict resolution needs to be performed. The planner terminates successfully and returns the following plan:

time step 0: take-out(o,l)
time step 1: move(l,m)

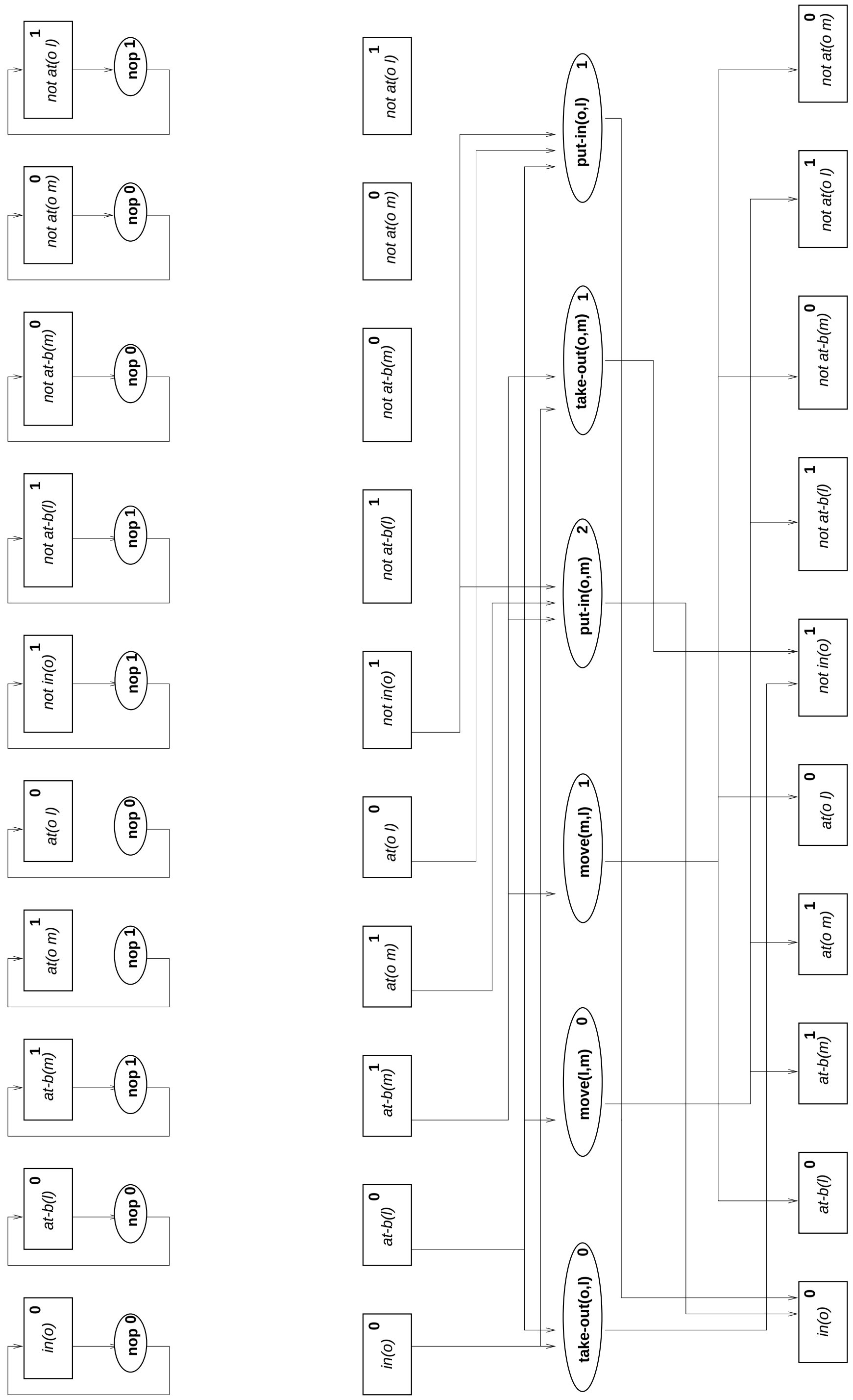

Figure 11: Planning graph for the briefcase example. The above part shows no-ops and the lower part shows the "normal" actions. The fact level is shown twice to separate preconditions and effects. Internally the effect edges point back into the same fact level, but this would be unreadable. Markers in the nodes say at which level an action or a fact occur for the first time. The graph has been extended to the second level only, i.e., no marker values greater than 2 occur.

