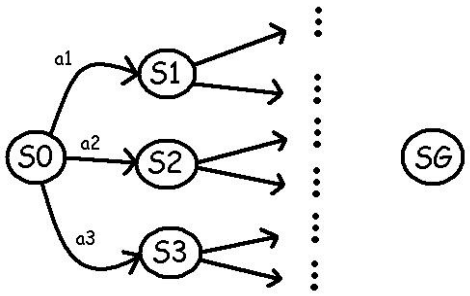
Translation of Planning Problems to Search Problems (review)

The state *S* is a conjunction of fluent literals, where
$S \models \zeta$

$\zeta^+ \subseteq S, \zeta^- \cap S = \phi$
If *a* is an action applicable in *S*, then
$S \models pre_a$
where *Pre* is the precondition of a.

The following is an example of a *progressive* search (s_i's are states, a_i's are actions). [Added: s1, s2, s3 are the actions executable in s0]



Heuristics

A *heuristic function h* is a function that maps from a state in the search problem to a positive [non-negative] real number.

$h : State \rightarrow R^+$

$h(s) \in R^+$
We want a heuristic function to be *admissible*, meaning that is does not overestimate the cost. This guarantees that we will find a solution to our search problem using the heuristic.

Another important feature is that the heuristic function must be easily computable. An example is the heuristic function *h(s) =length of the shortest plan achieving G*. This function is admissible, but is harder to compute than the original search problem [might be as hard as solving the problem].

To create a heuristic for a problem, we usually define a relaxed problem, and then solve the relaxed problem. There are two [common] ways to relax a [planning] problem. We can either remove preconditions of the actions, or we can assume that the subgoals are independent [or both].

Example	Airport:SFO, JFK Plane: A1 Cargo: C1,C2		
	Load(c,p,a)	*Unload*(c,p,a)	*Fly*(p,f,l)
	Pre: At(c,a), At(p,a)	Pre: At(p,a), In(c,p)	Pre: At(p,f)
	Add: In(c,p)	Add: At(c,a)	Add: At(p,t)
	Del: At(c,a)	Del: In(c,p)	Del: At(p,f)

We can implement relaxation in several different ways. The first is to remove the preconditions associated with the actions. The heuristic function then becomes *h(s)*=number of unsatisfied subgoals (Figure 1). [This heuristic is not always accurate]

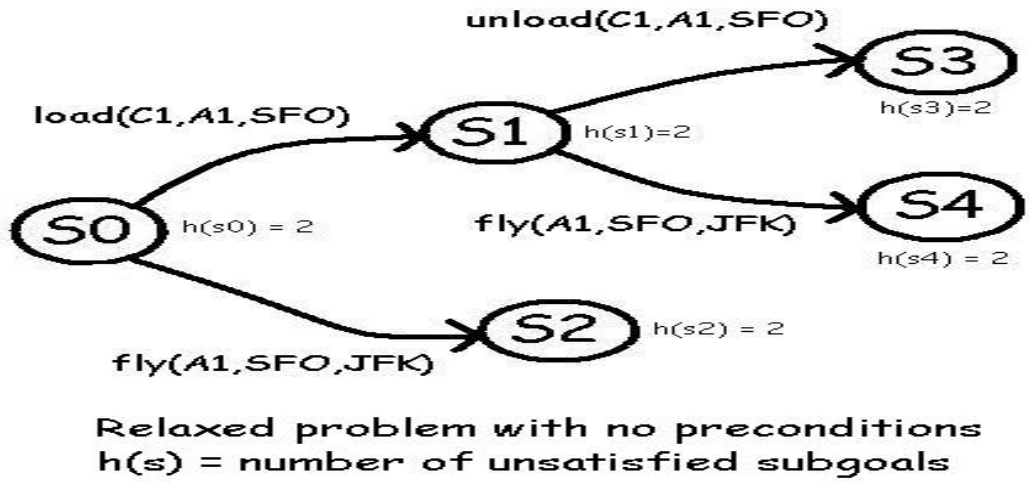


Figure 1

A second method to relax the problem is to remove the preconditions and negative effects. The heuristic function becomes the length of the shortest plan to *G* in the new state graph (Figure 2).

