CLASSICAL PLANNING

What is planning?

- Planning is an AI approach to control
- It is deliberation about actions
- Key ideas
  - We have a **model** of the world
  - Model describes **states** and **actions**
  - Give the planner a **goal** and it outputs a **plan**
  - Aim for **domain independence**
- Planning is search

Classical planning restrictions

1. S is **finite**
2. Environment is **fully observable**
3. Environment is **deterministic**
4. Environment is **static** (no external events)
5. S has a **factored representation**
6. Goals are restricted to **reachability**
7. Plans are **ordered sequences** of actions
8. Actions have **no duration**
9. Planning is done **offline**

Planning Languages

- Languages must represent..
  - States
  - Goals
  - Actions
- Languages must be
  - Expressive for ease of representation
  - Flexible for manipulation by algorithms

We will talk about Planning Domain Definition Language (PDDL)

State Representation

- A state is represented with a conjunction of positive literals
- Using
  - Logical Propositions: $Poor \land Unknown$
  - First order logic literals: $At(Plane1,OMA) \land At(Plane2,JFK)$
- Closed World Assumption
  - What is not stated are assumed false

Goal Representation

- Goal is a <u>partially</u> specified state
- A proposition satisfies a goal if it contains all the atoms of the goal and possibly others..
  - Example: $\mathrm{Rich} \land \mathrm{Famous} \land \mathrm{Miserable}$ satisfies the goal $\mathrm{Rich} \land \mathrm{Famous}$

Representing Actions

- Actions are described in terms of **preconditions** and **effects**.
  - Preconditions are predicates that must be true **before** the action can be applied.
  - Effects are predicates that are made true (or false) **after** the action has executed.
- Sets of similar actions can be expressed as a **schema**.

Example Action…

Applying an Action

- Find a substitution list θ for the variables using the current state description
- Apply the substitution to the propositions in the effect list
- Add the result to the current state description to generate the new state
- Example:
  - Current state: At(P1,JFK) ∧ At(P2,SFO) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)
  - It satisfies the precondition with θ={p/P1,from/JFK, to/SFO)
  - Thus the action Fly(P1,JFK,SFO) is applicable
  - The new current state is: At(P1,SFO) ∧ At(P2,SFO) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO)

Example: Air Cargo

$Action(Load(c,p,a)$
$\quad$ PRECOND: $At(c,a) \wedge At(p,a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
$\quad$ EFFECT: $\neg At(c,a) \wedge In(c,p))$
$Action(Unload(c,p,a)$
$\quad$ PRECOND: $In(c,p) \wedge At(p,a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$
$\quad$ EFFECT: $At(c,a) \wedge \neg In(c,p))$
$Action(Fly(p,from,to)$
$\quad$ PRECOND: $At(p,from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
$\quad$ EFFECT: $\neg\ At(p,from) \wedge At(p,to))$

$Init(At(C1, SFO) \wedge At(C2,JFK) \wedge At(P1,SFO) \wedge At(P2,JFK) \wedge Cargo(C1) \wedge Cargo(C2) \wedge$
$\quad Plane(P1) \wedge Plane(P2) \wedge Airport(JFK) \wedge Airport(SFO))$
$Goal(At(C1,JFK) \wedge At(C2,SFO))$

