Planning Algorithms

Murray Shanahan

Overview

- Computing the effects of actions
  - STRIPS
- Computing plans
  - Partial order planning (POP)
  - Forward planning with heuristics
  - Satisfiability planning (SAT)
  - GRAPHPLAN
- Residual topics

Logic Programs versus Explicit Algorithms

- The logic programming approach to planning tries to follow Bob Kowalski's dictum:

  Algorithm = Logic + Control

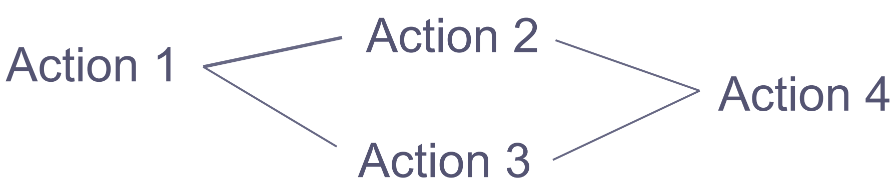
- This is a beautiful idea. But to build a practical planner it's usually easier to design a planning algorithm explicitly
- Moreover, the frame problem doesn't then arise at the implementation stage (as it does at the specification stage)
- Because in order to *compute* the effects of an action (as opposed to reasoning about them in logic), we can directly modify a data structure representing the situation (such as a list of fluents)

The STRIPS Approach

- A classic way to directly compute the effects of actions is the STRIPS approach
- According to the STRIPS approach, all we need to do to compute the effects of an action on a situation is:
  - Represent the situation as a list *L* of fluents
  - Represent the effects of an action with three lists: a list of fluents to be added (`Addlist`), a list of those to be deleted (`Deletelist`) and a list of `Preconditions`
  - Add fluents to and delete fluents from *L* according to the `Addlist` and `Deletelist`, assuming all the fluents in `Preconditions` occur in *L*
  - Obviously fluents that are in neither `Addlist` nor `Deletelist` will be unaffected by this

Partial Order Planning

- Partially ordered plans
  - A plan is *totally ordered* if it specifies the relative ordering of every action it contains
  - A plan that leaves the relative ordering of some of its actions unspecified is *partially ordered*



Executing actions in order 1, 2, 3, 4 or in order 1, 3, 2, 4 are both legitimate

- Partial order planning algorithms generate partially ordered plans, using the concepts:
  - Protected links
  - Threats
  - Promotion and demotion

Partial Order Planning Concepts

- Protected links
  - A protected link constrains a fluent to hold between two given time points
  - Protected links are established when an action is added to the plan to achieve a certain goal
- Threats
  - A protected link from T1 to T2 is threatened if an action is added to the plan that *could* occur between T1 and T2 which terminates the fluent being protected
- Promotion and demotion
  - Extra ordering constraints can be added to the plan to ensure either that such an action occurs before T1 or after T2

Partial Order Planning (POP) Algorithm

```
while goal list non-empty
   choose a goal <F1,T1> from goal list
   choose an action <A,T2> whose effects include F1
   for each precondition F2 of A add <F2,T2> to goal list
   add <A,T2> to plan (if not already a member)
   add T2 < T1 to plan
   add <T2,F1,T1> to protected links
   for each <A',T3> in plan that threatens some <T4,F3,T5>
      in protected links
      choose either
         promotion: add T3 < T4 to plan
         demotion: add T5 < T3 to plan
   end for
end while
```

Search in the POP Algorithm

- Note that lines 2, 3, and 9 of the algorithm are non-deterministic choices
- This defines a search tree
- Some search strategy has to be used to explore this search tree – depth-first or breadth-first, for example
- Note also that the pseudo-code here doesn’t mention the initial situation. But the necessary modifications are very simple

POP Example

- Consider the following event calculus effect axioms

$$Initiates(Go(x),At(x),t)$$
$$Terminates(Go(x),At(y),t) \leftarrow HoldsAt(At(y),t) \wedge x \neq y$$
$$Initiates(Buy(x),Have(x),t) \leftarrow HoldsAt(At(y),t) \wedge Sells(y,x)$$
$$Sells(HWS,Drill)$$
$$Sells(SM,Milk)$$

- and the following goal

$$HoldsAt(Have(Drill),T1)$$
$$HoldsAt(Have(Milk),T1)$$

- POP carries out the following computation

POP Computation (1)

