Extending Planning Graphs to an ADL Subset*

Jana Koehler Bernhard Nebel Jörg Hoffmann
Yannis Dimopoulos
Institute for Computer Science
Albert Ludwigs University
Am Flughafen 17
79110 Freiburg, Germany
<last-name>@informatik.uni-freiburg.de
http://www.informatik.uni-freiburg.de/~ koehler/ipp.html

Abstract

We describe an extension of GRAPHPLAN to a subset of ADL that allows conditional and universally quantified effects in operators. The data structure of planning graphs is extended to cope with the more expressive operators in such a way that most of the interesting properties of the original GRAPHPLAN formalism are preserved. A sound and complete planning algorithm extracts plans from planning graphs and terminates on unsolvable problems. A new efficient technique for subset memoization is presented to speed up the planner and we prove that GRAPHPLAN's termination test remains complete under subset memoization.

Results from an empirical evaluation demonstrate that this extension of the original GRAPHPLAN system comes with almost no computational overhead if carefully implemented. The resulting system is the kernel algorithm of our own interference progression planner IP^2 that competes very well with other complete planners for operators with conditional and universally quantified effects such as UCPOP and Prodigy.

Technical Report No. 88
1997

*This report is a significantly extended version of a paper that appeared in the Proceedings of the 4th European Conference in Planning, Toulouse 1997.

Contents

1 Introduction

Planning with planning graphs [2] has received considerable attention because of the spectacular runtime behavior reported for the GRAPHPLAN system. 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 us to use the approach as the kernel algorithm for our own planner IP^2 that we intend to use as the high-level front-end of the Pioneer mobile robot platform [1].[1] But GRAPHPLAN also has its limitations. First, its performance can decrease dramatically if too much irrelevant information is contained in the specification of a planning task, see [10] for a detailed analysis. Second, its simple representation language is restricted to pure STRIPS operators – no conditional or universally quantified effects are allowed and it was unclear whether the underlying planning algorithm could be extended to more expressive formalisms [3, 6, 9].

In principle, sets of STRIPS operators can be used to encode conditional effects. For example, the *move* operator from the well-known Briefcase domain that specifies that all objects which are inside a briefcase move whenever the briefcase moves (Figure 1) 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.

name:	**move-briefcase**
par:	l_1:location, l_2:location
pre:	at-b(l_1)
eff:	ADD at-b(l_2), DEL at-b(l_1)
	$\forall x$:object [in(x) $\Rightarrow$ ADD at(x,l_2), DEL at(x,l_1)].

Figure 1: Operator with Conditional and Universally Quantified Effects

But such an encoding leads to exponentially more operators which can make even small planning problems practically intractable, see Section 7. These observations motivated us to directly embed operators with conditional and universally quantified effects into planning graphs, while other features of ADL [11, 12] that are for example available in UCPOP [13] can be reasonably handled by preprocessing [6].

[1] IP^2 is an acronym for interference progression planner.

In Section 2, we introduce the basic notations and discuss a semantics for STRIPS operators with conditional effects. Section 3 outlines the main differences between the planning graphs constructed by GRAPHPLAN and IP^2 and defines the algorithm to construct planning graphs for the ADL subset. Section 4 describes the planning algorithm and in Section 5, we illustrate its behavior. Its main theoretical properties are proven in Section 6 where we also show that graphplan's termination test remains complete under subset memoization – a question that has previously remained unanswered. Results of a detailed empirical evaluation of IP^2 conclude the paper in Section 7.[2] A description of the new subset memoization algorithm that we use is contained in the Appendix, Section 9.

2 A Semantics for Parallel ADL Plans

One of the distinguished 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 operators can be executed in parallel and what the result of the execution is, cf. [3]. In the case of operators with universally quantified and conditional effects, several semantics are possible of which we discuss two in the following.

Definition 1 *An* operator *is a 4-tuple consisting of*

1. *a* name*, which is a string,*
2. *a* parameter list *of typed variables,*
3. *the* precondition φ_0*, which is a conjunction of atoms, and*
4. *the* effect *as a conjunction of possibly universally quantified formulas that are of the form $\varphi_i \Rightarrow \alpha_i, \delta_i$ where φ_i is the so-called effect condition (limited to a set of atoms) and α_i, δ_i are the actual effects (also limited to sets of atoms with α_i being the Add effects and δ_i being the Del effects).*

Note that as in GRAPHPLAN no explicit atomic negation is available in our language, instead we model atomic negation by introducing an additional predicate *not-p(x)* if $\neg p(x)$ is needed and by formulating Add and Delete effects correspondingly.

The universal quantifier may be absent and in the case of an unconditional effect $\varphi_i = \emptyset$ holds. An example of a valid operator is shown in Figure 1. An example of a non-valid effect representation is $\varphi_i \Rightarrow \forall x : \text{ADD } p(x)$, because the effect condition lies outside the scope of the universal quantifier.

[2] The system together with all benchmark problems that we used in the evaluation are available over the homepage of IP^2, which is listed in the references.

Definition 2 *The set of all ground atoms is denoted with P. As usual, a* state *$S \in 2^P$ is a set of ground atoms.*

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

$$\begin{aligned} o:\ & \varphi_0 \\ & \alpha_0, \delta_0 \\ & \varphi_1 \Rightarrow \alpha_1, \delta_1 \\ & \vdots \\ & \varphi_n \Rightarrow \alpha_n, \delta_n \end{aligned}$$

The $\varphi_i, \alpha_i, \delta_i$ are sets of ground atoms with φ_0 denoting the preconditions of o, α_0 being the Add *list and δ_0 being the so-called* Delete *list of o. A conditional effect contains the effect condition φ_i, Add list α_i and Delete list δ_i.*

Figure 2 shows a ground instance of the *move* operator for DOM(*object*)= {*letter, toy*} and DOM(*location*) = {*office, home*}.

name:	**move-briefcase**	
par:	office, home: location	
pre:	at-b(office)	
eff:	$\emptyset$	$\Rightarrow$ ADD at-b(home), DEL at-b(office);
	in(letter)	$\Rightarrow$ ADD at(letter,home), DEL at(letter,office);
	in(toy)	$\Rightarrow$ ADD at(toy,home), DEL at(toy,office).

Figure 2: A possible Ground Instance of the Move-Operator

In the following, we define the result of applying a single action to a given state.

Definition 4 *Let O be the set of all ground instances of the operators in $\mathcal{O}$, O^* denote 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 is defined as

[3]The ground instance of a universally quantified effect $\forall x\ [\varphi_i(x) \Rightarrow \alpha_i(x), \delta_i(x)]$ is the conjunction of ground instances $\bigwedge_{k=0}^{n}[\varphi_i([x/a_k]) \Rightarrow \alpha_i([x/a_k]), \delta_i([x/a_k])]$ 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.

$$Res(S, \langle o \rangle) = \begin{cases} \big(S \cup A(S,o)\big) \setminus D(S,o) & \textit{if } \varphi_0 \subseteq S \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

with

$$A(S,o) = \bigcup_{\varphi_i \subseteq S, i \geq 0} \alpha_i \qquad \textit{and} \qquad D(S,o) = \bigcup_{\varphi_i \subseteq S, i \geq 0} \delta_i$$

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

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

For a planning language of simple STRIPS actions, the definition of Res can be extended to a set of parallel actions in a straightforward way [3] such that the resulting state is uniquely defined. However, when conditional effects are allowed, it is very difficult and even very restrictive to guarantee the uniqueness property of Res. In order to achieve that a unique states results from the execution of a parallel set of ADL actions, one could come up with the following definition for $Res^\star$.

Definition 5 *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 $Res^\star(S, \langle\rangle) = S$. In the case of a sequence containing one non-empty set we obtain

$$Res^\star(S, \langle \{o_1, \ldots, o_n\} \rangle) = \begin{cases} \big(S \cup \bigcup_{i=1}^n A(S,o_i)\big) \setminus \bigcup_{i=1}^n D(S,o_i) & \textit{if (1) to (4) hold} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

(1) $\forall\, o_i : \varphi_0(o_i) \subseteq S$

(2) $\forall\, o_i, o_j : A(S,o_i) \cap D(S,o_j) = \emptyset$

(3) $\forall\, o_i, o_j, i \neq j : \big(\varphi_0(o_i) \cup \bigcup_{\varphi_l(o_i) \subseteq S} \varphi_l(o_i)\big) \cap D(S,o_j) = \emptyset$

(4) $\forall \varphi_l(o_i) : \varphi_l(o_i) \not\subseteq S \Rightarrow \varphi_l(o_i) \not\subseteq \bigcup_{j \neq i} A(S,o_j)$.

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

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

Condition (1) requires that all preconditions of all actions in the set need to be satisfied in the state S. 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 Add and Delete lists with satisfied effect conditions are disjoint. Condition (3) requires that no precondition or effect condition that was valid in the state S is made invalid by an unconditional or conditional Delete effect, while Condition (4) requires that no effect condition that was invalid in the state S is made valid by the Add effects of an action.

On one hand, this definition of $Res^\star$ leads to a unique resulting state similarly to what we obtain 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 particuar, to satisfy (2) 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

$$\begin{array}{llll} o_1 : \emptyset & & o_2 : \emptyset & \\ & a \Rightarrow \text{ADD } y, \text{ DEL } c & & b \Rightarrow \text{ADD } x, \text{ ADD } c \end{array}$$

with goal $\{x, y\}$ and initial state $S = \{a, b\}$. Trying to apply both actions to this state leads to an undefined state. But since we are not interested in the value of c (it is not a goal), we would rather obtain a set of states as result, one in which c holds and another one in which c does not hold. This can be accomplished by the following definition:

Definition 6 *Let $\mathcal{S}$ 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

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

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

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

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

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

Definition 6 will be the formal basis to prove the planner sound and complete in Section 6.

3 Planning Graphs for the ADL Subset

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 [3].

Definition 7 *A* planning problem $\mathcal{P}(\mathcal{O}, D, I, G)$ *is a 4-tuple where* $\mathcal{O}$ *is the set of operators,* D *is the domain of discourse (a finite set of typed objects), and* I *(the initial state) and* G *(the goal state) are sets of ground atoms.*

Given $\mathcal{O}$ and D, the set of all actions O is the set of possible ground instances of all operators in $\mathcal{O}$.

Definition 8 *As in* GRAPHPLAN, *we define a* planning graph $\Pi(N, E)$ *as 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 nodes and ground atom (also called fact) nodes with* $N_O \cap N_F = \emptyset$*, 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* $(N_F \times N_O)$ *and* E_A*,* E_D *which contain the Add effect edges and Del effect edges* $(N_O \times N_F \times N_F)$*.*[4]

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. All this is identical to GRAPHPLAN.

Furthermore, we keep the original notion that two actions *interfere* if one unconditionally deletes a precondition or an unconditional ADD effect of the other, i.e., they cannot simultaneously be executed in *any* world state considering only their unconditional effects. This implies that conditional effects are ignored when defining *interference* between actions, 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.

[4] We write effect edges 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, which represents the (conjunctive) effect condition under which o achieves f.

Definition 9 *Two actions o_1 and o_2* interfere *iff* $\big(\alpha_0(o_1) \cup \varphi_0(o_1)\big) \cap \delta_0(o_2) \neq \emptyset$.

We also adopt GRAPHPLAN's original notion of when two actions are mutually exclusive of each other, namely, if they interfere or if they 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. 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.

Definition 10 *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.*

Definition 11 *Two actions o_1 and o_2 at an action level $N_O(n \geq 1)$ have* competing needs *iff there exist facts $f_1 \in \varphi_0(o_1)$ and $f_2 \in \varphi_0(o_2)$ that are mutually exclusive.*

Definition 12 *Two facts $f_1, f_2 \in N_F(n \geq 1)$ are* mutually exclusive *iff there is no non-exclusive pair of actions $o_1, o_2 \in N_O(n-1)$ or no single action $o \in N_O(n-1)$ that conditionally or unconditionally adds f_1 and f_2.*

Intuitively, facts are exclusive if no possible state of the world can make both true – considering only their unconditional effects.

Definition 13 *An action o is* applicable in a fact level $N_F(n)$ *iff $\varphi_0(o) \subseteq N_F(n)$ and all $f \in \varphi_0(o)$ are non-exclusive of each other.*

Based on the previous definitions, IP^2 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.[5] Although, actions become not exclusive if potential conflicts between their conditional effects occur, we can sometimes propagate exclusivity information from preconditions and effect conditions to conditional effects.

Definition 14 *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 12.*

[5]We also explored a more pessimistic approach with a different notion of interference, but abandoned the idea in a very early stage of the project, since it complicated the algorithms. Similarly, we investigated how adding explicit atomic negation would change the definitions, but it makes them slightly more complicated.

Lemma 1 *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 $f_1 \in \alpha_i(o_1)$ and $f_2 \in \alpha_j(o_2)$ (with $i, j \neq 0$,i.e., both are conditional Add effects) one of the following conditions holds*

1. *o_1 and o_2 are mutually exclusive according to Definition 10 (including the case that the actions have competing needs, i.e., $\mathsf{mutex}(\varphi_0(o_1), \varphi_0(o_2))$ holds).*

2. *$\mathsf{mutex}(\varphi_i(o_1), \varphi_j(o_2))$, i.e., the conditional effects have competing effect conditions.*[6]

3. *$\mathsf{mutex}(\varphi_i(o_1), \varphi_0(o_2))$ or $\mathsf{mutex}(\varphi_0(o_1), \varphi_j(o_2))$, i.e., the effect conditions of one action compete with the precondition of the other.*[7]

Notice that the actions $o1$ and $o2$ are not marked as exclusive.
Proof: Case 1 follows directly from Definition 12. Case 2 and 3 follow indirectly from the same definition. 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. Consequently, not both conditional effects can be achieved in the same state, because always at least one effect condition will be unsatisfied. 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. ■

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 atoms from the initial state form the first fact level $N_F(0)$. 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 $\varphi_0(nop_f) = \{f\}$ and the only unconditional effect $\alpha_0(nop_f) = \{f\}$. For each "ordinary" action node o, precondition edges between each fact in $N_F(n)$ that is a precondition of o and the action node are established. IP^2 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 $\varphi_i(o) \Rightarrow \alpha_i(o), \delta_i(o)$, IP^2 first proceeds over the individual Add facts $f \in \alpha_i(o)$. They are added to $N_F(i+1)$ iff the following conditions are satisfied:

1. $\varphi_i(o) \subseteq N_F(n)$

2. all facts in $\varphi_i(o)$ are non-exclusive of each other in $N_F(n)$

[6] This property was independently discovered in [7].