3.6 Soundness and Completeness

Theorem 3 (Soundness) *Let $(\mathcal{O}, D, I, \mathcal{G})$ be a planning problem. If* IPP *returns a parallel plan $\mathcal{P} = \langle Q_1, Q_2, \ldots Q_{max-1}\rangle$ then*

$$\bigcap R(I, \langle Q_1, Q_2, \ldots, Q_n\rangle) \supseteq \mathcal{G}$$

holds, i.e., $\mathcal{P}$ is a solution.

Proof: The proof is based on the execution semantics as developed in Definition 23 and proceeds by induction over the length of the parallel plan, i.e., the covered time steps.

base case: $\mathcal{P} = \langle\rangle$

Definition 23 yields $R(I, \langle\rangle) = I$. This means $I \supseteq G$ must be proven. It follows from Part A of the planning graph generation algorithm, which returns the empty plan when max is 0. This happens only if the goals are contained in the initial state according to the termination condition of the graph generation algorithm.

induction step: Given the assumption $\bigcap R(I, \langle Q_0, Q_2, \ldots, Q_{k-1}\rangle) \supseteq \mathcal{G}_k$ it can be proven that $\bigcap R(I, \langle Q_0, Q_2, \ldots, Q_k\rangle) \supseteq \mathcal{G}_{k+1}$ follows for all time steps $0 \leq n \leq k$.

Following Definition 23, $R(I, \langle Q_1, Q_2, \ldots, Q_{k-1}\rangle)$ produces a set of states as a result. Each state S_P satisfies the goals $\mathcal{G}_k$, i.e., $S_P \supseteq \mathcal{G}_k$ holds (induction assumption). Let us consider one of these states S_P. In this state, each linearization of Q_k must be executable and lead to a state S'_P satisfying the goals $\mathcal{G}_{k+1}$ if $S_P \supseteq \mathcal{G}_k$ is true. Let us consider one such linearization $Q_k = \langle q_1, \ldots, q_m\rangle$ to prove this claim.

In order for $Res(S_P, \langle q_1, \ldots, q_m\rangle) \supseteq \mathcal{G}_{k+1}$ to hold, the following conditions must be satisfied:

1. Each action must be applicable.
2. Each condition of an effect that was selected to achieve a goal must be valid in the state where the action is executed.
3. Each harmful effect must be prevented.

In order to satisfy (1), the action's preconditions must be satisfied in the state in which the action is applied independent of the linearizations of the parallel action set. Similarly in order to satisfy (2), the effect conditions must be satisfied and remain valid in all linearizations. Finally, (3) means that no action in the linearization is allowed to make a precondition, an effect condition, and a required effect of another action invalid.

All preconditions of actions in Q_k and all effect conditions of selected effects are contained in the set $\mathcal{G}_k$, which follows from the construction of the search algorithm in Part B1. With the induction assumption, we know that $\mathcal{G}_k$ is satisfied, i.e., (1) and (2) are satisfied in S_P. The selected effects of each action achieve the set $\mathcal{G}_{k+1}$. It remains to show that $\mathcal{G}_k$ remains satisfied over all linearizations and that no conditional or unconditional effect of an action can invalidate $\mathcal{G}_{k+1}$.

Let us assume that there is an action q_i that has a harmful effect. This can happen in exactly four different ways:

1. q_i has an unconditional effect that is inconsistent with $\mathcal{G}_{k+1}$.