Goal list	Plan	Protected links
<Have(Drill),T1> <Have(Milk),T1>		
<Have(Milk),T1> <At(HWS),T2>	<Buy(Drill),T2>, T2<T1	<T2,Have(Drill),T1>
<At(HWS),T2> <At(SM),T3>	<Buy(Drill),T2>, T2<T1 <Buy(Milk),T3>, T3<T1	<T2,Have(Drill),T1> <T3,Have(Milk),T1>
<At(SM),T3>	<Buy(Drill),T2>, T2<T1 <Buy(Milk),T3>, T3<T1 <Go(HWS),T4>, T4<T2	<T2,Have(Drill),T1> <T3,Have(Milk),T1> <T4,At(HWS),T2>

POP Computation (2)

Goal list	Plan	Protected links
	<Buy(Drill),T2>, T2<T1 <Buy(Milk),T3>, T3<T1 <Go(HWS),T4>, T4<T2 <Go(SM),T5>, T5<T3	<T2,Have(Drill),T1> <T3,Have(Milk),T1> <T4,At(HWS),T2> <T5,At(SM),T3>
	<Buy(Drill),T2>, T2<T1 <Buy(Milk),T3>, T3<T1 <Go(HWS),T4>, T4<T2 <Go(SM),T5>, T5<T3 T5<T4, T3<T4	<T2,Have(Drill),T1> <T3,Have(Milk),T1> <T4,At(HWS),T2> <T5,At(SM),T3>

Under threat

Promotion of Go(SM)

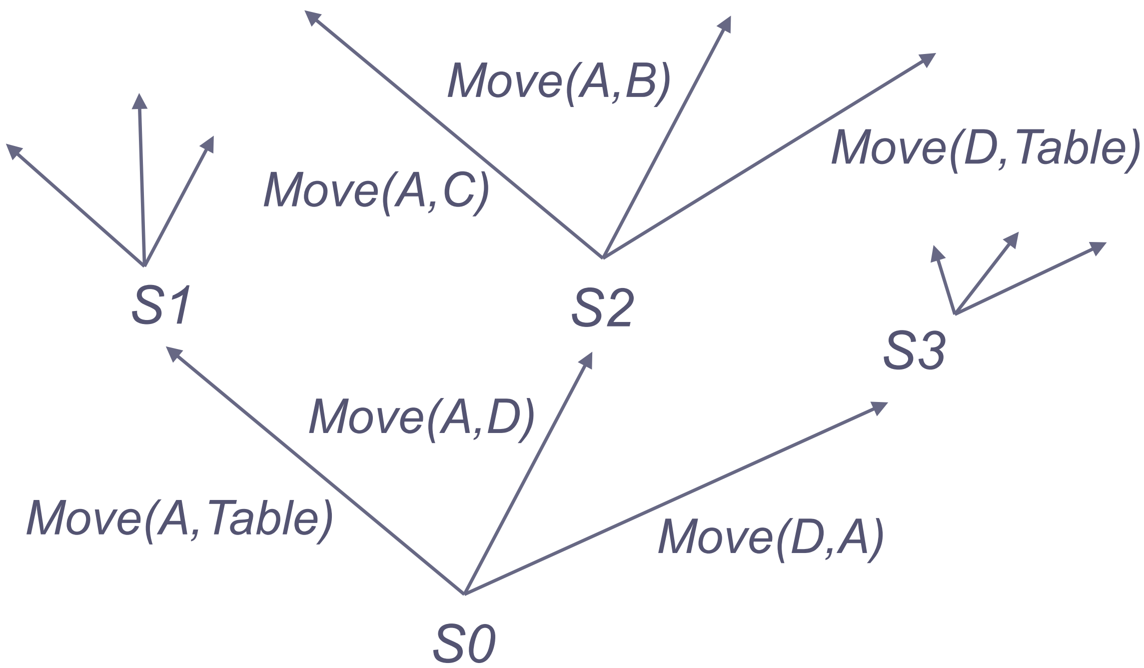
Demotion of Go(HWS)

Using Heuristics

- Most of the planners we are considering are general-purpose planners. That is to say, they assume no prior knowledge of the domain
- But with the use of a few domain-specific heuristics, we can build much faster planners
- One approach that uses heuristics is *forward planning*

Forward Planning (1)