[7] When adopting a more strict semantics such as the one in Definition 5, one can also mark two Add effects as exclusive if for all possible ways of achieving these effects holds that $\alpha_i(o_1) \cap \delta_j(o_2) \neq \emptyset$ and vice versa.

3. all facts in $\varphi_i(o)$ are non-exclusive of the facts in $\varphi_0(o)$ in $N_F(n)$

The first two conditions imply that the effect condition is applicable, similar to Definition 13. Condition 3 tests if precondition and effect condition can ever hold in the same state – although one can assume that any "reasonable" operator definition will satisfy this condition. 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 added. Otherwise, only an Add edge is established that is augmented with a pointer to all facts in $N_F(n)$ that occur in $\varphi_i(o)$. Since one 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. For each atomic Del effect $f \in \delta_i(o)$, the same tests are performed on the effect condition $\varphi_i(o)$ and a Del edge is established if $f \in N_F(n+1)$. After each level is completed, the mutual exclusive pairs of facts in $N_F(n+1)$ and actions in $N_O(n)$ are determined. The planning graph construction terminates when all goal atoms occur in $N_F(max)$ or if the goals are unreachable, see Theorem 1. The following algorithm summarizes the process of graph expansion and goal test.

---

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 */`
$\mathsf{Mutex}(n)$ `/* exclusive pairs at level n */`

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 */`
$E = E_P = E_A = E_D$	$:= \emptyset$ `/* no edges */`

---

0. termination condition: if $G \subseteq N_F(n)$ and G is non-exclusive in $N_F(n)$
return $\Pi(N, E)$, max, $\forall n$ $\mathsf{Mutex}(n)$.

1. Graph expansion:
loop
if the goals are unreachable at level $N_F(n)$ (Theorem 1)
  then return $\bot$
  else create a new fact level $N_F(n+1) := \emptyset$
    create a new action level $N_O(n) := \emptyset$
    call $\mathsf{expand}(n+1)$
    $n := n+1$, $max := n$
endif
endloop

$\underline{\mathsf{expand}(n+1)}$

1.1 add no-ops:

```
for all facts f ∈ N_F(n)
    create a no-operator nop_f
    N_O(n)     := N_O(n) ∪ {nop_f} /* add no-op to new action level */
    E_P(n)     := E_P(n) ∪ {(f, nop_f)} /* update set of precondition edges */
    N_F(n+1)   := N_F(n+1) ∪ {f} /* add atom as effect to new fact level */
    E_A(n)     := E_A(n) ∪ {(nop_f, f, ∅)} /* update set of effect edges */
endfor
```

1.2. action level expansion and update of precondition edges:

```
compute set Σ(n) of all actions o that are applicable in level N_F(n) (Definition 13)
N_O(n) := N_O(n) ∪ Σ(n) /* update action nodes, one node per action in Σ(n) */
for all o ∈ Σ(n)
    for all preconditions p ∈ φ_0(o) /* p ∈ N_F(n) */
        E_P(n) := E_P(n) ∪ {(p, o)} /* update precondition edges */
    endfor
endfor
```

1.3. fact level expansion and update of effect edges:

```
for all o ∈ Σ(n) /* only actions other than no-ops */
    for all φ_i(o) ⇒ α_i(o) and all atomic effects e ∈ α_i(o)
        if φ_i(o) ⊆ N_F(n), all facts in φ_i(o) ⊆ N_F(n) are non-exclusive, and
          φ_i(o) is non-exclusive of φ_0(o) /* trivially satisfied for φ_i(o) = ∅ */
          then N_F(n+1) := N_F(n+1) ∪ {e} if e ∉ N_F(n+1)
                /* new nodes are only created for new facts */
                E_A(n) := E_A(n) ∪ {(o, e, φ_i(o))}
          else nothing /* ignore effects with unsatisfied conditions */
        endif
    endfor
endfor
for all o ∈ Σ(n) /* almost same loop to deal with DEL effects */
    for all φ_i(o) ⇒ δ_i(o) and all atomic effects e ∈ δ_i(o) with e ∈ N_F(n+1)
        if φ_i(o) ⊆ N_F(n) and the same conditions as above hold
          then E_D(n) := E_D(n) ∪ {(o, e, φ_i(o))}
          else nothing
        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)
```

The Add effects are processed first and added to the new fact level to make sure that the planner knows the complete set of facts nodes before the Del effects are processed because Del edges can only be established between existing nodes.

Given the following sets of actions $\{Op1, Op2, Op3\}$

Op1	Op2	Op3
pre: d1	pre: d2	pre: d3
eff: ADD a, DEL d1.	eff: ADD b, DEL d2;	eff: ADD c;
	y $\Rightarrow$ ADD x;	y $\Rightarrow$ ADD x;
	x $\Rightarrow$ DEL a.	z $\Rightarrow$ ADD y.

together with the initial state $I = \{d1, d2, d3, x, y, z\}$ and goals $G = \{a, b, c\}$, Figure 3 shows the planning graph that is generated until the goals are reached for the first time (no-ops are omitted).

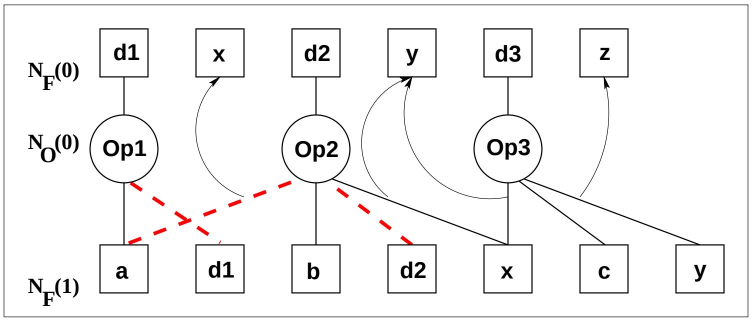


Figure 3: An Example Planning Graph. The delete edges are drawn with dashed lines and each conditional effect edge has a pointer to its effect condition. The three actions are non-exclusive because they interfere only over their conditional effects.

Planning graphs for the ADL subset inherit all properties of the original planning graphs as described in [2]: First, their size is polynomially restricted in their depths and the number of actions and facts in the initial state. Second, fact levels grow monotonically, i.e., if a fact f is contained in a level $N_F(n)$ it will also be contained in all $N_F(m)$ with $m > n$, which follows immediately from the use of no-ops. We state two more important properties in the following lemmas.

Lemma 2 *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 9). Thus there is at least one way of achieving both facts with non-exclusive actions and therefore they remain non-exclusive. ■

As a consequence of this lemma, it follows that the number of mutual exclusion relationships can only decrease, but never increase if two fact levels contain the same fact nodes. This allows us to incorporate the original "level-off" test into IP^2 that defines a simple and sufficient, but not necessary criterion that a planning problem has no solution.

Definition 15 *Let* $\mathsf{Mutex}(n)$ *be the set of all exclusive pairs at fact level* n*. We say that the planning graph has* leveled off *at level* n *iff* $|N_F(n)| = |N_F(n+1)|$ *and* $|\mathsf{Mutex}(n)| = |\mathsf{Mutex}(n+1)|$.

Lemma 3 *Every planning graph levels off after a finite number of expansions.*

Proof: Fact levels grow monotonically in the number of facts and mutual exclusion relationships cannot increase for identical fact levels. Furthermore, the actions we consider are deterministic (Definition 4) and each operator can only cause a finite number of effects. This is also valid for universally quantified effects since we assume a finite domain of discourse. Furthermore, every action that was applicable at a level n will also be applicable at level $n+1$. Thus, the number of facts that can be achieved with a fixed set of actions is finite. Consequently, since levels grow monotonically, the fact levels of each planning graph will converge to this finite set and after a finite number of identical fact levels, the number of mutual exclusion relationships cannot decrease any further and will reach a fixed point as well. ∎

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

Proof: The set of applicable actions in a state is uniquely defined, i.e., the same set of actions is applicable in identical states. If $|N_F(n)| = |N_F(n+1)|$ and the mutual exclusion relationships have not changed at both levels, the same sets of actions can be applied to both fact levels, i.e., the action levels $N_O(n)$ and $N_O(n+1)$ contain identical nodes. Thus $N_F(n+2) = N_F(n+1)$ and with induction over the action levels it follows that a fixed point has been reached. ∎

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.*

Proof: If the graph has leveled off at n it follows with Corollary 1 that all subsequent fact and action levels will contain identical nodes. No new atoms are added to future fact levels and the exclusion relationships between facts have already reached a fixed point, i.e., a missing goal atom will never be added and two exclusive goal atoms can never be made non-exclusive. ∎

4 Finding a Valid Plan in Planning Graphs

The first part of the planning algorithm is identical 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 the goal state holds already in the initial state. If the goals are unreachable, $\perp$ ("no solution exists") is returned. If they are already satisfied in the initial state, $\langle\rangle$ ("empty plan") is returned.

A. initial creation of planning graph
call the planning graph generation algorithm
returns $\Pi(N, E)$ or $\perp$ `/* shortest graph containing goals */`
max `/* the number of proposition levels generated */`
$\mathsf{Mutex}(n)$ `/* all exclusive pairs at level` n `*/`
if $max = 0$ then return $\langle\rangle$ and stop planning.
if $\Pi(N, E) = \perp$ then return $\perp$ and stop planning.

If none of the special cases applies, a recursive search algorithm $\mathsf{search}(n)$ 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 three points from its GRAPHPLAN predecessor:

1. the input, which is a pair of sets,
2. the selection procedure for actions at each level, which takes into consideration that an action can possibly achieve the same goal atom under different effect conditions, and
3. the resolution of conflicts caused by conditional effects.

The input consists of two sets of facts instead of only one: (1) the *goals* $\mathcal{G}_n$ that have to be achieved at level n by selecting actions at $N_O(n-1)$ and (2) so-called *negative goals* $\mathcal{C}_n$ that the selected actions have to avoid, i.e., no state in which $\mathcal{G}_n$ is established must entail $\mathcal{C}_n$.[8] If one of the negative goal atoms were established by the selected actions, effect conditions of undesired conditional effects would be made true leading to harmful interaction among actions that were selected to achieve the goals at level $n+1$.

[8]Note that the input is $\mathcal{G}_{max} = G$ and $\mathcal{C}_{max} = \emptyset$ for the initial call $\mathsf{search}(max)$.

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 */`

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

$\forall\, \mathcal{C}_{1 \leq n \leq max}$:= $\emptyset$ `/* negative goals at level` n, $\mathcal{C}_{max}$ `remains` $\emptyset$ `*/`

$\forall\, \Delta_n$:= $\emptyset$ `/* 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 3)

      then return $\perp$ and stop planning

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

    endif

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

endif

endloop

Given a set of goal atoms $\mathcal{G}_n \subseteq N_F(n)$ and a set of negative goals $\mathcal{C}_n \subseteq N_F(n)$, action selection proceeds as follows: First, a set of Add edges $\Delta_{n-1} \subseteq E_A(n-1)$ is selected at action level $N_O(n-1)$ that enable the atoms in the goal set $\mathcal{G}_n$. In contrast to GRAPHPLAN that selects an action, IP^2 selects a particular Add edge that is linked to the goal, because the same action can achieve a goal atom under different types of effects (conditional or not) or under different effect conditions. By selecting an edge, an action is chosen indirectly (marked as "used").

1. choice point: select set of Add edges Δ_{n-1}

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

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

    then $\Delta_{n-1} := \Delta_{n-1} \cup \{(o, g, \emptyset)\}$ and $\mathcal{G}_n := \mathcal{G}_n \setminus g$

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

    else select edge $(o, g, \varphi_i(o))$ of an action o such that

    a) already "used" actions at level n and o are non-exclusive of each other

    b) o does not unconditionally

      - delete a goal $g \in \mathcal{G}_n$ or effect condition of another selected edge

      - add a negative goal $c \in \mathcal{C}_n$

    c) no already "used" action unconditionally deletes an atom in $\varphi_i(o)$

  endif

endfor

The conditions a) to c) guarantee that the "used" actions are independent of each other wrt. their preconditions, unconditional effects, effect conditions of used conditional effects, and that no negative goal is unconditionally added.

Given the example in Figure 3 and the goals $\mathcal{G}_1 = \{a, b, c\}$ and $\mathcal{C}_1 = \emptyset$, the planner has as its only choice the edges $(Op1, a, \emptyset)$, $(Op2, b, \emptyset)$, and $(Op3, c, \emptyset)$. All conditions are satisfied, i.e., the actions are non-exclusive and no unconditional Del effects occur.

As a next step, IP^2 computes the goals at level $n-1$ based on the preconditions and effect conditions of the selected Add edges. If the resulting set $\mathcal{G}_{n-1}$ is mutually exclusive, the planner backtracks to a new choice of edges, because mutually exclusive goals can never be achieved in the same state.

2. compute goal set $\mathcal{G}_{n-1}$:
$\mathcal{G}_{n-1} := \bigcup_k \varphi_0(o_k) \cup \bigcup_k \varphi_i(o_k)$ with $(o_k, g, \varphi_i(o_k)) \in \Delta_{n-1}$
`/* note that` $\varphi_i(o_k) = \emptyset$ `for an unconditional effect */`
if $\mathcal{G}_{n-1}$ contains mutually exclusive facts
  then backtrack to Step 1
  else continue with Step 3
endif

In the small example from Figure 3, this step is trivial, because all selected edges are unconditional and the new goals $\mathcal{G}_0$ are obtained as the preconditions $\{d1, d2, d3\}$ of the actions.