$[Load(C1,P1,SFO), Fly(P1,SFO,JFK), Load(C2,P2,JFK), Fly(P2,JFK,SFO)]$

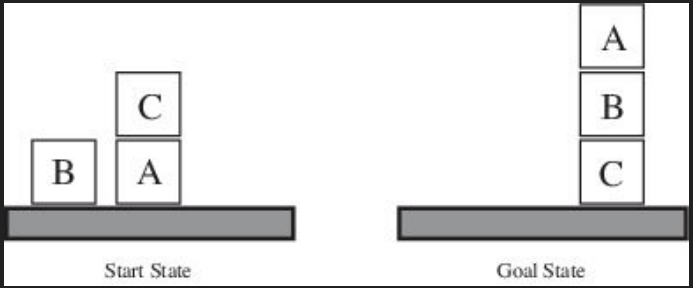
Example: Spare Tire Problem

$Init(At(Flat, Axle) \land At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
  PRECOND: $At(Spare, Trunk)$
  EFFECT: $\neg At(Spare, Trunk) \land At(Spare, Ground))$
$Action(Remove(Flat, Axle),$
  PRECOND: $At(Flat, Axle)$
  EFFECT: $\neg At(Flat, Axle) \land At(Flat, Ground))$
$Action(PutOn(Spare, Axle),$
  PRECOND: $At(Spare, Ground) \land \neg At(Flat, Axle)$
  EFFECT: $\neg At(Spare, Ground) \land At(Spare, Axle))$
$Action(LeaveOvernight,$
  PRECOND:
  EFFECT: $\neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)$
    $\land \neg At(Flat, Ground) \land \neg At(Flat, Axle))$

Example: Blocks World

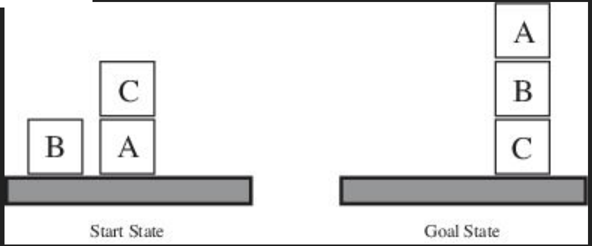
$$\begin{aligned}&Action(Move(b, x, y),\\ &\quad \text{PRECOND}: On(b, x) \wedge Clear(b) \wedge Clear(y),\\ &\quad \text{EFFECT}: On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))\end{aligned}$$

$$\begin{aligned}&Action(MoveToTable(b, x),\\ &\quad \text{PRECOND}: On(b, x) \wedge Clear(b)),\\ &\quad \text{EFFECT}: On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))\end{aligned}$$



Example: Blocks World 2

$Init(On(A, Table) \land On(B, Table) \land On(C, Table)$
$\quad \land Block(A) \land Block(B) \land Block(C)$
$\quad \land Clear(A) \land Clear(B) \land Clear(C))$
$Goal(On(A, B) \land On(B, C))$
$Action(Move(b, x, y),$
$\quad$ PRECOND: $On(b, x) \land Clear(b) \land Clear(y) \land Block(b) \land$
$\qquad (b \neq x) \land (b \neq y) \land (x \neq y),$
$\quad$ EFFECT: $On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y))$
$Action(MoveToTable(b, x),$
$\quad$ PRECOND: $On(b, x) \land Clear(b) \land Block(b) \land (b \neq x),$
$\quad$ EFFECT: $On(b, Table) \land Clear(x) \land \neg On(b, x))$