- A forward planning algorithm starts in the initial situation, then considers all possible first actions that can be performed, then all possible second actions, and so on, until it finds a plan that leads to the goal

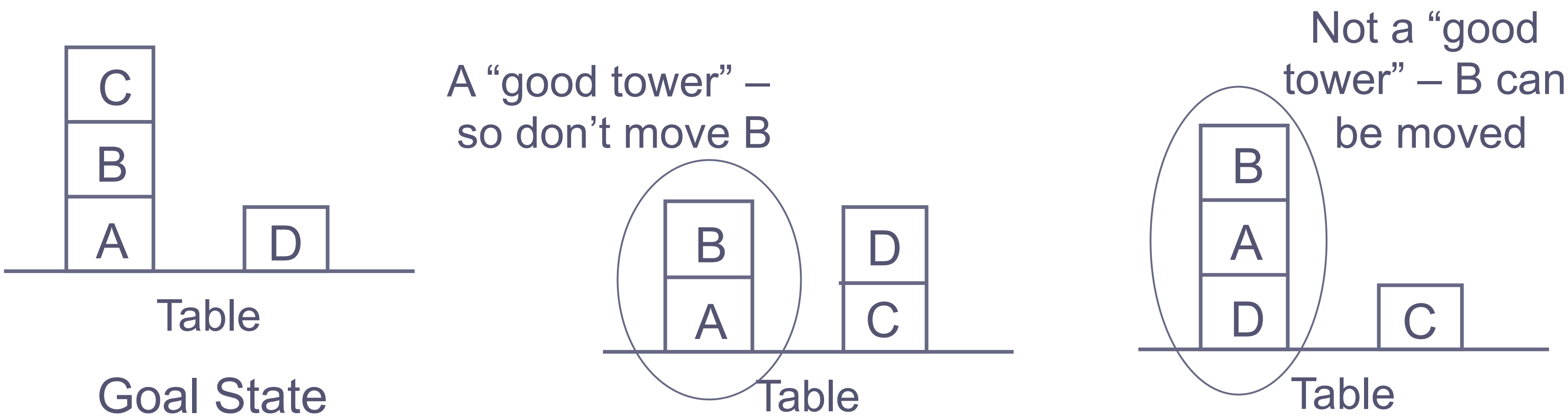


Forward Planning (2)

- The search tree is typically very large. And the number of nodes contained in the tree increases very rapidly with the number of actions considered
- It is impractical to explore the whole tree using, say, unmodified breadth-first search
- But a few simple domain-specific heuristics allow many branches of the tree to be pruned at each stage. This results in an effective planner
- To preserve the completeness of the planning algorithm, the heuristics must never prune branches that could lead a solution, unless they are provably redundant

Blocks World Heuristics

- Here are two useful Blocks World heuristics
  - Don't move *x* from *y* to *z* and then move *x* from *z* back to *y* again
  - Don't demolish a "good tower"
- A *good tower* is a pile of blocks resting on the table that is in the order required by the goal state



Satisfiability Planning

- Partial order planning was the most efficient type of *general purpose* planning algorithm for many years. But its performance is still very poor for some problems, eg: the Blocks World with more than about six blocks!
- In the late 1990s, several planning techniques were developed that out-perform POP on such problems
- One of these is satisfiability planning
- The idea is to translate a given planning problem into a (typically very large) instance of propositional satisfiability
- This new problem can then be solved using an efficient off-the-shelf satisfiability algorithm

Satisfiability Problems

- Satisfiability (SAT) problems are the quitessence of combinatorial problem solving
- 3-variable SAT problems are the basis of the theory of NP-completeness
- Given a *propositional conjunctive normal form* (CNF) formula, such as

$$(A \vee \neg B \vee \neg C) \wedge (B \vee C \vee D)$$

the task is to find an assignment of true or false to each variable such that the whole formula is true

SAT Planning: the Translation

- The challenge is to translate a given planning problem into a propositional conjunctive normal form formula. A CNF formula is a conjunction of disjunctions of negated or un-negated atomic propositions. In other words, it has the form:

$$([\neg]P_{1,1} \vee \ldots \vee [\neg]P_{1,n}) \wedge \ldots \wedge ([\neg]P_{k,1} \vee \ldots \vee [\neg]P_{k,m})$$