2. q_i has a conditional effect that is inconsistent with $\mathcal{G}_{k+1}$.

3. q_i has an unconditional effect that is inconsistent with $\mathcal{G}_k$.

4. q_i has a conditional effect that is inconsistent with $\mathcal{G}_k$.

All four possibilities are tested by the planning algorithm. Cases 1 and 3 are tested in part B1, conditions c) and d) and would never admit the choice of q_i. Cases 2 and 4 are tested during conflict resolution when inconsistencies of all action's effects with the sets $\mathcal{G}_k$ and $\mathcal{G}_{k+1}$ are investigated. One disjunct of the negated effect condition $\text{pre}_i(q_i)$ causing the harmful effect would be added to $\mathcal{G}_k$ and the conflict resolution repeated. This guarantees that $\mathcal{G}_k \models \neg\text{pre}_i(q_i)$ and that there is no other action in the set that would reestablish $\text{pre}_i(q_i)$ because in this case, its effect had to be inconsistent with the augmented goal set, which is impossible because of the fixpoint process during the conflict resolution. Note that the effect completion process, which adds conditional effects of actions whose effect conditions could probably be established by other actions at the same level in the graph, is crucial for the planner to be aware of all potential conflicts. ■

An immediate consequence is the termination of IPP on unsolvable problems.

Theorem 4 *If* IPP *returns* $\perp$ *then* $(\mathcal{O}, D, I, \mathcal{G})$ *has no solution.*

Proof: Follows from the proofs of Theorems 1, 2, and 3. ■

Completeness follows from the observation that

- planning graphs contain all applicable actions at each time step,
- a systematic iterative-deepening backward search is performed that tries all possibilities to achieve a goal.

Theorem 5 *If* $(\mathcal{O}, D, I, \mathcal{G})$ *has a solution* $\mathcal{P} = \langle Q_1, Q_2, \ldots, Q_{max-1}\rangle$ *of at most the length* $k = |Q_1| + |Q_2| + \cdots + |Q_{max-1}|$*, i.e., the length of the linear plan* $\langle o_1, o_2, \ldots, o_k\rangle$ *not containing parallel action sets, then* IPP *outputs a solution after* k' *graph expansion steps with* $k' \leq k$*.*

Proof: First, if IPP finds a solution after k' expansion steps with $k' < k$, then nothing needs to be proven. Second, let us assume that IPP returns $\perp$ after k' expansion steps with $k' < k$. This is impossible, because with Theorem 4 it follows that no solution can exist in this case, which contradicts the assumption. Finally, let us assume IPP has performed k expansion steps and constructed the kth fact level and not yet returned a plan. The systematic search starting in level k must now find the solution $\langle o_1, o_2, \ldots, o_k\rangle$ or another one with the same number of time steps because it enumerates all possible choices of actions to achieve the goals at each level. ■

Unfortunately, it is impossible to prove that IPP generates shortest plans in all cases according to the execution semantics. In the original GRAPHPLAN paper, the problem of unnecessary separation of actions is avoided by defining interference between actions in a very strict way, see the discussion at the beginning of this section in Section 3.1. Though a unique state does not need to result from a parallel conditional action set anymore, IPP will sometimes separate conditional actions although they could be executed in parallel. The following example illustrates a subtle interference among conditional effects that is interpreted as a conflict by the planner. Given the two actions

$$\begin{array}{llll} \mathbf{o_1} : & \top & \qquad\qquad \mathbf{o_2} : & \top \\ & \top \Rightarrow b & & \top \Rightarrow d \\ & c \;\Rightarrow e\;,\; \neg c & & c \;\Rightarrow e\;,\; \neg c \end{array}$$

together with the initial state $I = \{c\}$ and goal $G = \{b, d, e\}$, IPP decides that both actions cannot be executed in parallel, because one action deletes an effect condition of the other. However, in this special case, each of the actions can add the desired goal e and in fact, one could ignore this interference and generate the parallel plan $\langle\{o_1, o_2\}\rangle$. Although testing for such special cases is possible in principle, trading optimality for efficiency seems to be a reasonable decision in such situations.