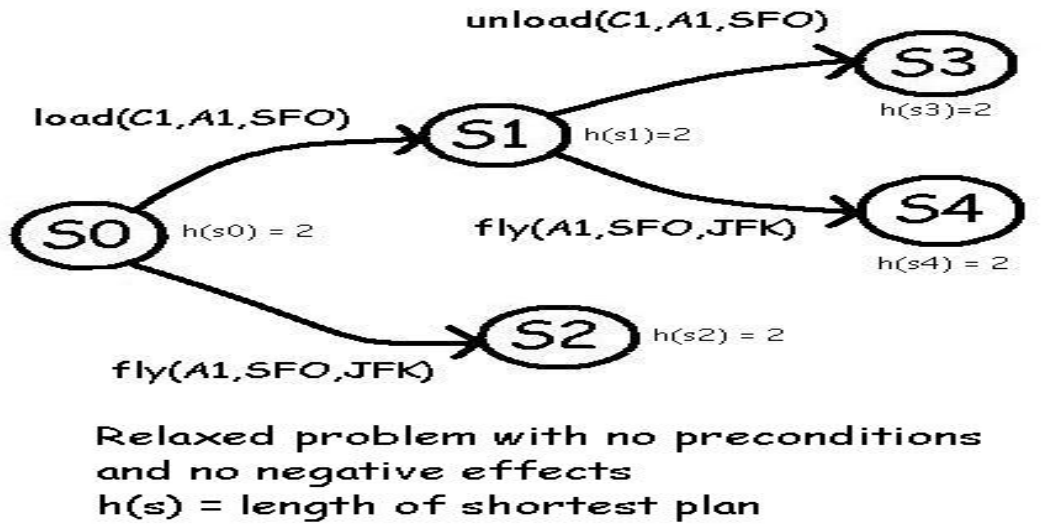


Figure 2

The last method is to only remove the delete effects. Then the function is *h(s)=length of the shortest plan to G* (Figure 3). This function is polynomial time to compute since the maximum value of *h(s*) is the number of fluents in the planning problem.

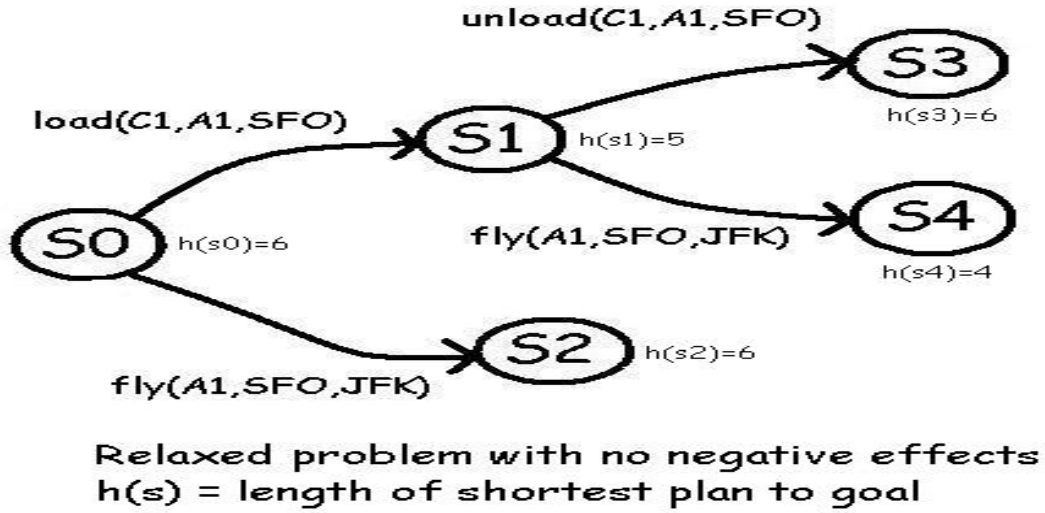


Figure 3

HSP

HSP uses two types of heuristics:

1. Ignore negative effects
2. $g_S =$

$$\begin{cases} 0, if\ \ p \in s \\ \min\limits_{\substack{a \in act \\ p \in Add}} [1 + g_s(pre_a)], otherwise \end{cases}$$

[I don't know how to change the figure – the second condition should be p ∈ Add_a]

There are two different methods to compute g_s [I would say that there are two different ways to define]

1. (additive)

$$g_s(c) = \sum_{r \in c} g_{s(r)}$$

2. (max)

$$g_s = \max_{r \in c} g_{s(r)}$$

Then to compute g_s:

If $p \notin s, g_s(p) = \infty$

Then the action *a* is applicable in *s*.

$p \in Add_a$

$g_s(p) = \min[g_s(p), 1 + g_s(pre_a)]$

Example

$p = At(C1, JFK)$

$g_{s0}(p) = 1 + g_{s0}(pre_a) = 3$

$a = Unload(C1, A1, JFK)$

$g_{s0}(In(C1, A1) = \min[1 + g_{s0}(pre_{load(C1, A1, SFO)},$

$1 + g_{s0}(pre_{load(C1, A1, JFK)}] = 1$

$g_{s0}(At(C2, SFO)) = 4$

The heuristics used in HSP are not admissible, but using them can lead to solving more problems, and sometimes getting better plans.

Graphplan

Graphplan works by preprocessing a planning problem to form a structure that contains information about the problem, a *planning graph*. A planning graph is different from a state graph in that it does not represent the possible states of the problem in each node. Instead, it ties the execution of an action to time. Each node is composed of a proposition of literals that can be true up to that point, and actions, or literals that be executed (Figure 4).