- We'll see how to represent effects, preconditions, and frame axioms in CNF
- Recall that:

$$P \leftarrow Q \text{ is equivalent to } P \vee \neg Q$$
$$P \leftarrow Q \wedge R \text{ is equivalent to } P \vee \neg Q \vee \neg R$$

SAT Planning Effects

- Let's consider the same Blocks World example as before
- Our CNF formula must include (propositional equivalents of) *every ground instance* of:

$$\neg Happens(Move(x,y),t) \lor Holds(On(x,y),t+1)$$

- and, for y≠z:

$$\neg Happens(Move(x,y),t) \lor \neg Holds(On(x,z),t+1)$$

- This will include, for example:

$$
\begin{array}{c}
(\neg Happens\text{-}Move\text{-}A\text{-}B\text{-}0 \lor Holds\text{-}On\text{-}A\text{-}B\text{-}1) \land \\
(\neg Happens\text{-}Move\text{-}A\text{-}C\text{-}0 \lor Holds\text{-}On\text{-}A\text{-}C\text{-}1) \land \\
\vdots \\
(\neg Happens\text{-}Move\text{-}A\text{-}B\text{-}1 \lor Holds\text{-}On\text{-}A\text{-}B\text{-}2) \land \\
(\neg Happens\text{-}Move\text{-}A\text{-}C\text{-}1 \lor Holds\text{-}On\text{-}A\text{-}C\text{-}2) \land \\
\vdots
\end{array}
$$

SAT Planning Preconditions

- In addition, the CNF formula will include every ground instance of:

$$\neg \mathit{Happens}(\mathit{Move}(x,y),t) \lor \mathit{Holds}(\mathit{Clear}(x),t)$$
$$\neg \mathit{Happens}(\mathit{Move}(x,y),t) \lor \mathit{Holds}(\mathit{Clear}(y),t)$$

- (These axioms say that if the action happens, the preconditions must hold)

$$
\begin{array}{c}
(\neg \mathit{Happens\text{-}Move\text{-}A\text{-}B\text{-}0} \lor \mathit{Holds\text{-}Clear\text{-}A\text{-}0}) \land \\
(\neg \mathit{Happens\text{-}Move\text{-}A\text{-}B\text{-}0} \lor \mathit{Holds\text{-}Clear\text{-}B\text{-}0}) \land \\
\vdots \\
(\neg \mathit{Happens\text{-}Move\text{-}A\text{-}B\text{-}1} \lor \mathit{Holds\text{-}Clear\text{-}A\text{-}1}) \land \\
\vdots
\end{array}
$$

SAT Planning Frame Axioms

- And, furthermore, the CNF formula will include every ground instance of:

$$Holds(On(x1,y),t+1) \vee \neg Holds(On(x1,y),t) \vee \neg Happens(Move(x2,z),t)$$

such that $x1 \neq x2$

- So we have, for example:

$$(Holds\text{-}On\text{-}B\text{-}C\text{-}1 \vee \neg Holds\text{-}On\text{-}B\text{-}C\text{-}0 \vee \neg Happens\text{-}Move\text{-}D\text{-}A\text{-}0) \wedge$$
$$(Holds\text{-}On\text{-}B\text{-}C\text{-}1 \vee \neg Holds\text{-}On\text{-}B\text{-}C\text{-}0 \vee \neg Happens\text{-}Move\text{-}A\text{-}D\text{-}0) \wedge$$
$$\vdots$$
$$(Holds\text{-}On\text{-}B\text{-}C\text{-}2 \vee \neg Holds\text{-}On\text{-}B\text{-}C\text{-}1 \vee \neg Happens\text{-}Move\text{-}D\text{-}A\text{-}1) \wedge$$
$$\vdots$$

- We will also need formulae ruling out concurrent actions

SAT Planning Initial Situations

- Our CNF formula is getting very large. But it still needs to include a description of the initial state:

(Holds-On-A-B-0 ∧ *Holds-On-B-C-0* ∧ *Holds-On-C-Table-0* ∧ ...

- And we need a description of the goal state. But for this, we have to make a guess at the maximum number of steps *n* the plan will require, and we force the goal state to hold at time *n*. Let's suppose *n*=5.

(Holds-On-A-D-5 ∧ *Holds-On-B-C-5* ∧ *Holds-On-C-Table-5* ∧ ...