Now, the action set can be tested for minimality because the test requires that the new goals $\mathcal{G}_{n-1}$ have to be known in order to determine all the atoms that are made true by a selected action. For each used action, all unconditional Add edges and all conditional Add edges are collected, whose effect conditions are completely contained in the new goals $\mathcal{G}_{n-1}$. This means, the planner considers all effects that will necessarily be made true when the new goals are achieved. 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.

Now, the planner can determine the new negative goals $\mathcal{C}_{n-1}$. This process is quite complex and comprises two main tasks:

1. The planner has to decide if negative goals from the set $\mathcal{C}_n$ are explicitly destroyed by actions at level $n-1$ or progressed through the set $\mathcal{C}_{n-1}$.

2. The selected actions at level $n-1$ are tested for interference caused by conditional effects. To prevent harmful conditional effects, atoms from the corresponding effect conditions are added to the new negative goals.

For the first task, the planner checks if there is a negative goal c in the set $\mathcal{C}_n$ that is non-exclusive of the new goals $\mathcal{G}_{n-1}$ and that is added by a no-op. This means that c could potentially hold in the state where $\mathcal{G}_{n-1}$ is achieved and that it could "survive" the execution of the selected actions at time step $n-1$ if the no-op is non-exclusive of the selected actions, i.e., there is no action in the set Δ_{n-1} that deletes c. The planner can deal with this potential conflict in two ways: either by adding c to the set of new negative goals $\mathcal{C}_{n-1}$ or by selecting a Del edge from a new or already used action to destroy c. In the first case, we obtain the first entry for the new negative goals. In the second case, the new goals have to be augmented by the preconditions (and possibly effect conditions) of the selected action. Note that this action needs to be a valid choice wrt. interference with already used actions as tested in Part1 of the search algorithm. To avoid the generation of non-optimal plans, progression is tried first and in the second case, Del edges of used actions are preferred.[9]

3. deal with negative goals $\mathcal{C}_n$:
determine the set $\mathcal{C}_n^-$ of negative goals in C_n that are non-exclusive of $\mathcal{G}_{n-1}$
$\mathbb{C} = \emptyset$`/* initialization */`
for all $c \in \mathcal{C}_n^-$
  if c is Add effect of a no-op
    then $\mathbb{C} := \mathbb{C} \cup \{c\}$
    else nothing
  endif
endfor
for all $c \in \mathbb{C}$ **choice point**:
  progress c as a new negative goal: $\mathcal{C}_{n-1} := \mathcal{C}_{n-1} \cup \{c\}$
  or
  select a Del edge from $E_D(n-1)$
  $\Delta_{n-1} := \Delta_{n-1} \cup \{(o, c, \varphi_i(o))\}$
  $\mathcal{G}_{n-1} := \mathcal{G}_{n-1} \cup \varphi_i(o) \cup \varphi_0(o)$
endfor

For the second task, the planner has to deal with conditional interference among actions that was totally ignored until now. The difference between the two types of effects is that unconditional effects always occur when an action is selected, while conditional effects additionally require the effect condition to hold. This gives the planner the possibility to prevent an undesired conditional effect by making sure that its effect conditions do not hold in the state in which the action is executed. But IP^2 does not know which specific linearization will be chosen for execution because it selects a set of "parallel" actions. Therefore, it has to address two tasks:

[9] One could also imagine to select an action that makes c exclusive because this action has preconditions which are exclusive of c. However, this is the same as forwarding c as a new negative goal.

1. By forming a set of negative goals 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.[10]

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

Four different possibilities for conditional interference have to be tested: A selected action is not allowed to conditionally delete a goal (an atom from the set $\mathcal{G}_n$), a precondition of another selected action (an atom from the set $\mathcal{G}_{n-1}$), an effect condition of a desired conditional effect (also an atom from $\mathcal{G}_{n-1}$), or to conditionally add a negative goal from the set $\mathcal{C}_n$.

If the planner discovers conditional interference, it checks if the effect conditions $\varphi_i(o)$ 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 $\varphi_i(o)$ 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. If actions at level $N_O(0)$ have harmful conditional effects, backtracking is also necessary because the effect conditions hold in the initial state, which cannot be altered by IP^2. On all levels $n \geq 1$, the set $\varphi_i(o) \setminus \mathcal{G}_{n-1}$ of effect condition atoms that are not in the new goal set is added to a set of sets $\mathbb{S}$ and a (not necessarily smallest) hitting set for $\mathbb{S}$ forms the set of new negative goals $\mathcal{C}_{n-1}$. This means, one element out of each set in $\mathbb{S}$ is selected to form $\mathcal{C}_{n-1}$, because a conditional effect is only achieved if the effect condition as a whole (set) is established, i.e., preventing one single atom per effect condition already avoids the undesired effect. Considering the set difference $\varphi_i(o) \setminus \mathcal{G}_{n-1}$ avoids that goals and negative goals overlap.

To illustrate why the set of new negative goals is a hitting set of the effect conditions let us consider the following abstract example where the planner has to prevent the conditional effects below:

$$\begin{array}{cl} x \wedge y \wedge z & \Rightarrow \text{ADD } a \text{ , ADD } b \\ u \wedge w & \Rightarrow \text{ADD } a \\ x \wedge v & \Rightarrow \text{ADD } b \end{array}$$

None of the conditional effects $a \wedge b$, a, and b can be achieved if at least one fact out of each conjunctive effect condition $x \wedge y \wedge z$, $u \wedge w$, and $x \wedge v$ is prevented. This leads to 12 different alternatives that have to be explored by the planner to achieve completeness: $\{x,u\}$, $\{x,w\}$, $\{x,u,v\}$, $\{x,w,v\}$, $\{y,u,x\}$, $\{y,u,v\}$, $\{y,w,x\}$, $\{y,w,v\}$, $\{z,u,x\}$, $\{z,w,v\}$, $\{z,w,x\}$, $\{z,u,v\}$. We do not focus on computing smallest hitting sets, because this would make the planner

[10] Notice that a separate set of negative goals is necessary because we do not have explicit atomic negation. With negation, one would simply add the negated goal atom to the set of goals. The reasoning tasks to deal with negated goals, however, will be the same as for the negative goals.

incomplete. For example, if $\{a,b\}$ and $\{o,p,q\}$ are two possible hitting sets, then also the larger second set needs to be considered, because the planner could fail in preventing the first. But the planner memoizes all negative goal sets that have been unsuccessful at a given level n, i.e., that could not be prevented when selecting actions at step level $N_O(n-1)$. Clearly, any superset of an unsuccessful negative goal set does not need to be considered during backtracking and can be skipped immediately. For example, if the planner cannot prevent the negative goals $\{x,u\}$, then it will not be able to prevent the negative goals $\{x,u,v\}$, $\{y,u,x\}$, or $\{z,u,x\}$.

In the example from Figure 3, the planner finds out that $Op2$ deletes the goal a under condition x, which becomes the first new negative goal, i.e., $\mathcal{C}_0 = \{x\}$. Now the planner has to resolve the "linearization problem" by making sure that no selected action adds any of the new negative goals in $\mathcal{C}_0$. The planner checks if any used actions other than $Op2$ add x, which is the case for $Op3$ that adds x under condition y. This means, if $Op3$ is executed before $Op2$ in a state in which y holds, the condition x becomes true and $Op2$ will delete the goal a. Therefore, y must be added as a new negative goal to make sure that this effect is prevented as well. It does not matter that $Op2$ adds x under condition y, because the effect x of $Op2$ will always hold in the next state after the condition x for the delete effect $x \Rightarrow$ DEL a of $Op2$ has been evaluated. If the effect conditions were a set of atoms instead of a singleton, one of the atoms would be selected as a new negative goal, i.e., the planner has a choice point. Since $\mathcal{C}_0$ has been extended, the whole test is repeated with $\mathcal{C}_0 = \{x,y\}$. Fortunately, only $Op3$ can add the effect condition y of its own harmful conditional effect and this does not matter. Thus, a fixed point is reached and all effect conditions of effects that potentially add new negative goals are already prevented by $\mathcal{C}_0$.[11]

4. compute negative goals $\mathcal{C}_{n-1}$
  for all actions o marked as "used" in $N_O(n-1)$
    if exists an edge $(o, g, \varphi_i(o)) \in E_D(n-1)$ with $g \in \mathcal{G}_n$ or $g \in \mathcal{G}_{n-1}$
      `/* goal, precondition, or effect condition threatend */`
      or $(o, g, \varphi_i(o)) \in E_A(n-1)$ and $g \in \mathcal{C}_n$
      `/* negative goal could be made true */`
      then add $\varphi_i(o) \setminus \mathcal{G}_{n-1}$ to the set of sets $\mathbb{S}$
      else nothing
    endif
  endfor
  if $\emptyset \in \mathbb{S}$ (exists $\varphi_i(o) \subseteq \mathcal{G}_{n-1}$) or $[n = 1$ and $\mathbb{S} \neq \{\emptyset\}]$ (actions at level $N_O(0)$)
    then backtrack
    else **choice point:** choose $\mathcal{C}_{n-1}$ as a hitting set for $\mathbb{S}$
  endif

[11] Since $\mathcal{C}_0 \neq \emptyset$, the planner would fail on extracting a valid plan from this graph, because negative goals cannot be eliminated from the fixed initial state.

Fixpoint Iteration: complete $\mathcal{C}_{n-1}$ by effect conditions
of used actions that add atoms in $\mathcal{C}_{n-1}$
if $\mathcal{G}_{n-1}$ and $\mathcal{C}_{n-1}$ are disjoint
then invoke $\mathsf{search}(n-1)$
else backtrack
endif

The completion process fails if an effect condition is completely contained in $\mathcal{G}_{n-1}$, because this would make $\mathcal{G}_{n-1}$ and $\mathcal{C}_{n-1}$ overlap (a relevant fact had to be made true and false at the same time), or if an action unconditionally adds a new negative goal. This is in fact the same backtracking condition as on the set $\mathbb{S}$, i.e., the planner will not be able to prevent the effect condition, which means that under certain linearizations of the action set, a "cascade"-like chain of harmful side effects will occur that is enabled by the effect condition. During backtracking, all choices of new negative goals (i.e., hitting sets) are tried before a different choice of Δ_{n-1} becomes necessary.

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 one of the sets $\mathcal{G}_n$, $\mathcal{C}_n$ by selecting different actions at level n because it has proven the pair to be unsolvable. The pair $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$ is memoized as unsuccessful at fact level n, see the Appendix, Section 9..

Backtracking at the maximum level of the graph is not possible and leads to a failure of $\mathsf{search}(n)$ 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 3 in Section 6.

5 An Example

Given the 3 operators from the Briefcase domain as shown in Figure 4, the planner has to achieve the goal *at(o,l)* and *at-b(m)* starting in the initial state

$$I = \{\ in(o),\ at(o,l),\ at\text{-}b(l)\ \}.$$

Figure 5 shows the complete planning graph that is generated for the example.[12] In action level $N_O(0)$ of the graph, the two actions *take-out(o)* and *move(l,m)* are applicable, but they are mutual exclusive because *move(l,m)* deletes the precondition *at-b(l)* of the *take-out* action. The *move* action also deletes the

[12]Delete edges are marked with dashed (red) lines, goals have a light gray (green) frame and negative goals have a dark gray (red) frame.

```
move
:v ?m ?l location
:p at-b(?m)
:e ADD at-b(?l) DEL at-b(?m);
ALL ?o object in(?o) => ADD at(?o ?l) DEL at(?o ?m).

take-out
:v ?o object ?l location
:p in(?o) at-b(?l)
:e ADD not-in(?o) DEL in(?o).

put-in
:v ?o object ?l location
:p not-in(?o) at(?o ?l) at-b(?l)
:e ADD in(?o) DEL not-in(?o).
```

Figure 4: Briefcase Operators in IP^2 Input Syntax

subgoal *at(o,l)* because the object *o* is in the briefcase in the initial state, i.e., the two subgoals are exclusive at level $N_F(1)$. Therefore, IP^2 adds another action and fact level and tries to achieve the goals at fact level $N_F(2)$.

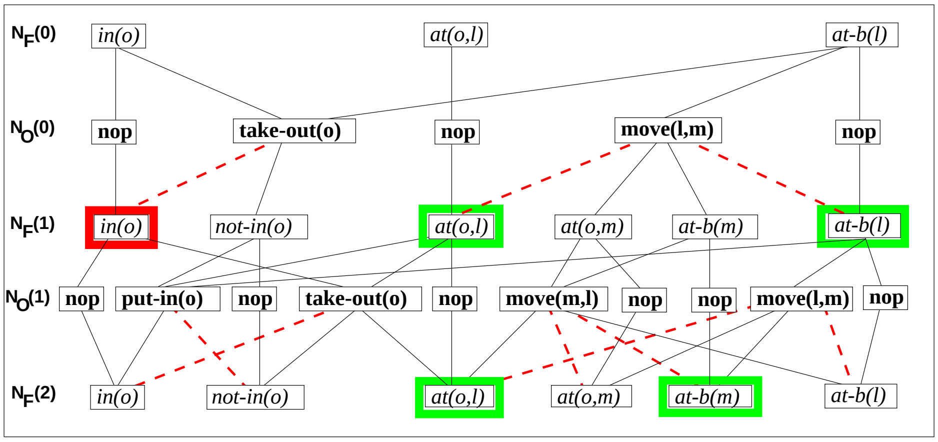


Figure 5: Planning Graph for the Briefcase Example

Given the goals *at(o,l)* and *at-b(m)* at level $N_F(2)$, IP^2 has several choices to achieve them. The first subgoal can be achieved using the *take-out* action, a no-op, or the action *move(m,l)*. The second subgoal can be achieved using either *move(l,m)* or a no-op. We only consider the choice that will lead to a valid plan, i.e., the no-op $nop_{at(o,l)}$ and the action *move(l,m)*. The selected action set is minimal. After checking that no unconditional interference occurs between the selected actions, the planner determines the preconditions of both actions with *at(o,l)* from the no-op and *is-at(l)* from the move-action as the new goals $\mathcal{G}_1$. Finally, IP^2 checks for conditional interference and finds out that *move(l,m)* deletes the subgoal *at(o,l)* because of the effect condition *in(o)* in level $N_F(1)$. This condition becomes the negative goal $\mathcal{C}_1 = \{in(o)\}$.