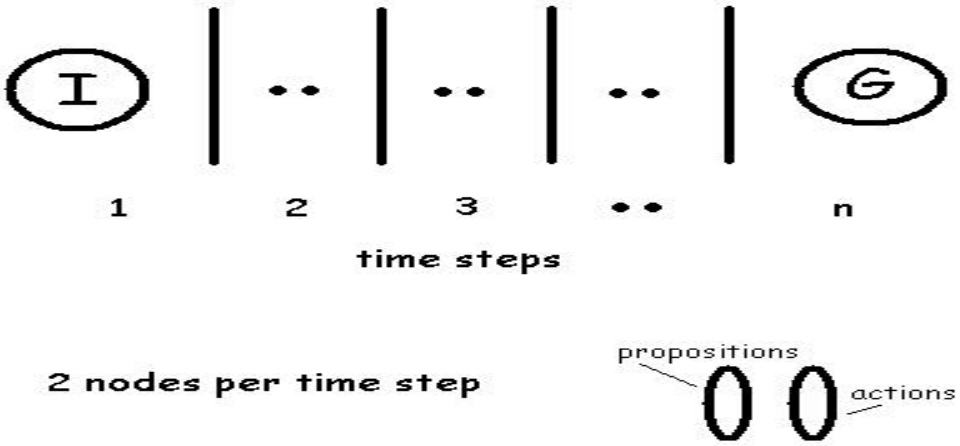


Figure 4

The goal of using a planning graph is to be able to find a solution faster by not having to search through the state-space.

Example – cake domain

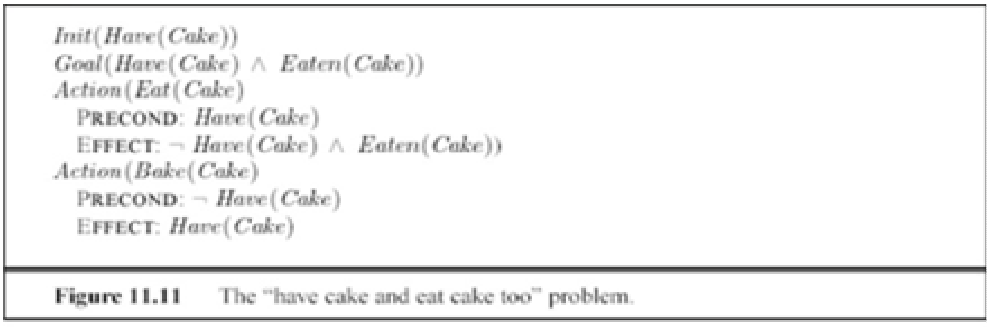
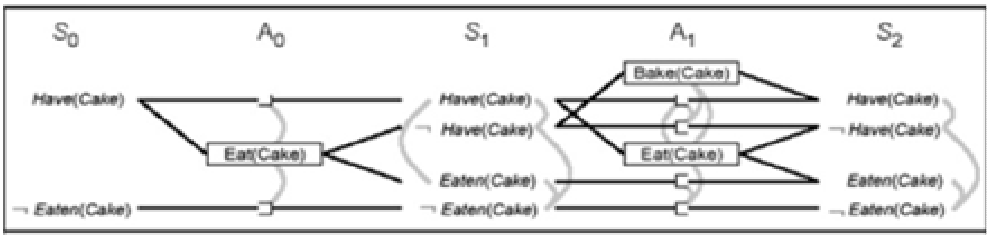

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake)$

Figure 11.11 The "have cake and eat cake too" problem.



The gray lines in the planning graph are mutexes (mutually exclusive). Mutexes between action have several different causes:
1. Inconsistent effects: Two effects are said to be inconsistent if the effects of one negate the effect of the other.
2. Interference: This occurs when one of the effects of one action is the negation of a precondition of the other.
3. Competing needs: When one of the preconditions of one action is mutually execlusive with a precondition of the other.

Mutexes also hold between literals at the same level. This can occur if the two literals are contradictory, or if for all pair of actions achieving the two are mutually exclusive.

We do not mark all possible mutexes in the planning graph, since to find all possible mutexes is as difficult as the original search problem (NP-hard).

Heuristic functions from planning graphs

1. If a literla *l* does not appear in the planning graph, then *l* is not achievable.
Then, h(s) = infinity if *s* |= *l*.
2. Level cost: the minimal cost [level] where *l* appears. Level cost is admissible for a single goal, but not for conjunctions.
3. For a conjunction of literals, different possible functions include:
a. Max cost of all the level costs (admissible, but tends to severely underestimate the cost and therefore is slow).
b. Sum cost of all the level costs (not admissible, but is fast).
c. Set level, the level where all the fluents appear with no mutexes.