- (A full SAT planning system will try the same problem for several values of *n*)

Solving the Satisfiability Problem

- Now we have a large CNF formula Ψ, which we can submit to a satisfiability solver. This will attempt to find an assignment of *True* or *False* to every proposition in Ψ such that Ψ evaluates to *True*
- If this is successful, we can read a plan off the resulting assignment. It is the set of all *Happens* propositions that are assigned *True*
- There are many off-the-shelf solvers that can do this
- The surprise is that this planning method is efficient, even though CNF satisfiability is NP-complete *and* we are dealing with very large CNF formulae

The WalkSAT Algorithm

- WalkSAT is one algorithm (among many) for solving satisfiability problems
- It is a stochastic algorithm. It works by making random flips

```
function WalkSAT(F)
     randomly assign values to all variables in F
     while F not satisfied
          randomly pick unsatisfied clause C in F
          with probability p
               randomly pick variable V in C
          else
               pick V in C to minimise unsatisfied
                    clauses in F
          flip value of V
```

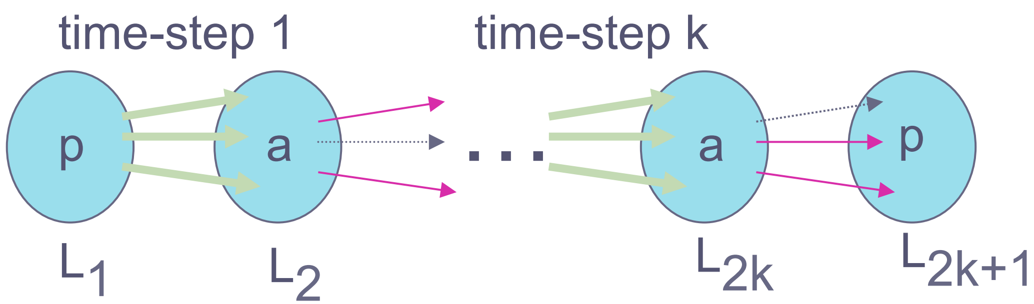
GRAPHPLAN

- GRAPHPLAN is another efficient general-purpose planning algorithm
- It works in two phases
- First it constructs a *planning graph* for *n* time-steps
- Then it tries to find a sub-graph of the plan graph that conforms to certain constraints. This represents the plan
- If it fails then it expands the planning graph to *n+1* time-steps and tries again
- GRAPHPLAN cleverly combines forward and backward reasoning — forwards from the initial state to build the graph, then backwards from the goal state to search for a plan

Imperial College London

Planning Graphs

- The planning graph is organised into levels
- Levels alternate between proposition nodes (L_1 and L_{2k+1}) and action nodes (L_2 and L_{2k})



- The arcs come in three varieties
  - Precondition
  - Add 
  - Delete 

GRAPHPLAN Example 1

- Our example problem domain concerns a loading vehicle that transports goods around a warehouse. It only has enough fuel for one journey
- Consider the following event calculus effect axioms for the actions *Move(r,x,y)*, *Unload(c,r,x)* and *Load(c,r,x)*
- First the *Move* action

$$Initiates(Move(r,x,y),At(r,y),t) \leftarrow$$
$$\qquad HoldsAt(At(r,x),t) \wedge HoldsAt(HasFuel(r),t) \wedge x \neq y$$
$$Terminates(Move(r,x,y),At(r,x),t) \leftarrow$$
$$\qquad HoldsAt(At(r,x),t) \wedge HoldsAt(HasFuel(r),t) \wedge x \neq y$$
$$Terminates(Move(r,x,y),HasFuel(r),t) \leftarrow$$
$$\qquad HoldsAt(At(r,x),t) \wedge HoldsAt(HasFuel(r),t) \wedge x \neq y$$

GRAPHPLAN Example 2

- Now the *Unload and Load* actions

  Initiates(Unload(c,r,x),At(c,x),t) ← *HoldsAt(At(r,x),t)* ∧ *HoldsAt(In(c,r)),t)*

  Terminates(Unload(c,r,x),In(c,r),t) ← *HoldsAt(At(r,x),t)* ∧ *HoldsAt(In(c,r)),t)*