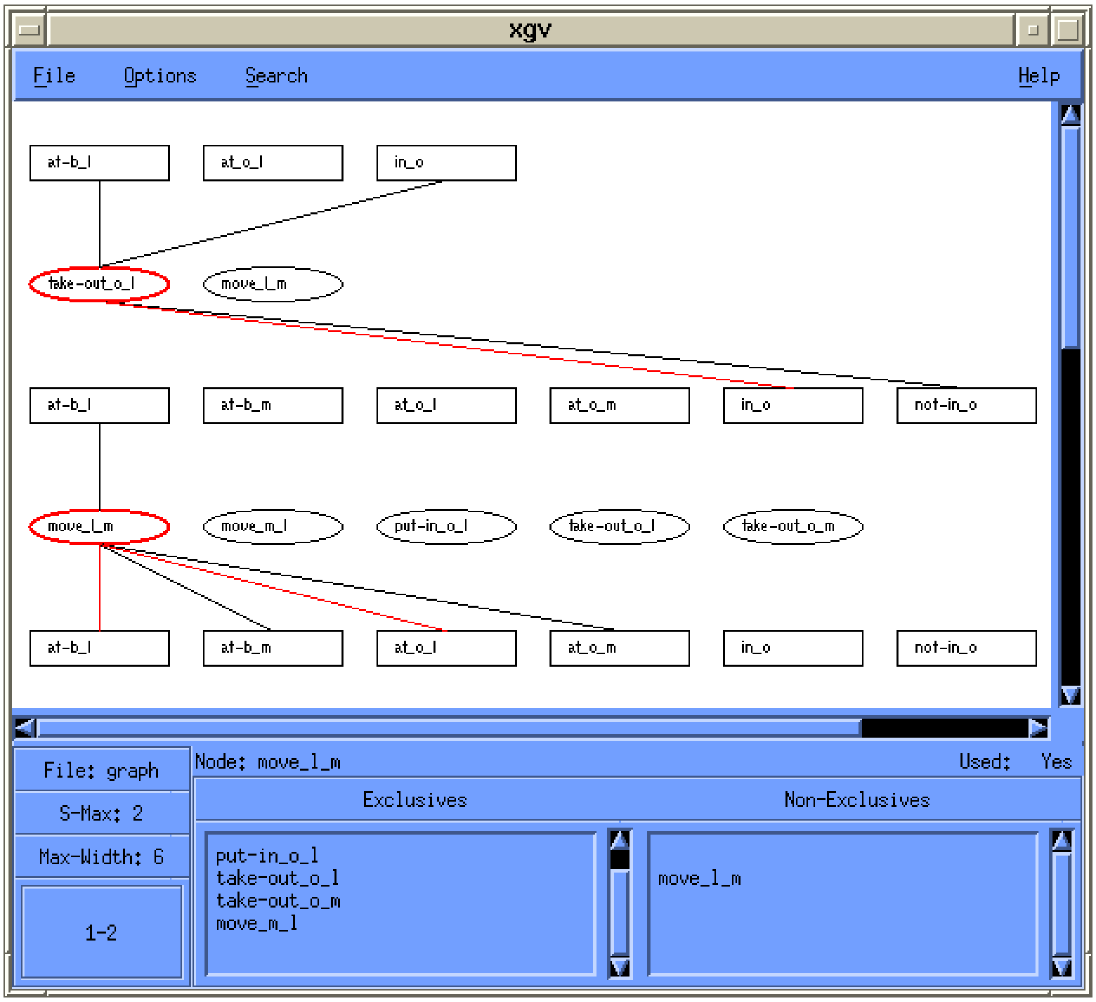


Figure 6: The Plan in the Example Planning Graph

To achieve the goals $\mathcal{G}_1$, only two no-ops can be selected at $N_O(0)$, because there are no other actions at this level that add the goals. Now, IP^2 tests if the negative goal *in(o)* is non-exclusive from the goals and if it is an Add effect of a no-op. This is indeed the case and the planner has to make this no-op exclusive. Progressing the negative goal to the initial state is impossible, thus IP^2 selects *take-out(o)* as the only possible action, which deletes the condition *in(o)* and guarantees that moving the briefcase in the next time step does not delete the goal *at(o,l)*.

Figure 6 shows the plan that is extracted from the planning graph as it is generated for the example and drawn in the graphical interface XGV [14]. No-ops are not shown to reduce the width of action levels.

6 Termination, Soundness, Completeness, and Optimality

To speed up search, the original GRAPHPLAN system *memoizes* at each proposition level *goal sets* that could not be achieved. If such a memoized goal set is tried again, one knows immediately that the goal set cannot be achieved. In addition to making the search more efficient, memoizing enables us to detect unsolvability of a planning problem [3, Theorem 3], i.e., it guarantees *termination*.

Of course, also a goal set $\mathcal{G}_n$ that is a superset of a memoized unsuccessful goal set $\mathcal{G}$ cannot be achieved, and one can immediately backtrack at this point. Although this obvious extension is not mentioned in the paper describing GRAPHPLAN [3], the system contains an option that enables such *subset memoization* (in a limited way). Unfortunately, with subset memoization, the termination check may not be sound any longer because the proof of Blum and Furst's [3] Theorem 3 does not apply to subset memoization. For this reason, the termination check in GRAPHPLAN is disabled when the subset-memoization option is enabled.

We begin our proofs of theoretical properties of IP^2 with the proof that even with subset memoization it is possible to detect unsolvability of a planning problem. Moreover the same termination check as used with plain memoization remains valid. In other words, it is safe to enable GRAPHPLAN's termination check even if subset memoization is enabled.

In the proof, we consider goal sets, but the results immediately apply to the pairs of goals/negative goals used by IP^2.

Let U_n^{max} be the set of unsuccessful goal sets at level n after an unsuccessful stage max. Under the plain memoization strategy we have for all levels n with $0 \leq n \leq max$

$$U_n^{max} \supseteq U_{n+1}^{max}, \tag{1}$$

since GRAPHPLAN tries to achieve a goal with *no-ops* first. Further, if m denotes the level where the graph has *leveled off*, then we know that $U_{m+1}^{max} = U_m^{max-1}$. Finally, if we detect that $U_m^{max-1} = U_m^{max}$, we can conclude that the goals are unachievable [3, Theorem 3].

Let T_n^{max} be the unsuccessful goal sets memoized under the *subset memoization* strategy. Since some unsuccessful goal sets are not memoized under this strategy, it follows immediately that

$$T_n^{max} \subseteq U_n^{max} \tag{2}$$

after an unsuccessful stage max for all levels $n \leq max$.

Furthermore, we do not necessarily have $T_n^{max} \supseteq T_{n+1}^{max}$,i.e., for the subset-memoization strategy, Relation (1) does not hold. In order to see this, assume that we have a memoized goal set $\mathcal{G}$ on level $n+1$, i.e., $\mathcal{G} \in T_{n+1}^{max}$. One way to achieve this goal could be to select *no-op* actions, which leads to the goal set $\mathcal{G}$ on level n. Under the plain memoization strategy, $\mathcal{G}$ would be memoized on level n, from which one can deduce that $U_n^{max} \supseteq U_{n+1}^{max}$. Under the subset memoization strategy, however, it may be the case that $\mathcal{G}$ is not memoized on level n because there is already a smaller set $\mathcal{F} \in T_n^{max}$ such that $\mathcal{F} \subseteq \mathcal{G}$.

Since Relation (1) does not hold for the subset-memoization strategy, the condition $T_m^{max-1} = T_m^{max}$, where m is the level-off index, does not guarantee that the planning problem is unsolvable. Changing perspective, we will analyze the set of *excluded goal sets* generated by the plain memoization method $\hat{U}_n^{max}$, which is defined as:

$$\hat{U}_n^{max} = \{U \subseteq P \mid U \supseteq \mathcal{G}, \mathcal{G} \in U_n^{max}\} \tag{3}$$

Similar to the set of excluded goal sets generated by the plain memoization method, we define the set of excluded goal sets generated by the subset-memoization strategy:

$$\hat{T}_n^{max} = \{U \subseteq P \mid U \supseteq \mathcal{G}, \mathcal{G} \in T_n^{max}\} \tag{4}$$

From those definitions it follows immediately that the set of excluded goal sets generated by plain memoization is always a superset of the set of excluded goal sets generated by subset memoization.

Proposition 1 *After all unsuccessful stages max and for all levels $n \leq max$, we have $\hat{T}_n^{max} \subseteq \hat{U}_n^{max}$.*

We now show that the subset memoization strategy computes the same sets of excluded goal sets as the plain memoization strategy, i.e., $\hat{T}_n^{max} = \hat{U}_n^{max}$. In order to do so, we first need to define the notion of *generated goal sets*. We write $\mathsf{gen}_n(\mathcal{G})$ for the set of goal sets GRAPHPLAN generates in level $n-1$ from the goal set $\mathcal{G}$ on level n. It is obvious that goal-set creation has a monotonicity property as spelled out below.

Proposition 2 *Let $\mathcal{F}$ and $\mathcal{G}$ be goal sets on level n such that $\mathcal{F} \subseteq \mathcal{G}$. Then for each $\mathcal{G}' \in \mathsf{gen}_n(\mathcal{G})$ there exists a goal set $\mathcal{F}' \in \mathsf{gen}_n(f)$ such that $\mathcal{F}' \subseteq \mathcal{G}'$.*

Theorem 2 *For all unsuccessful stages max and for all levels $n \leq max$, we have $\hat{U}_n^{max} = \hat{T}_n^{max}$.*

Proof sketch:
"$\supseteq$": follows from Proposition 1.
"$\subseteq$": Proof by induction over n with values ranging from max down to 0 for arbitrary max.
Base step: For $n = max$ the claim is trivially true because $U^{max}_{max} = T^{max}_{max} = \{G\}$ with G being the original goals of the planning problem.
Induction step: Assume the claim is true for all n such that $0 < n_0 \leq n \leq max$. We will show that it also holds for $n = n_0 - 1$.

Suppose that $\hat{U}^{max}_{n_0-1} \not\subseteq \hat{T}^{max}_{n_0-1}$. This implies that there is a goal set $\mathcal{G} \in U^{max}_{n_0-1}$ such that there is no $\mathcal{F} \in T^{max}_{n_0-1}$ with $\mathcal{F} \subseteq \mathcal{G}$. On the other hand, for all $\mathcal{H} \in U^{max}_{n_0}$ there exists a goal set $\mathcal{H}' \in T^{max}_{n_0}$ such that $\mathcal{H}' \subseteq \mathcal{H}$ because of the induction hypothesis $\hat{U}^{max}_{n_0} \subseteq \hat{T}^{max}_{n_0}$. Since $\mathcal{G}$ was produced by some goal set $\mathcal{H} \in U^{max}_{n_0}$, i.e., $\mathcal{G} \in \mathsf{gen}_{n_0}(\mathcal{H})$, there must be a goal set $\mathcal{H}' \in T^{max}_{n_0}$ such that $\mathcal{H}' \subseteq \mathcal{H}$. By Proposition 2 it follows that there exists a goal set $\mathcal{G}'$ with $\mathcal{G}' \subseteq \mathcal{G}$ and $\mathcal{G}' \in \mathsf{gen}_{n_0}(\mathcal{H}')$.

Now it could be the case that $\mathcal{G}' \in T^{max}_{n_0-1}$. In this case we would have $\mathcal{G}' \in \hat{T}^{max}_{n_0-1}$ and therefore $\mathcal{G} \in \hat{T}^{max}_{n_0-1}$, which is a contradiction. If $\mathcal{G}' \not\in T^{max}_{n_0-1}$, then the reason must be that there is already a goal set $\mathcal{G}'' \in T^{max}_{n_0-1}$ with $\mathcal{G}'' \subseteq \mathcal{G}'$ and $\mathcal{G}'$ was ignored because of subset memoization. However, because $\mathcal{G}'' \subseteq \mathcal{G}$, we also have $\mathcal{G} \in \hat{T}^{max}_{n_0-1}$, which is again a contradiction. For this reason, it is impossible that $\hat{U}^{max}_{n_0-1} \not\subseteq \hat{T}^{max}_{n_0-1}$ and the claim holds for $n_0 - 1$ as well, which proves the theorem. ■

The arguments in the proof of Theorem 3 in [3] are all valid for the sets of excluded goal sets, which we spell out below.

Lemma 4 *A planning problem is unsolvable if the graph has leveled off on level* m *and* $\hat{U}^{max}_m = \hat{U}^{max+1}_m$.

Of course, detecting this might be computationally difficult. However, we can, of course, use the "representation" of the excluded sets in terms of U^{max}_m or T^{max}_m and test those for equality.

Theorem 3 *Let the planning graph for a planning problem* $\mathcal{P}(\mathcal{O}, D, I, G)$ *be leveled off at some level* m. *Let* $max \geq m$ *be the current maximum fact level. The problem* $\mathcal{P}(\mathcal{O}, D, I, G)$ *is unsolvable if and only if extending the planning graph to level* $max + 1$ *and searching the extended graph leads to the same number of unsuccessful goals at level* m *under the subset memoization strategy, i.e.,*

$$|T^{max}_m| = |T^{max+1}_m|.$$

T is representing pairs of goal/negative goal sets for IP^2.

We continue with proving the soundness of IP^2.

Theorem 4 (Soundness) *Let $\mathcal{P}(\mathcal{O}, D, I, G)$ be a planning problem. If IP^2 returns a parallel plan $\mathcal{P} = \langle Q_1, Q_2, \ldots Q_n\rangle$ then*

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

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

Proof:

To prove soundness we rely on the semantics as developed in Definition 6 and proceed by induction over the length of the sequence of action sets.