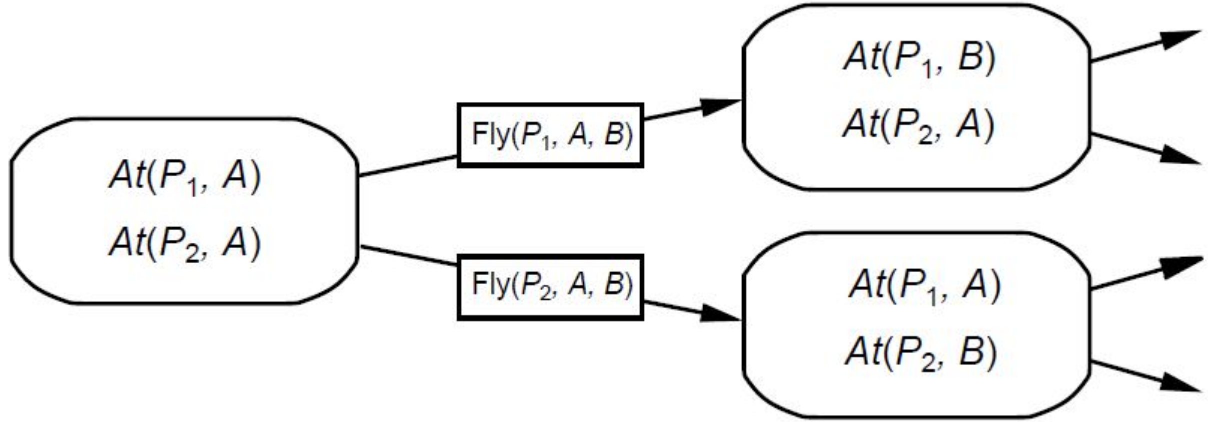


Planning through Search

- Search the space of states connected by actions
- Each action takes a single timestep
- Use familiar algorithms
  - BFS
  - DFS
  - A*
  - …

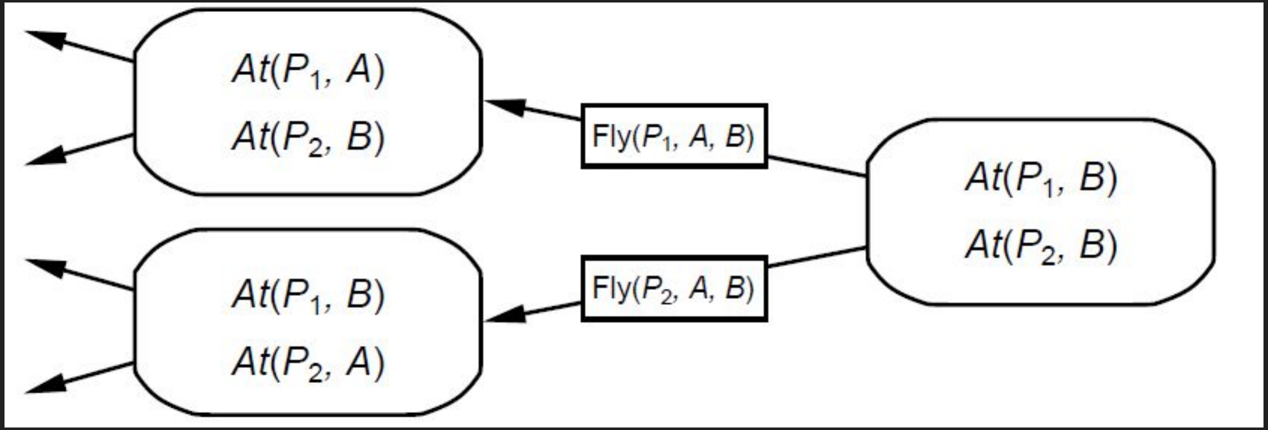
Forward Search

(a)



- Forward (progression) state-space search, starting in the initial state and using the problem’s actions to search forward for the goal state.

Backward Search



- Backward (regression) state-space search: search starting at the goal state(s) and using the inverse of the actions to search backward for the initial state.

Relevant Actions

- An action is relevant
  - In Progression planning, when its preconditions match a subset of the current state
  - In Regression planning, when its effects match a subset of the current goal state

Planning Graph

- A planning graph consists in a sequence of *levels* that correspond to time steps
  - Level 0 is the initial state
- Each level contains a set of literals that *could* be true at this time step
- Each level contains a set of actions that *could* be applied at this time step

Have Cake and Eat it Too

Init(*Have*(*Cake*))

Goal(*Have*(*Cake*) ∧ *Eaten*(*Cake*))

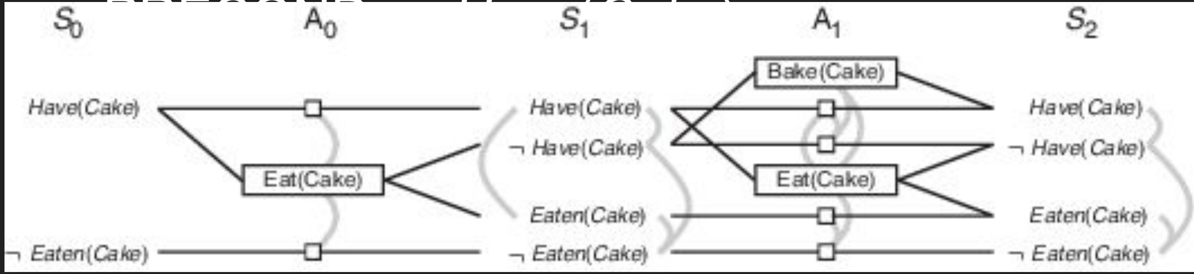
Action(*Eat(Cake*)