  Initiates(Load(c,r,x),In(c,r),t) ← *HoldsAt(At(r,x),t)* ∧ *HoldsAt(At(c,x)),t)*

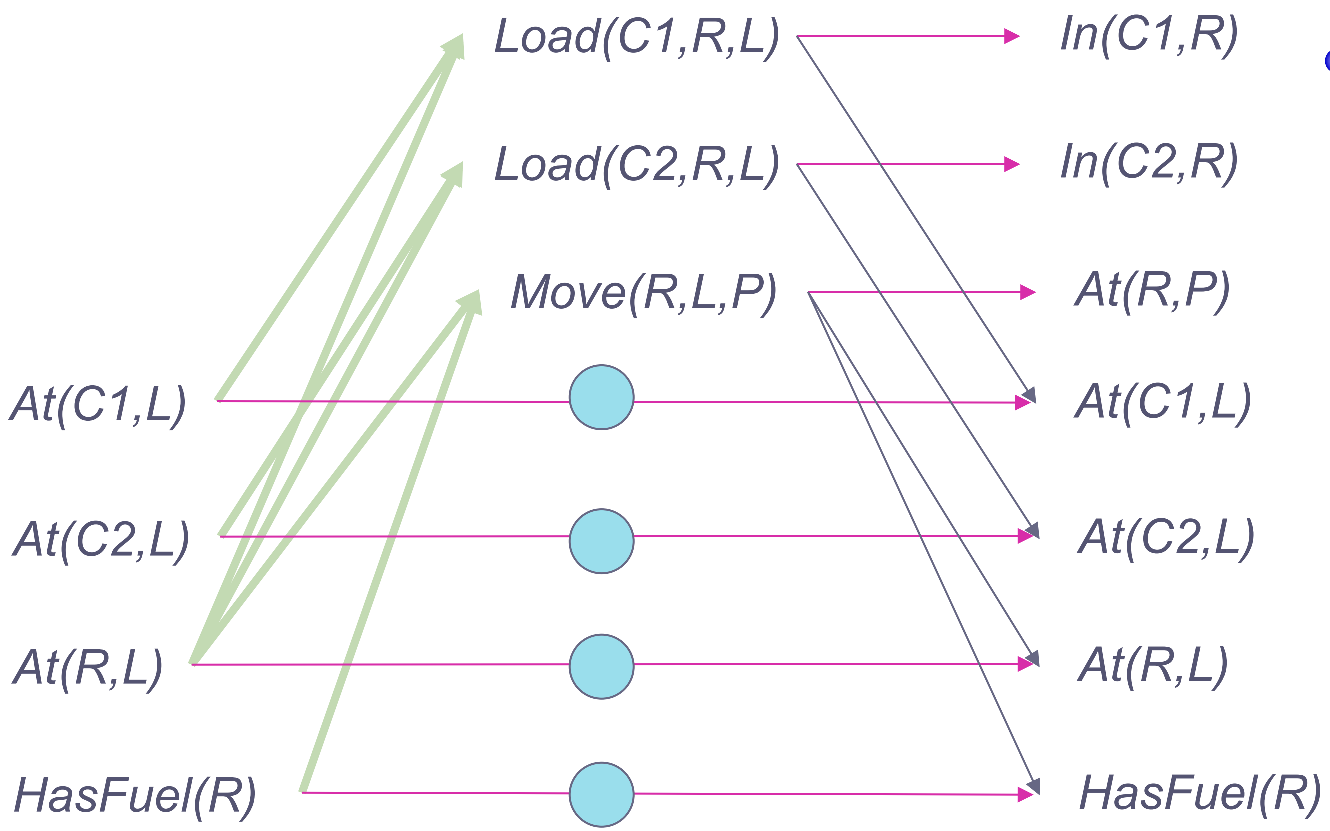
  Terminates(Load(c,r,x),At(c,x),t) ← *HoldsAt(At(r,x),t)* ∧ *HoldsAt(At(c,x)),t)*

- The goal is to make *At(C1,P)* and *At(C2,P)* hold, given the initial situation