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

We have to show that $R(I, \langle\rangle)$ is defined and that the goals G hold in all possible resulting world states. With Definition 6 we obtain $R(I, \langle\rangle) = I$, i.e., $I \supseteq G$. Indeed, in Part A of the planning algorithm, $\mathcal{P} = \langle\rangle$ is returned only in this case, since otherwise a non-empty set of actions will always be selected.

induction step: $\mathcal{P} = \langle Q_1, Q_2, \ldots Q_n\rangle$

With Definition 6 we obtain for the recursive case

$$\bigcap R(R(I, \langle Q_1, Q_2, \ldots, Q_{n-1}\rangle), \langle Q_n\rangle) \supseteq G \tag{5}$$

with the induction hypothesis being that $R(I, \langle Q_1, Q_2, \ldots, Q_{n-1}\rangle) = \mathcal{S}$ is defined and that

1. $\mathcal{G}_n$, i.e., the goals at time step n hold in all resulting states $S \in \mathcal{S}$,
2. $\mathcal{C}_n$, i.e., the negative goals at time step n do not hold in any of the resulting states $S \in \mathcal{S}$.

Simplifying (5) to $R(\mathcal{S}, \langle Q_n\rangle)$ and applying Definition 6 to it, we obtain

$$R(\mathcal{S}, \langle Q_n\rangle) = \begin{cases} \{T \subseteq P \mid S \in \mathcal{S}, q \in Seq(Q_n), T = Res(S, q)\} & \text{if } Res(S, q) \text{ defined} \\ & \forall\, S \in \mathcal{S}, q \in Seq(Q_n) \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{6}$$

According to the induction hypothesis we know for any $S \in \mathcal{S}$ that $\mathcal{G}_n \subseteq S$ and $\mathcal{C}_n \not\subseteq S$ hold. Let us consider an arbitrary such S and $q = \langle q_1, \ldots, q_n\rangle$ in the following. To $Res(S, q)$ in (6) we can apply Definition 4 and obtain

$$Res(S, \langle q_1, \ldots, q_n\rangle) = Res(Res(S, \langle q_1, \ldots, q_{n-1}\rangle), \langle q_n\rangle) \tag{7}$$

with the base case defined as

$$Res(S,\langle q\rangle) = \begin{cases} \big(S \cup A(S,q)\big) \setminus D(S,q) & if\, \varphi_0 \subseteq S \\ \text{undefined} & \text{otherwise} \end{cases} \tag{8}$$

where

$$A(S,q) = \bigcup_{\varphi_i \subseteq S, i \geq 0} \alpha_i \qquad \text{and} \qquad D(S,q) = \bigcup_{\varphi_i \subseteq S, i \geq 0} \delta_i$$

We continue by induction over the length of q. For the base case, let us assume that q contains only the single action q_0.[13] According to the graph construction algorithm, the planner chooses q_0 such that all goal atoms in $\mathcal{G}_{n+1} = G$ are added by this action.

$$G \subseteq A(S, q_0) \tag{9}$$

holds, because $\mathcal{G}_n \subseteq S$ (induction hypothesis) and $\mathcal{G}_n$ is build from the preconditions $\varphi_0(q_0)$ and all effect conditions of selected conditional effects $\varphi_i(q_0)$ of the action q_0. We have to show that

$$G \subseteq \big(S \cup A(S,q_0)\big) \setminus D(S,q_0) \tag{10}$$

which requires to prove that $G \cap D(S,q_0) = \emptyset$ because of (9). We proceed indirectly and assume that $G \cap D(S,q_0) \neq \emptyset$, i.e., one of the following conditions has to hold:

1. q_0 unconditionally deletes a goal from G: $\exists\, g \in G : g \in \delta_0(q_0)$ or
2. q_0 conditionally deletes a goal from G and the effect condition is satisfied: $\exists\, g \in G : g \in \delta_i(q_0)$ and $\varphi_i(q_0) \subseteq S$

Both cases lead to a contradiction. Case 1 is impossible because of the action selection in Part 1 of search algorithm (test condition b), i.e., q_0 would never be a choice by the planner. Case 2 is impossible because of Part 4 of the search algorithm, where $\varphi_i(q_0)$ would be added to $\mathbb{S}$ and used to construct the set $\mathcal{C}_n$. Since $\mathcal{C}_n$ does not hold in S (induction hypothesis), $\varphi_i(q_0)$ cannot hold either. Thus, we have $G \cap D(S,q_0) = \emptyset$ and with that $Res(S,q_0) \supseteq G$.
Now let us consider the general case that $q = \langle q_1, q_2, \ldots, q_n\rangle$ as an arbitrary, but fixed linearization of Q_n. With $\mathcal{G}_n \subseteq S$, we know that for all o_i holds $\varphi_0(q_i) \subseteq S$, which follows from the construction of $\mathcal{G}_n$. Furthermore, we have $G \subseteq \bigcup_{i=1}^{n} A(S,q_i)$. It remains to show that

[13] Assuming the empty set as base case does not make sense, since the planner returns only non-empty sets of actions at each time step.

1. $\mathcal{G}_n$ holds after execution of each action q_i, i.e., the preconditions and effect conditions remain valid after each action,

2. $\mathcal{C}_n$ remains invalid after each action,

3. no goal in G is made invalid by any of the actions in the sequence.

Each of the conditions is explicitly tested by the planner. Unconditional deletion of a goal or new goal and unconditional addition of a negative goal discard an action from the choice. Conditional deletion/addition lead to an extension of the set $\mathcal{C}_n$ with the planner repeating the test on the extended $\mathcal{C}_n$ (fixed point iteration of the planner on negative goals in Part 4 of the algorithm).

Thus, we can conclude that $G \subseteq Res(S, q)$ for all linearizations q and all states $S \in \mathcal{S}$ satisfying the conditions from the induction hypothesis. It follows that $\bigcap R(I, \langle Q_1, Q_2, \ldots, Q_n \rangle) \supseteq G$ holds.

■

Completeness follows from soundness and the fact that IP^2 terminates on unsolvable planning problems under the subset memoization strategy.

Theorem 5 *If* IP^2 *returns* $\perp$ *then* $\mathcal{P}(\mathcal{O}, D, I, G)$ *has no solution.*

Proof: Follows from the proofs of Theorems 3, 1, and 4. ■

Unfortunately, we cannot prove that IP^2 generates shortest plans in all cases according to our execution semantics. The following counter example illustrates a subtle interference among conditional effects that is interpreted as a conflict by the planner. Given the two actions

o_1	o_2
pre: $\emptyset$	pre: $\emptyset$
eff: ADD b	eff: ADD d
$c \Rightarrow$ ADD e, DEL c.	$c \Rightarrow$ ADD e, DEL c.

together with the initial state $I = \{c\}$ and goal $G = \{b, d, e\}$, IP^2 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, we argue that trading optimality for efficiency is a reasonable decision in such situations.

7 Empirical Evaluation

During the empirical evaluation of IP^2 we first investigated if the extension to a more expressive operator language causes any computational overhead. Therefore, we tested the system on the examples that are available in the GRAPHPLAN test suite.[14] The test suite ranges over several artificial and natural domains and the only domains that we did not encode for IP^2 are the artificial test domains by Barrett and Weld, because it was rather tedious and we did not expect any results different from those obtained for the other domains. The results for the example suite are shown in Figure 7. The last column shows the runtime of IP^2 in per cent relative to GRAPHPLAN's runtime when set to 100.

	GRAPHPLAN			IP^2				
Problem	Graph	Planning	Total	Graph	Planning	Total	%	
tyre-fixit	0.07	0.03	0.10	0.06	0.04	0.10	100	
blocks-shuffle	0.31	0.18	0.49	0.25	0.17	0.42	86	*
blocks-impossible	0.05	0	0.05	0.04	0	0.04	80	
mblocks	0.41	0.26	0.67	0.37	0.24	0.61	91	*
rocket-8	0.08	0	0.08	0.07	0	0.07	88	
rocket-10	0.10	0	0.10	0.08	0	0.08	80	
monkey-1	0.06	0	0.06	0.05	0	0.05	83	
monkey-2	0.16	0.06	0.22	0.15	0.08	0.23	105	
monkey-3	0.21	0	0.21	0.19	0	0.19	90	
TSP-complete	0.04	3.27	3.31	0.05	2.93	2.98	90	*
TSP-peterson	0.03	0.21	0.24	0.05	0.26	0.31	129	*
fridge-1	0.41	0	0.41	0.15	0	0.15	37	
fridge-2	0.87	0.26	1.13	0.23	0.05	0.28	25	*
link-10	0.03	0	0.03	0.03	0	0.03	100	
link30	0.22	0	0.22	0.24	0.02	0.26	118	
logistics	0.33	0	0.33	0.38	0	0.38	115	
logistics-4	0.41	0	0.41	0.42	0	0.42	102	
logistics-4pp	0.41	0	0.41	0.38	0	0.38	93	
logistics-6	0.59	0	0.59	0.61	0	0.61	103	
logistics-6pp	0.59	0	0.59	0.54	0	0.54	92	
logistics-6h	0.58	2.84	3.42	0.57	2.93	3.50	102	*
logistics-8	0.81	0	0.81	0.72	0	0.72	89	

Figure 7: IP^2 versus GRAPHPLAN in simple STRIPS domains

For both systems, we measured the effort for the initial graph creation until the goals are reached for the first time and the effort for interleaved search and

[14] IP^2 is implemented in C and available at http://www.informatik.uni-freiburg/~koehler/ipp.html. The current distribution also includes an improved graphical interface to analyse planning graphs. All runtimes are given in seconds of CPU time on a SPARC Ultra 1/170 if not indicated otherwise and were obtained by 5 runs of each system, i.e., we show the average runtime for each planning task. We added a CPU time measuring function to GRAPHPLAN to obtain comparable measures and used IP^2 2.0 in the tests.

graph expansion with activated subset memoization. It turned out that most of the examples in GRAPHPLAN's test suite do not require significant search. We marked those examples in the table with * that required some search effort during planning and in which we expected to find a computational overhead. As it turned out, IP^2 is even faster on these examples with the exception of TSP-peterson, which might be caused by the representational trick that GRAPHPLAN uses to encode the graph structure, cf. the sample distribution of GRAPHPLAN.

The reason why IP^2 can outperform GRAPHPLAN lies in implementational details of both systems. We have not simply extended GRAPHPLAN's code, but made significant changes where we thought to have a more efficient solution. First, we use a different algorithm for subset memoization, cf. the Appendix. Second, when building planning graphs, GRAPHPLAN determines the operators that are applicable at a given level and then computes their ground instances (actions) on "demand". Internally, this requires very costly type conversions between the atoms in fact levels and the preconditions of actions because of different data types. IP^2 reads in operators and the planning problem, "pre"-computes all possible ground instances and uses the same data type for fact levels and preconditions in the actions in order to avoid the conversion overhead, but it might compute actions that are never applicable. For most of the examples in the test suite, this difference seems to pay off during the initial graph expansion phase. Finally, with subset memoization GRAPHPLAN switches the termination test off, while IP^2 always performs the test, i.e., it does more computations, which could also explain why IP^2 is slower on some examples.

In general, the test suite did not deliver a clear picture and we argue that the examples are too small to deliver clear evidence of a computational overhead. It seems likely that on small examples an overhead in search is compensated by savings during subset memoization. To get a clearer picture, we ran both systems on SATPLAN examples taken from [8] with and without subset memoization yielding the following results, cf. Table 8.

Problem	GP	GP+PartialSubset	IP^2	IP^2+UBTree
bw_large.a	2.00	1.68	1.62	1.56
bw_large.b	322.93	243.71	244.02	128.92
logistics.a	1700.98	1465.75	2179.83	1701.97
logistics.b	656.67	601.82	1088.73	858.38
rocket_ext.a	114.35	120.45	195.90	281.98
rocket_ext.b	355.08	650.23	564.11	700.03

Figure 8: IP^2 versus GRAPHPLAN in SATPLAN domains

The Satplan examples indicate a computational overhead on large examples, i.e., IP^2 is always slower than GRAPHPLAN with or without subset memoization except in the blocks world. Interestingly, in the rocket domain, both planners show decreasing performance under subset memoization, which can be explained by the ratio between the effort to memoize goal sets and the utility that a goal set is eliminated by relating it to a previously unsolvable subset. For example, in bw_large.b IP^2 inserts 15,777 memo entries and has 6,681 successful memo lookups (a ratio of 2.4 to 1). In the rocket domain on example rocket_ext.a, much more effort is spent in inserting memo entries (583,151), but only a small number of them leads to successfull look-ups (15,912), which results in a ratio of 36.6 to 1. For example rocket_ext.b, a ratio of 39.4 to 1 is obtained.

We also investigated the influence of the different memoization algorithms by running IP^2 with the original GRAPHPLAN algorithms and the UBTree algorithm. The test examples show that UBTree is better than GRAPHPLANs complete subset check. It is, although not to such a great extent, better than the partial subset lookup technique implemented in GRAPHPLAN, cf. Figure 9.

domain	IP^2 with UBTree	IP^2 with GPcomplete	IP^2 with GPpartial
tyre-fixit	0.11	0.11	0.11
blocks-shuffle	0.50	0.50	0.50
blocks-large-b	125.86	-	133.49
brief-shuffle4	1.03	1.27	1.10
brief-t4	1.41	2.82	1.76
brief-shuffle5	12.46	91.93	14.34
brief-t5	106.68	-	209.75

Figure 9: Runtimes in seconds on a SUN Ultra 1/170 station for some standard examples with different memoizing techniques. GPcomplete is short for GRAPHPLAN's complete subset check and GPpartial is short for GRAPHPLAN's partial subset check, i.e., pairs $\langle \mathcal{G}'_n, \mathcal{C}'_n \rangle : \mathcal{G}'_n \subseteq \mathcal{G}_n, \mathcal{C}'_n \subseteq \mathcal{C}_n$ with $|\mathcal{G}'_n| = |\mathcal{G}_n| - 1$.