References

[AHEK94] W. Anderson, J. Hendler, M. Evett, and B. Kettler. Massively parallel matching of knowledge structures. In H. Kitano and J. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 53–71. MIT Press, Cambridge, MA, 1994.

[AW98] C. Anderson and D. Weld. Conditional effects in Graphplan. In J. Allen, editor, *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems*, pages 44–53. AAAI Press, Menlo Park, 1998.

[BF95] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1636–1642. Morgan Kaufmann, San Francisco, CA, 1995.

[BF97] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1–2):279–298, 1997.

[Eck98] J. Eckerle, 1998. personal communication.

[FL97] M. Fox and D. Long. An efficient algorithm for managing partial orders in planning. In *SIGART Bulletin*. ACM Press, 1997.

[FL98] M. Fox and D. Long. Efficient implementation of the plan graph in STAN. unpublished, 1998.

[GK97] B. Gazen and C. Knoblock. Combining the expressivity of UCPOP with the efficiency of Graphplan. In Steel [Ste97], pages 221–233.

[HK99] J. Hoffmann and J. Koehler. A new method to index and query sets. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 462–467. Morgan Kaufmann, San Francisco, CA, 1999.

[KLP97] S. Kambhampati, E. Lambrecht, and E. Parker. Understanding and extending Graphplan. In Steel [Ste97].

[KNHD97] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. Technical Report 88, University of Freiburg, Institute of Computer Science, 1997. available at http://www.informatik.uni-freiburg.de/~ koehler/ipp.html.

[Kno94] C. Knoblock. Generating parallel execution plans with a partial-order planner. In K. Hammond, editor, *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*, pages 98–103. AAAI Press, Menlo Park, 1994.

[Koe99] J. Koehler. Metric planning using planning graphs - a first investigation. Technical Report 127, University of Freiburg, Institute of Computer Science, 1999. available at http://www.informatik.uni-freiburg.de/~ koehler/ipp.html.

[KS92] H. Kautz and B. Selman. Planning as satisfiability. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363. John Wiley & Sons, Chichester, New York, 1992.

[KS96] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 14th National Conference of the American Association for Artificial Intelligence*, pages 1194–1201. AAAI Press, 1996.

[KS98] H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98*. unpublished material, 1998.

[LF98a] D. Long and M. Fox. Domain independent planner compilation. In *Working notes of the Workshop on Knowledge Engineering and Acquisition for Planning, held in conjunction with AIPS-98*. unpublished material, 1998.

[LF98b] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. unpublished manuscript, 1998.

[Lot98] A. Lotem. Applying the Graphplan approach to HTNplanning (research proposal). unpublished, 1998.

[McD96] D. McDermott. A heuristic estimator for means-ends analysis in planning. In B. Drabble, editor, *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems*, pages 142–149. AAAI Press, Menlo Park, 1996.

[NDK97] B. Nebel, Y. Dimopoulos, and J. Koehler. Ignoring irrelevant facts and operators in plan generation. In Steel [Ste97], pages 338–350.

[Neb98] B. Nebel. On the compilability and expressive power of propositional planning formalisms. Technical Report 101, Albert-Ludwigs-University Freiburg,, 1998.

[Ped91] E. Pednault. Generalizing nonlinear planning to handle complex goals and actions with context-dependent effects. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 240–245. Morgan Kaufmann, San Francisco, CA, 1991.

[PW92] J. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In B. Nebel, W. Swartout, and C. Rich, editors, *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–113. Morgan Kaufmann, San Mateo, 1992.

[SG98] I. Serina and A. Gerevini. Local search techniques for planning graphs (preliminary report). unpublished, 1998.

[Ste97] S. Steel, editor. *Proceedings of the 4th European Conference on Planning*, volume 1348 of *LNAI*. Springer, 1997.