  Initially(At(R,L)) *Initially(HasFuel(R))*

  Initially(At(C1,L)) *Initially(At(C2,L))*

Building the Planning Graph

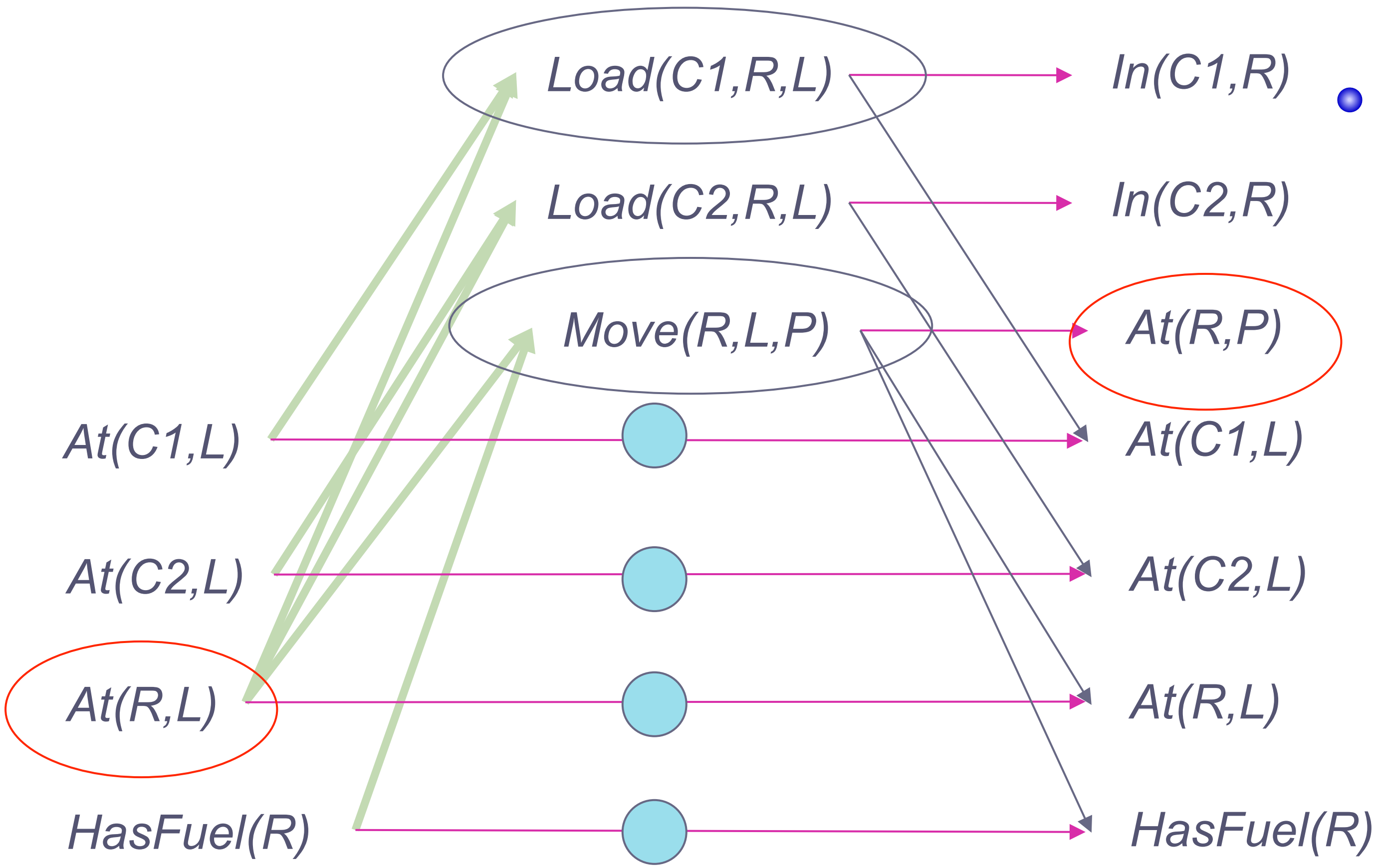


- To handle the persistence of fluents that don't change, GRAPHPLAN uses null actions, or "no-ops"