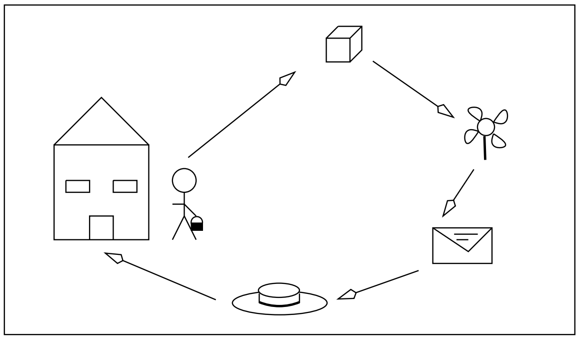
Experience in running IP^2 on problems from different domains shows that the UBTree memoization strategy is as fast as GRAPHPLAN's partial one on examples when there are not many subsets to find; the more often memoized subset pairs are found and the bigger the examples grow, the better UBTree memoization performs compared to GRAPHPLAN's partial look up technique. Figure 10 shows the number of memoizing hits for both strategies on the test examples.

	IP^2 with UBTree		**IP^2 with GPpartial**	
domain	simple	subset	simple	subset
tyre-fixit	54	1	55	0
blocks-shuffle	19	2	21	0
blocks-large-b	ca. 16,000	ca. 7,000	ca. 18,000	ca. 6,500
brief-shuffle4	1,760	812	2,280	946
brief-t4	4,541	1,791	6,817	1,331
brief-shuffle5	ca. 20,000	ca. 14,000	ca. 27,000	ca. 16,000
brief-t5	ca. 340,000	ca. 170,000	ca. 630,000	ca. 220,000

Figure 10: Number of memoizing hits for some standard examples

Simple means the number of times the planner found that a pair $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$ had been memoized already, *subset* means the number of times the planner found that a pair $\langle \mathcal{G}'_n, \mathcal{C}'_n \rangle : \mathcal{G}'_n \subseteq \mathcal{G}_n, \mathcal{C}'_n \subseteq \mathcal{C}_n$ had been memoized already. In the case of the GPpartial technique, $|\mathcal{G}'_n| = |\mathcal{G}_n| - 1$ must hold.

The next interesting question is how IP^2 compares to other planners that support universally quantified and conditional effects. We used the Briefcase domain to compare IP^2 to UCPOP [13], Prodigy [5], and GRAPHPLAN using the equivalent translation of operators into sets. We looked at the simple task of initially having the briefcase at home and several objects in different locations with the goal of finding a roundtrip to bring all objects home. Figure 11 shows the planning task and a plan for 4 objects.



Plan: *move(home,loc1)*
put-in(block)
move(loc1,loc2)
put-in(flower)
move(loc2,loc3)
put-in(letter)
move(loc3,loc4)
put-in(hat)
move(loc4,home)

Figure 11: Planning for 4 Objects in the Briefcase domain

A comparison to UCPOP turned out to be very difficult, because the system is very sensitive to subgoal permutations. For example, runtime varied from 5.6 s (with the "right" subgoal ordering) and 398 s under the *zlifo* strategy to generate a plan containing 12 actions. Since there is no way of finding the "right" subgoal ordering in advance, we tested all of them and decided to show the worst-case performance of the systems on each planning task. This made not a very big difference for Prodigy and almost no difference for GRAPHPLAN and IP^2.[15]

GRAPHPLAN shows a dramatic loss in performance because of the explosion in the number of ground instances of operators. For 4 objects and 5 locations, GRAPHPLAN has $2^4 = 16$ different *move* operators, which can take 5 different locations as start and end destination, i.e., GRAPHPLAN is faced with $16 \times 5 \times 5 = 400$ different possible ground instances of the *move*-operator plus 40 possible *put-in* and *take-out* actions. The average effort to expand the planning graph was about 4 s and most of the time was spent on search. Problems involving 5 objects were unsolvable, since the system was faced with 840 ground instances,

[15] Runtimes were measured on a SPARC 4, since UCPOP/LISP caused a segmentation fault on SUN Ultras.

which made the search of planning graphs beyond time step 8 impossible because of the huge number of possible goals.

Objects	Time Steps	Actions	GRAPHPLAN	UCPOP	Prodigy	IP^2
1	3	3	0.02	0.04	0.24	0.01
2	5	5	0.11	0.14	0.71	0.07
3	7	7	1.14	0.58	0.55	0.26
4	9	9	173.48	24.80	0.89	3.83
5	11	11	-	-	1.1	285.14

Figure 12: GRAPHPLAN, UCPOP, Prodigy, and IP^2 in the Briefcase domain

Prodigy is very fast on this test set, but it did not generate the roundtrip we had in mind and that was generated by all other planners. Instead, the system generates non-optimal plans with unnecessary *move* actions, see Figure 13. This behavior is probably explained with the lack of domain-specific control that we did not provide to any of the planners, but on which Prodigy's system philosophy puts a very strong emphasis.

(move home loc1) (put-in loc1 block) (move loc1 home)
(move home loc2) (put-in loc2 flower) (move loc2 home)
(move home loc3) (put-in loc3 letter) (move loc3 home)
(move home loc4) (put-in loc4 hat) (move loc4 home)

Figure 13: Prodigy's Plan for 4 Objects in the Briefcase domain

IP^2 has no problems in quickly solving small examples (and some larger ones if action parallelism can be exploited), but for 5 objects runtime explodes. The reason is not the large number of ground actions (which is 65 for 4 objects and 96 for 5)[16], but the huge increase in search. First, nothing can be done in parallel in this test set, which makes the graph's depth grow very quickly. Second, mutual exclusion relations do not help very much in solving the tasks. Two subgoals such as *at(letter,home)* and *at(letter,office)* are non-exclusive because they are added by conditional effects and propagating mutex information over conditional effects according to Lemma 1 is not sufficient to rule out inconsistent goal sets – it can only reduce them significantly. This forces IP^2 to exhaustively traverse the whole search space and prove that no plan exists that can make both subgoals true simultaneously, because this property of the domain follows from the interplay of the operator set with the initial state. As Blum and Furst pointed out in [3]

[16]The reader should also notice the fact proven by Blum and Furst that planning graphs are polynomially restricted in size of the planning problem, i.e., IP^2 is searching a finite search space of polynomial size.

already, if an object starts being in two different locations in the initial state it can continue to do so in all future world states. But although the basic physical law that no object can be in two different locations at the same time holds in all "normal" domains, a planning system has no explicit knowledge of it. Interestingly, as Figure 14 shows, subset memoization is able to eliminate most of the inconsistent goal sets, but not the explosion in the consistent goal sets. The figure shows IP² on a roundtrip and a permutation task in the briefcase world involving 5 objects and 6 different locations. The permutation task requires to transport the 5 objects from different locations in the initial state to different locations in the goal state. We show the number of consistent and inconsistent goal sets that are constructed at each time step.

	roundtrip		**permutation**	
time step	cons.	incons.	cons.	incons.
5	0	0	0	0
6	161	0	6	0
7	2,337	0	47	0
8	25,382	0	338	0
9	296,576	0	1,849	85
10	2,464,470	1,044,330	8,965	1,529
11	-	-	37,691	24,700
12			146,659	225,015
13			539,198	1,774,951
	roundtrip+subset		**permutation+subset**	
time step	cons.	incons.	cons.	incons.
5	0	0	0	0
6	161	0	6	0
7	2,337	0	47	0
8	25,345	0	123	0
9	187,218	0	449	0
10	866,291	0	1,529	0
11			6,124	0
12			19,459	240
13			68,424	1,980

Figure 14: Inconsistent and Consistent Goal Sets in the Briefcase World

With subset memoization, the number of actions tried for solving the permutation problem decreases from 14,211,487 (without subset memoization) to 502,047. The total number of inconsistent goals is reduced from 2,171.234 to 4,689, while the total number of consistent goals can only be reduced from 790,163 to 106,491. Nevertheless, runtime decreases from 11.1 minutes to 20.9 seconds.

We also constructed more difficult planning tasks in the Briefcase domain and tried to find the largest problem instances the systems could still handle. The problems 4i to 5c are contained in the code distribution of IP². In contrast to the above roundtrip tasks, these planning problems require to *put-in* and *take-*

out objects, which can often be done in parallel. For each problem, we give the number of different objects and locations in the initial state together with the number of different locations in the goal state, and the number of time steps and actions necessary for the shortest solution plan, see Figure 15. Runtimes were measured on a SUN SPARC 4. For UCPOP, we show the average runtime for each possible subgoal ordering. For Prodigy, the number of different locations in the goal state seemed to be especially critical, since it solved only one example in which all objects have to be transported to the same goal destination.

Problem	# Objects/ # Loc-I/Loc-G	Time Steps Actions	GRAPHPLAN	UCPOP	IP^2	Prodigy
4i	4/5/5	9/11	41.39	144	0.64	-
4j	4/5/4	9/12	51.47	308	0.67	-
4f	4/5/3	11/13	868.96	211	2.75	-
5a	5/6/1	11/11	-	190	337.25	0.32
5b	5/6/2	11/12	-	210	124.15	-
5c	5/6/3	11/13	-	312	35.9	-

Figure 15: GRAPHPLAN, UCPOP, IP^2 Prodigy in the Briefcase domain

GRAPHPLAN's explosion in runtime in example 4f is caused again by the huge number of ground instances of operators and the critical depth limit of 11, i.e., to solve the example, the system tried 158,761 different actions before a solution was found.

We also compared IP^2 to UCPOP and Prodigy in an extended version of the scheduling domain, originally developed for UCPOP. The domain contains 11 operators and many operators have quite complex conditional effects, see as an example the *roll* operator in Figure 16.

The planning tasks in this test suite require to process different objects by rolling, lathing, drilling, and painting in a shop. To paint an object, the right supply of paint of the right color has to be available, which sometimes means to go to a store and buy paint.

IP^2 and UCPOP generated the same plans to solve the planning tasks, but Prodigy's plans again contained superfluous *paint* and *move* actions. It had to repaint the objects several times, which also required to go to the store, get the right amount of paint, return to the shop, and prepare the paint for spraying. This makes its solution plans much longer than necessary. However, it has to be pointed out that we did not provide the system (as any other system) with additional domain-specific control knowledge. IP^2 was able to outperform UCPOP and Prodigy, because action parallelism and exclusivity constraints can be better exploited in this domain than in the Briefcase domain, see Figure 17. Surprisingly, subset memoization was useless since no goal set is constructed more than once.

```
roll
:v ?x object
:p is-at(sched)
:e ADD temperature(?x hot)
       shape(?x cylindrical)
       surface-condition(?x smooth);
   ALL ?old paint-cond n-e(?old nil)
            ⇒ DEL painted(?x ?old);
   ALL ?old shape-cond n-e(?old cylindrical)
            ⇒ DEL shape(?x ?old);
   ALL ?old temp-cond n-e(?old hot)
            ⇒ DEL temperature(?x ?old);
   ALL ?old surf-cond n-e(?old smooth)
            ⇒ DEL surface-condition(?x ?old);
   ALL ?old-width width ?old-orient orient
            ADD no-hole(?x ?old-width ?old-orient)
            DEL has-hole(?x ?old-width ?old-orient).
```

Figure 16: *Roll* operator from the Scheduling domain in IP^2 Input Syntax

Problem	time steps	actions	UCPOP	Prodigy	IP^2 Graph	IP^2 Plan	IP^2 Total
sched1	3	6	0.94	2.07	0.94	0.01	0.95
sched2	5	8	4.20	10.27	0.93	0.36	1.29
sched3	6	9	4.80	56.71	0.93	0.64	1.57
sched4	7	11	103.00	2.42	1.41	0.22	1.63
sched5	9	16	-	78.28	1.43	3.58	5.01
sched6	9	17	-	37.8	1.41	4.78	6.19

Figure 17: IP^2, Prodigy, and UCPOP in the Scheduling domain

8 Conclusion

We have presented an extension of planning graphs to deal with conditional and universally quantified effects in STRIPS operators. The resulting algorithm is sound and complete and terminates on unsolvable problems. It is fast when compared to other planners, although no sophisticated control strategies have been added yet and it relies on blind exhaustive search. The description in this report covers the functionality of version 2.0. In the meanwhile, version 3.0 has extended the system in four directions that were not described in this paper:

First, we have incorporated techniques that have been developed in our group to eliminate irrelevant facts and ground instances in simple STRIPS representations [10]. Based on a backchaining technique that works from the goal towards the initial state, different heuristics were developed that turned out to be very powerful. Extending this approach to deal with operators containing conditional and universally quantified effects has lead to the RIFO preprocessing heuristic that can be used with IP^2.

Second, we have added a more sophisticated instantiation of operators into ground actions that generates only those instances that satisfy static facts (inertia) from the initial state. For example, in Traveling Sales Person, only those *move* actions are generated for which the corresponding *connected* relation is given. This reduces significantly the size of planning graphs. We have also experimented with an approach to embed a background domain theory into the planner to add *integrity constraints* in order to detect inconsistent goal sets. However, first tests in the Briefcase domain showed that testing each goal set for inconsistency leads to a computational overhead that is not compensated by the gainings in search. However, it seems to be useful to use integrity constraints for a more extensive propagation of mutual exclusiveness relations over facts.

Third, we extended goal specifications to conditional and universally quantified goals such as “carry all blue objects that are in Room A to Room B”. Such planning tasks occur frequently in the robot domain that we are mainly focusing on and we want to provide a specification language that supports a natural formulation of robot instructions. This extension comes almost for free, because such goal specifications can simply transformed into sets of ground goals by preprocessing.

Finally, we have experimented with heuristics that estimate the branching factor in the search space and prefer most or least constrained actions. We obtained mixed results that will be described in a forthcoming paper.

Currently, we are working on a meta planning approach to split complex planning problems into reasonable subproblems and we are planning to investigate further language extensions.