Mutex Constraints

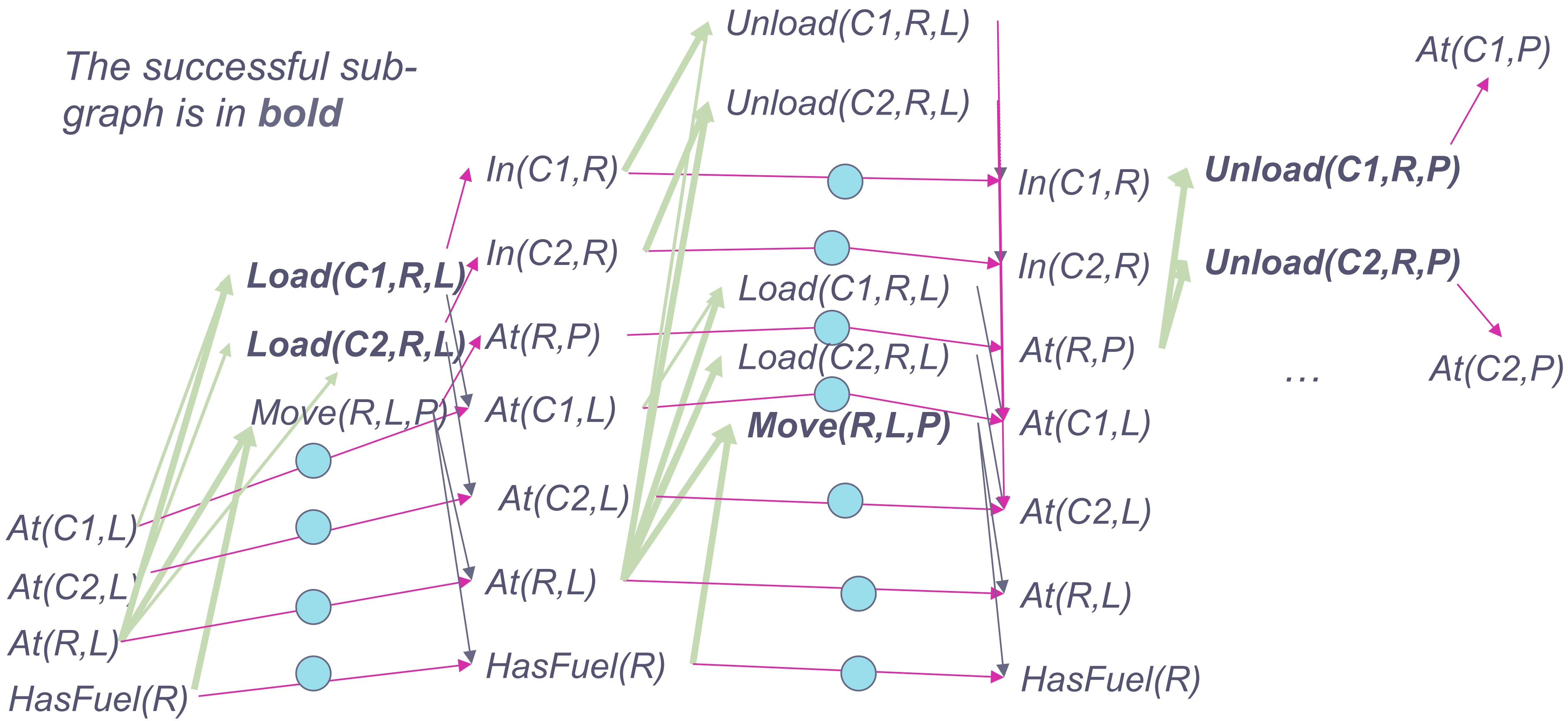
- Actions A_1 and A_2 at time-step T are marked as *mutually exclusive* if
  - A_1 and A_2 have incompatible effects and / or preconditions
  - There exist a precondition of A_1 and a precondition of A_2 marked as mutually exclusive of each other at time-step T-1
- Propositions P_1 and P_2 at time-step T are marked as *mutually exclusive* if
  - Each action A_1 having an add-arc to P_1 is marked as mutually exclusive of each action A_2 having an add-arc to P_2
- The task is then to find a sub-graph containing the goal state in the final level, and with no mutex actions or propositions

Mutex Example



- The *Load(C1,R,L)* and *Move(R,L,P)* actions are mutex because the precondition of *Load* is incompatible with the effect of *Move*

Searching the Graph



Residual Topics

- These are concepts you should be aware of, although there isn’t room for them in the course
- Hierarchical planning
  - This is another domain-specific planning technique that involves the decomposition of a high-level plan into lower-level sub-plans. This process is repeated until low-level executable actions are reached
- Plan execution
  - Finding a plan is one thing. Executing it is another
  - Plan execution is particularly tricky if the world changes during execution in such a way as to frustrate the plan
  - Interleaving planning (especially anytime planning) and execution is often a good idea in fast-changing domains