9 Appendix: Subset Memoization

In an efficient implementation, we are facing the following problem: Trying to solve a goal and negative goal set $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$ at time n, how do we find out if one of the (exponentially many) subset pairs $\langle \mathcal{G}'_n, \mathcal{C}'_n \rangle : \mathcal{G}'_n \subseteq \mathcal{G}_n, \mathcal{C}'_n \subseteq \mathcal{C}_n$ has been memoized yet? While in GRAPHPLAN the memoizing algorithms only have to handle goal sets $\mathcal{G}_n$, IP^2's memoizing routines must deal with pairs of such sets, i.e., matters become more complicated.

Nevertheless, GRAPHPLAN's solution to the problem is ineffective. All goal sets that have been proven unsolvable at a certain time n are stored in a hash table memo_table as well as in a list unsolvable[n]. The routine to find out if a goal set $\mathcal{G}_n$ at time n has been memoized yet, is a simple hash table lookup, which is reasonably fast. For doing the subset test, however, the planner has to go through all entries at unsolvable[n] and check if the goal set is a subset of the current set. There can be an almost unlimited number (2^m, where m is the total number of facts at time n) of entries in such a list. Therefore, this approach will become practically intractable even for small sets of facts, or rather – because the time complexity of this algorithm does not directly depend on the size of the goal set – for any reasonably big planning task with usually thousands of memoized goal sets $\mathcal{G}_n$ at any time n. Being aware of this problem, Blum and Furst have developed the partial subset test, i.e., GRAPHPLAN tries to look up all subsets $G'_n \subseteq G_n$ with $|G'_n| = |G_n| - 1$ from memo_table. Obviously, this can be done in linear time, i.e., $O(|G_n|)$, but it is likely to result in more search effort.

To come up with an efficient solution to the complete subset test, we replaced both the hash table memo_table and the list unsolvable[n] by the UBTree (Unlimited Branching Tree) memoization procedure which uses the following complex tree data structure:

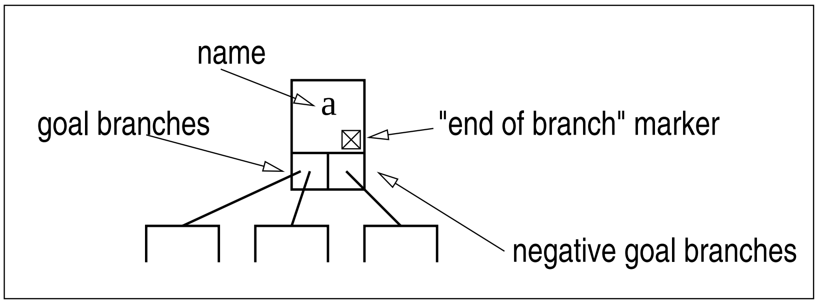


Figure 18: Representation of a single Node

A node N (see Figure 18) in the UBTree data structure consists of four components:

- N.name: the name, i.e., the fact it is representing;

- N.G: the goal branches, a set of other tree nodes;
- N.C: similarly the negative goal branches;
- N.eob : the end-of-branch marker.

At each level of the graph we have a set R[n] of such nodes. The following functions need to be defined for this tree:

- memoize$(n, \mathcal{G}_n, \mathcal{C}_n)$ inserts an unsolvable pair $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$ into UBTree. It is called each time after a pair has been proven to be unsolvable.
- Boolean memoized$(n, \mathcal{G}_n, \mathcal{C}_n)$ tries to look up $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$ from UBTree, i.e., it returns TRUE iff $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$ has already been memoized. It is called each time before the planner tries to solve a new pair of goal sets.
- Boolean subset_memoized$(n, \mathcal{G}_n, \mathcal{C}_n)$ does the complete subset check, i.e., it returns TRUE iff any subset pair $\langle \mathcal{G}'_n, \mathcal{C}'_n \rangle$ with $\mathcal{G}'_n \subseteq \mathcal{G}_n, \mathcal{C}'_n \subseteq \mathcal{C}_n$ has already been memoized. It is called when memoized has returned FALSE.

As their input, all functions take the time n as well as the goal set $\mathcal{G}_n$ and the negative goal set $\mathcal{C}_n$. Let $k = |\mathcal{G}_n|$ with $\mathcal{G}_n = \{g_0 \dots g_{k-1}\}$, and $m = |\mathcal{C}_n|$ with $\mathcal{C}_n = \{c_0 \dots c_{m-1}\}$. Let us assume that $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$ is unsolvable at time n. The memoize algorithm makes use of the function insert, which is defined in Figure 19 and takes a single goal g and a set of tree nodes S as arguments and returns a node N. Note that S is a VAR parameter, i.e., its value can be changed by the insert function. The memoize algorithm proceeds as defined in Figure 20.

```
do
    if ∃N ∈ S : N.name = g.name
        return N
    else
        create a new node N : N.name = g.name
        S := S ∪ N
        return N
end
```

Figure 19: insert(g, S)

It is our intention to have exactly one *branch* in the tree corresponding to each pair $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$ the planner memoizes. So let us first define what we mean by *branch*:

Definition 16 *Let* $\mathcal{G}_n = \{g_0 \dots g_{k-1}\}$ *and* $\mathcal{C}_n = \{c_0 \dots c_{m-1}\}$ *be ordered sets of goals. A branch* $\beta(n, \mathcal{G}_n, \mathcal{C}_n)$ *at time* n *corresponding to* $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$ *in* UBTree *is a tuple of tree nodes* $\mathsf{N}_0 \dots \mathsf{N}_{\mathsf{k+m-1}}$ *such that*

1. $\mathsf{N_0} \in \mathsf{R[n]}$ *and* $\mathsf{N_0.name} = g_0.name$

2. $\forall j \in 1 \ldots k-1 : \mathsf{N_j} \in \mathsf{N_{j-1}.G}$ *and* $\mathsf{N_j.name} = g_j.name$

3. $\forall j \in k \ldots m+k-1 : \mathsf{N_j} \in \mathsf{N_{j-1}.C}$ *and* $\mathsf{N_j.name} = c_{j-k}.name$

4. $\mathsf{N_{k+m-1}.eob} = \mathsf{TRUE}$

```
do
    sort(𝒢_n)
    sort(𝒞_n)/* in arbitrary, i.e., alphabetical order */
    N := insert(g_0, R[n])
    for j := 1 to N do
        N := insert(g_j, N.G)
    for j := 0 to m do
        N := insert(c_j, N.C)
    N.eob := TRUE
end
```

Figure 20: $\mathsf{memoize}(n, \mathcal{G}_n, \mathcal{C}_n)$

We need a way of ordering the goal sets to make sure that there is exactly one branch for each pair $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$. The easiest way to achieve a total ordering, of course, is to arrange the goal names in alphabetical order. To insert a goal pair, memoize starts at the root of the tree. Note that we assume $\mathcal{G}_n \neq \emptyset$, which seems to be a quite natural assumption. Then it follows the already existing branches in the tree, creating a new branch only when the current goal/negative goal is not in N.G/N.C. The end of each branch is marked with $\mathsf{TRUE} = \boxtimes$. This way, the created branch is exactly $\beta(n, \mathcal{G}_n, C_n)$ as defined above.

Figure 21 shows an example with the four pairs $\langle \{b, a\}, \{d, c\} \rangle$, $\langle \{b\}, \{\} \rangle$, $\langle \{a, b\}, \{c\} \rangle$ and $\langle \{c, a\}, \{d\} \rangle$ that are one by one inserted into $R[n]$ at some time n. Remember that the goal sets are ordered alphabetically before memoization.

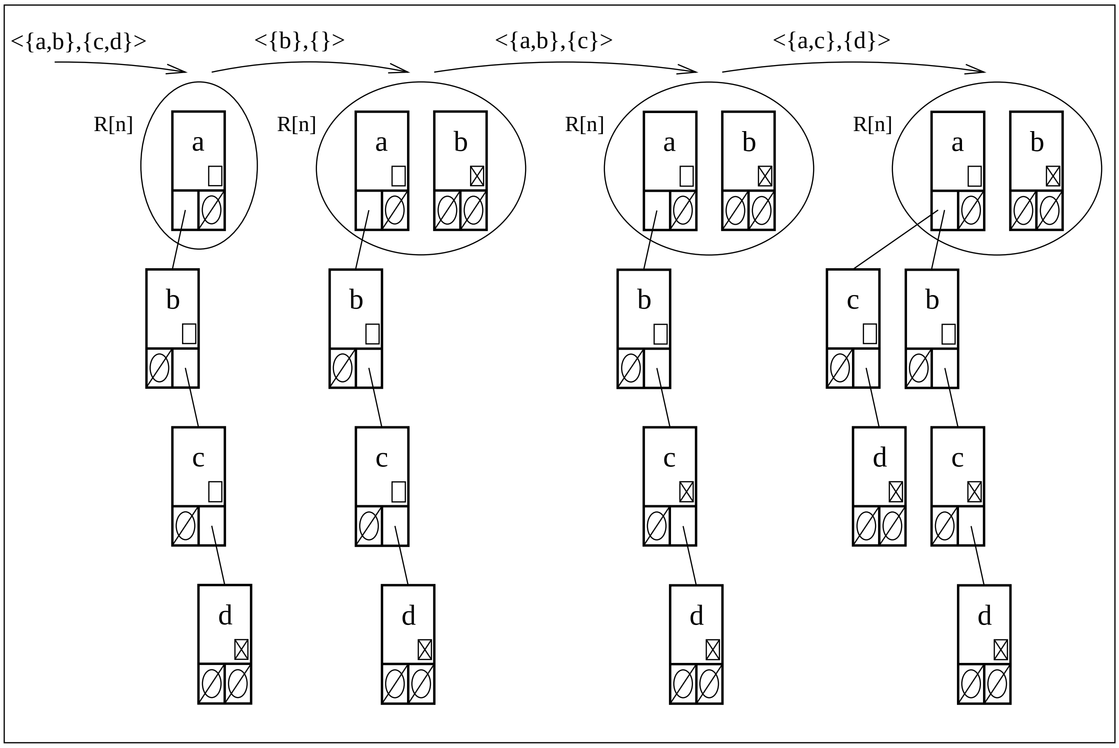


Figure 21: Iterative Memoization of 4 Goal Pairs

To test if a goal pair (or a corresponding pair of subgoals) has already been inserted into the UBTree data structure, the memoized and subset_memoized procedures are used by IP^2.

The memoized functions make use of a simple function look_up, which takes a goal g, a set of tree nodes S as well as a VAR Parameter node N. It returns TRUE iff there is a node N′ in S that corresponds to g. In this case, N′ is assigned to N, cf. the definition of look_up in Figure 22.

do
    if $\exists \mathsf{N}' \in \mathsf{S} : \mathsf{N}'.\mathsf{name} = g.name$
      N := N′
      return TRUE
    else
      return FALSE
end

Figure 22: $\mathsf{look\_up}(g, \mathsf{S}, \mathsf{N})$

To test if a pair $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$ at time n has already been memoized, the function memoized$(n, \mathcal{G}_n, \mathcal{C}_n)$ simply follows the branch $\beta(n, \mathcal{G}_n, \mathcal{C}_n)$ in the tree which represents $\langle \mathcal{G}_n, \mathcal{C}_n \rangle$ until it either finds that β does not exist or all goals $g_0 \dots g_{k-1}$ and $c_0 \dots c_{m-1}$ have been processed. It returns TRUE iff it comes across a node N with N.eob = TRUE, see Figure 23.

```
do
    sort(G_n)
    sort(C_n)
    if not(look_up(n, R[n], N)) return FALSE
    if N.eob return TRUE
    for j := 1 to k do
        if not(look_up(n, N.G, N)) return FALSE
        if N.eob return TRUE
    for j := 0 to m do
        if not(look_up(n, N.C, N)) return FALSE
        if N.eob return TRUE
    return FALSE
end
```

Figure 23: memoized$(n, \mathcal{G}_n, C_n)$

Note that this routine already does some just-by-chance subset checking: if it comes across a node N with N.eob = TRUE without having checked all the goals and negative goals, it found a branch β that lies completely within the current branch and therefore represents an unsolvable subset pair $\langle \mathcal{G}'_n, \mathcal{C}'_n \rangle$ with $\mathcal{G}'_n \subseteq \mathcal{G}_n, \mathcal{C}'_n \subseteq \mathcal{C}_n$ that has already been memoized. To make this more clear, let us define what we mean by a *sub_branch* β':

Definition 17 *Let* $\mathcal{G} = \{g_1 \dots g_{k-1}\}$ *and* $\mathcal{C} = \{c_0 \dots c_{m-1}\}$ *be ordered sets of goals and* N_0 *a node in* UBTree. *A sub_branch* $\beta'(\mathsf{N}_0, \mathcal{G}, \mathcal{C})$ *corresponding to* N_0*,* $\mathcal{G}$*, and* $\mathcal{C}$ *is a tuple of tree nodes* $\mathsf{N}_1 \dots \mathsf{N}_k$ *such that:*
$\exists\, h \in 0 \dots k$ *and ordered subsets* $\mathcal{G} \supseteq \mathcal{G}' = \{g_{l_1} \dots g_{l_h}\}, \mathcal{C} \supseteq \mathcal{C}' = \{c_{l_{h+1}} \dots c_{l_k}\}$*:*

1. $\forall j \in 1 \dots h : \mathsf{N}_\mathsf{j} \in \mathsf{N}_{\mathsf{j}-1}.\mathsf{G}$ *and* $\mathsf{N}_\mathsf{j}.\mathsf{name} = g_{l_j}.name)$

2. $\forall j \in h+1 \dots k : (\mathsf{N}_\mathsf{j} \in \mathsf{N}_{\mathsf{j}-1}.\mathsf{C}$ *and* $\mathsf{N}_\mathsf{j}.\mathsf{name} = c_{l_j}.name)$

3. $\mathsf{N}_\mathsf{k}.\mathsf{eob} = \mathsf{TRUE}$

To do the complete subset check, the planner has to lookup all branches $\beta(n, \mathcal{G}'_n, \mathcal{C}'_n)$ in the tree that represent a pair $\langle \mathcal{G}'_n, \mathcal{C}'_n \rangle$ with $\mathcal{G}'_n \subseteq \mathcal{G}_n$ and $\mathcal{C}'_n \subseteq \mathcal{C}_n$ and return TRUE iff one of those has been memoized already. This is worst case exponential in the size of the given goal sets. To be more precise, there are exactly $(2^{|\mathcal{G}_n|} - 1) * 2^{|\mathcal{C}_n|}$ such branches: the number of possible subsets except the empty set $|\{\mathcal{G}'_n|\mathcal{G}'_n \subseteq \mathcal{G}_n\} \setminus \emptyset|$ multiplied with the number of possible negative goal subsets $|\{\mathcal{C}'_n|\mathcal{C}'_n \subseteq \mathcal{C}_n\}|$. Since many branches can be excluded very early during search due to the total ordering of the goals, the algorithm turns out to be much faster in the average case, though.

We now state a lemma that is the key to the $\mathsf{subset\_memoized}$ algorithm.

Lemma 5 *Let $\mathcal{G}_n = \{g_0 \dots g_{k-1}\}$ and $\mathcal{C}_n = \{c_0 \dots c_{m-1}\}$ be ordered sets of goals at time n and let $\mathcal{G}'_n \subseteq \mathcal{G}_n, \mathcal{C}'_n \subseteq \mathcal{C}_n$ be ordered subsets of them. The goal pair $\langle \mathcal{G}'_n, \mathcal{C}'_n \rangle$ has been memoized already if and only if there exists a goal $g_j \in \mathcal{G}_n$ and a node $\mathsf{N_0} \in \mathsf{R[n]}, \mathsf{N_0.name} = g_j.name$ such that there is a sub-branch $\beta'(\mathsf{N_0}, \{g_{j+1} \dots g_{k-1}\}, \mathcal{C}_n)$ corresponding to $\mathsf{N_0}, \{g_{j+1} \dots g_{k-1}\}$ and $\mathcal{C}_n$ in UBTree.*

Proof: $\Rightarrow$: If $\langle \mathcal{G}'_n, \mathcal{C}'_n \rangle$ is memoized, there is a branch $\beta(n, \mathcal{G}'_n, \mathcal{C}'_n) = \mathsf{N_0} \dots \mathsf{N_{k'+m'-1}}$ in the tree, where $\mathsf{N_0}$ corresponds to one of the goals $g_j \in \mathcal{G}_n$. Obviously $\mathsf{N_1} \dots \mathsf{N_{k'+m'-1}}$ satisfies the conditions for a sub-branch corresponding to $\mathsf{N_0}$, $\mathcal{G}_n$, and $\mathcal{C}_n$. Furthermore, $\mathcal{G}'_k \cap \{g_0 \dots g_{j-1}\} = \emptyset$. Otherwise, $\beta(n, \mathcal{G}'_n, \mathcal{C}'_n)$ would start with the goal $g \in \mathcal{G}'_n \cap \{g_0 \dots g_{j-1}\}$ due to the total ordering of the goals.

$\Leftarrow$: Given a sub-branch $\beta' = \mathsf{N_1} \dots \mathsf{N_{k'+m'-1}}$ corresponding to a node $\mathsf{N_0} \in \mathsf{R[n]}$ with $\mathsf{N_0.name} = g_j.name$ and goal sets as defined above, we know from the definition of *sub_branch* that $\mathsf{N_0} \dots \mathsf{N_{k'+m'-1}}$ must be the branch $\beta(n, \mathcal{G}'_n, C'_n)$ for some goal sets $\mathcal{G}'_n$ and C'_n. It is easy to see that these goal sets must be subsets of $\mathcal{G}_n$ and C_n, i.e., $\mathcal{G}'_n \subseteq \mathcal{G}_n$ and $\mathcal{C}'_n \subseteq \mathcal{C}_n$. ■

Considering this result, the complete subset test consists of two functions:

- the main function, $\mathsf{subset\_memoized}(n, \mathcal{G}_n, \mathcal{C}_n)$, which just provides some sort of setup for the other function,
- $\mathsf{exists\_sub\_branch}(\mathsf{N_0}, \mathcal{G}, \mathcal{C})$, which is recursive and takes a tree node $\mathsf{N_0}$ as well as two goal sets $\mathcal{G}$ and $\mathcal{C}$ as input; it returns TRUE iff there is a sub_branch $\beta'(\mathsf{N_0}, \mathcal{G}, \mathcal{C})$ corresponding to the input.

```
do
    sort(G_n)
    sort(C_n)
    for j := 0 to n do
        if look_up(g_j, R[n], N)
            if exists_sub_branch(N, {g_{j+1} ... g_{k-1}}, C_n) return TRUE
    return FALSE
end
```

Figure 24: $\mathsf{subset\_memoized}(n, \mathcal{G}_n, \mathcal{C}_n)$

Let us first have a look at subset_memoized, as shown in Figure 24. This procedure initializes the exists_sub_branch routine with all possible starting points for a subset branch, i.e., with the trees that have one of the goals $g_j \in \mathcal{G}_n$ as roots. Note that if, for example, the first goal g_0 happens to be not in R[n] (i.e., there is no corresponding $\mathsf{N} \in$ R[n]), the number of branches to be examined is immediately reduced to $(2^{|\mathcal{G}_n|-1} - 1) * 2^{|\mathcal{C}_i|}$, which is only about half of all possible subset branches in the tree. To store the current starting node in the recursion pattern,exists_sub_branch needs a local variable N, cf. Figure 25.

```
do
    if G ≠ ∅
      /* see if we can find a negative goal subset here */
      if exists_sub_branch(N0, ∅, C) return TRUE
      if N0.G = ∅ return FALSE/* no possible ways to go */
      for j:=0 to k do
          if look_up(gj, N0.G, N)
            /* found an existing branch; see if we have success here */
            if exists_sub_branch(N, {gj+1 ... gn}, C) return TRUE;
      /* all the goals have failed */
      return FALSE
    else
      if N.eob return TRUE/* we can stop here */
      if N0.C = ∅ return FALSE
      for j:=0 to m do
          if look_up(gj, N0.C, N)
            if exists_sub_branch(N, ∅, {cj+1 ... cm}) return TRUE
      return FALSE
end
```

Figure 25: exists_sub_branch$(\mathsf{N}_0, \mathcal{G}, \mathcal{C})$

From a formal point of view, exists_sub_branch makes use of the following lemma:

Lemma 6 *Given ordered sets of goals $\mathcal{G} = \{g_0 \dots g_{k-1}\}$ and $\mathcal{C} = \{c_0 \dots c_{m-1}\}$ as well as a starting point N_0 in the tree, there exists a sub_branch $\beta'(\mathsf{N}_0, \mathcal{G}, \mathcal{C})$ iff at least one of the following conditions holds:*

1. N_0.eob = TRUE
2. *there is a goal $g_j \in \mathcal{G}$ and a node $\mathsf{N} \in \mathsf{N}_0$.G corresponding to this goal such that there exists a sub-branch $\beta'(\mathsf{N}, \{g_{j+1} \dots g_{k-1}\}, \mathcal{C})$ in* UBTree.
3. *there is a goal $c_j \in \mathcal{C}$ and a node $\mathsf{N} \in \mathsf{N}_0$.C corresponding to this goal such that there exists a sub-branch $\beta'(\mathsf{N}, \emptyset, \{c_{j+1} \dots c_{m-1}\})$ in* UBTree.

Proof: $\Rightarrow$: Let $\mathsf{N}_1 \ldots \mathsf{N}_\mathsf{k}$ be a sub-branch corresponding to N_0, $\mathcal{G}$, and $\mathcal{C}$. One of the following three cases must hold:

1. $k = 0$: N_0.eob $=$ TRUE follows directly from the definition of *sub_branch*.

2. $k > 0$ and N_1 corresponds to a goal $g_j \in \mathcal{G}$: Let $\beta' := \mathsf{N}_2 \ldots \mathsf{N}_\mathsf{k}$. Obviously, β' is a sub-branch corresponding to N, $\mathcal{G}$, and $\mathcal{C}$. Furthermore, no node in β' can correspond to a goal $g \in \{g_0 \ldots g_j\}$ due to the total ordering of the goals.

3. $k > 0$ and N_1 corresponds to a goal $c_j \in \mathcal{C}$: similar to 2.

$\Leftarrow$:

1. If N_0.eob $=$ TRUE, the empty tuple of nodes is a legal sub_branch: let $h = 0$ in Definition 17.

2. If $\mathsf{N}_1 \ldots \mathsf{N}_\mathsf{k}$ is a sub-branch corresponding to N, $\{g_{j+1} \ldots g_{n-1}\}$, and $\mathcal{C}$ with N.name $= g_j.name$, then $\mathsf{N}, \mathsf{N}_1 \ldots \mathsf{N}_\mathsf{k}$ obviously is a sub_branch corresponding to N_0, $\mathcal{G}$, and $\mathcal{C}$.

3. similar to 2.

■

Thus exists_sub_branch does the following: First, it checks if a negative goal subset starts at node N_0. If this fails, it tries to look_up all the goals $g_j \in G$ from N_0.G. Once exists_sub_branch found such a goal g_j, it calls itself on the new node N and $\{g_{j+1} \ldots g_{k-1}\}$ with $\mathcal{C}$ remaining the original set. Trying to find a negative goal subset proceeds in the same way, except that exists_sub_branch can stop and return TRUE (which is propagated back through the recursion pattern) if N_0.eob $=$ TRUE, see Lemma 6.

To make this more explicit, consider the example in Figure 26, where subset_memoized tries to look up any subset of the pair $\langle\{a, b\}, \{d\}\rangle$ in the tree resulting from the memoization process illustrated in Figure 21.

First, there is the initial call to subset_memoized($n, \{a, b\}, \{d\}$), which looks up node1 in R[n] and calls exists_sub_branch(node1, $\{b\}, \{d\}$). exists_sub_branch first calls itself on (node1, $\emptyset$, $\{d\}$) (i.e., it tries to find a negative goal subset here) and finds that node1.eob $=$ FALSE and node1.C $= \emptyset$. It returns FALSE and looks up a node corresponding to b in node1.G, which is node4. Thus the next call is exists_sub_branch(node4, $\emptyset$, $\{d\}$). There are no more goals to examine, so this instance of exists_sub_branch only does the negative goal subset test. Since node4.eob $=$ FALSE, it tries to lookup a node corresponding to the negative goal d in node4.C, which fails. This causes control to return to the instance of exists_sub_branch that was called on (node1, $\{b\}$, $\{d\}$). This instance tries to lookup the next goal in node1.G. As there are no more goals (b was the only one and it has already been looked up), this fails too, and FALSE is returned which terminates the recursion. Now subset_memoized tries to get the next goal, which is b, in the root set R[n]. It finds a matching node, namely node2 and calls exists_sub_branch on node2, $\mathcal{G} = \emptyset$ and $\mathcal{C} = \{c\}$. exists_sub_branch finds that node2.eob = TRUE, which leads to a success: TRUE is returned to subset_memoized, and, in turn, subset_memoized returns TRUE to the planner.

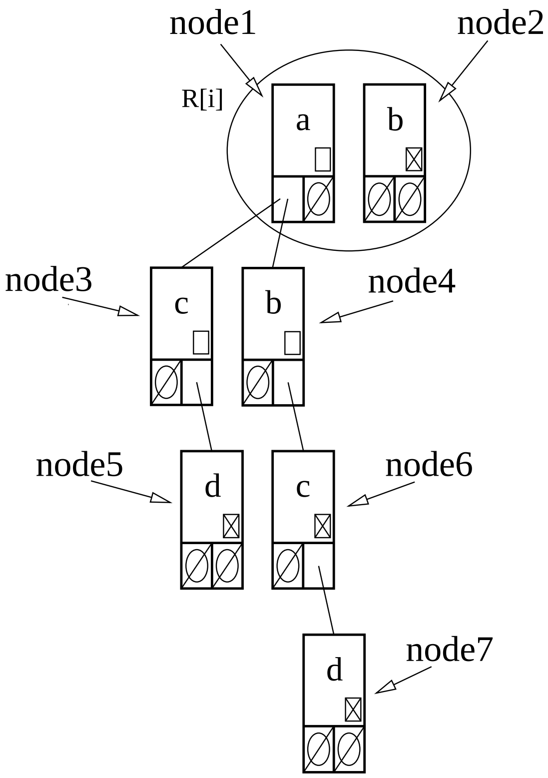


Figure 26: Looking up $\langle\{a, b\}, \{d\}\rangle$ in this Tree leads to a Success

References

[1] K. Konolige A. Saffioti and E. Ruspini. A multivalued logic approach to integrated planning and control. *Artificial Intelligence*, 76(1–2):481–526, 1995.

[2] 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.

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

[4] B. Drabble, editor. *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems*. AAAI Press, Menlo Park, 1996.

[5] E. Fink and M. Veloso. Prodigy planning algorithm. Technical Report CMU-94-123, Carnegie Mellon University, 1994.

[6] B. Gazen and C. Knoblock. Combining the expressivity of UCPOP with the efficiency of Graphplan. In Steel [15].

[7] S. Kambhampati, E. Lambrecht, and E. Parker. Understanding and extending Graphplan. In Steel [15].

[8] 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, 1994.

[9] D. McDermott. A heuristic estimator for means-ends analysis in planning. In Drabble [4], pages 142–149.

[10] B. Nebel, Y. Dimopoulos, and J. Koehler. Ignoring irrelevant facts and operators in plan generation. In Steel [15].

[11] E. Pednault. ADL: Exploring the middle ground between STRIPS and the Situation Calculus. In R. Brachman, H.J. Levesque, and R. Reiter, editors, *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, Toronto, Canada, 1989. Morgan Kaufmann.

[12] 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.

[13] 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.

[14] Frank Rittinger. XGV - a graphical interface to analyse planning graphs. System Documentation, available over http://www.informatik.uni-freiburg.de/~ koehler/ipp.html, 1997.

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