PRECOND: *Have*(*Cake*)

EFFECT: ¬*Have*(*Cake*) ∧ *Eaten*(*Cake*))

Action(*Bake(Cake*)

PRECOND: ¬*Have*(*Cake*)



Planning Graph

- Level *A*0 contains all the actions that *could* occur in state S0.
    - Persistence actions (small boxes) represent the fact that one literal is not modified.
    - Mutual exclusions (*mutexes*, gray lines) represent *conflicts* between actions.
- To go from level 0 to the level 1, you pick a set of non exclusives actions (for instance, action *Eat*(*Cake*))

- Level *S*1 contains all the literals that could result from picking any subset of actions in *A*0.
- Mutexes represent *conflicts* between literals.

How to build the planning graph

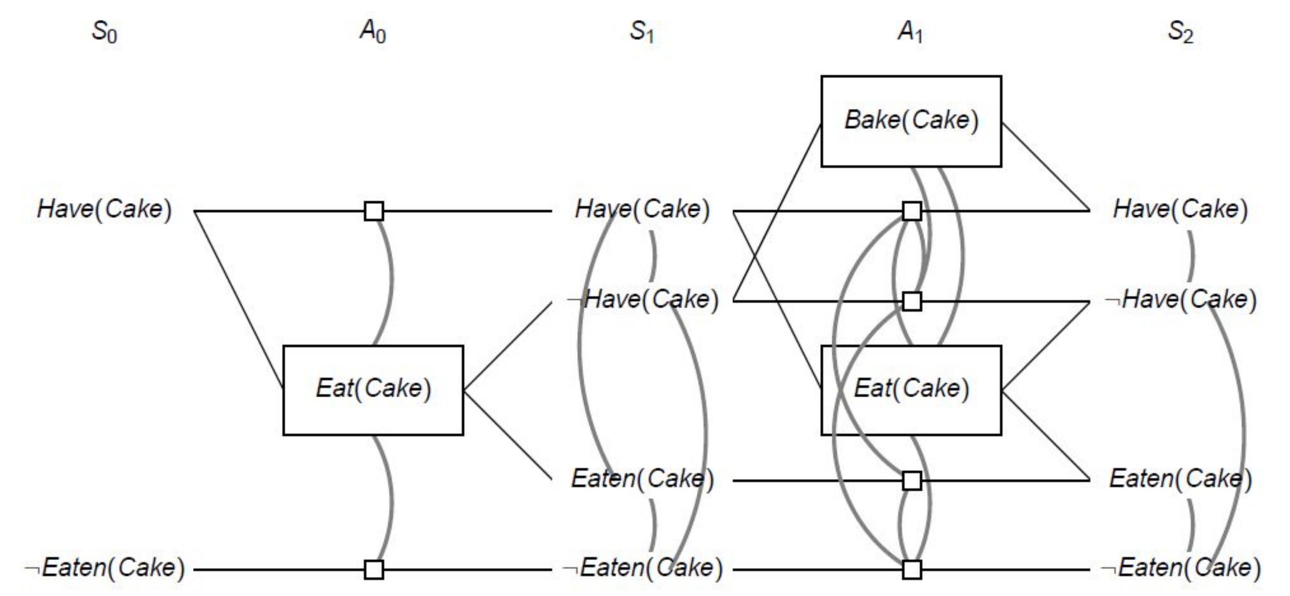
1. Start from S_0
2. $i = 0$
3. Find all the actions of A_{i+1} applicable in S_i given the mutexes
4. Compute the mutexes between the actions of A_{i+1}
5. Compute the literals reachable in S_{i+1}
6. Compute the mutexes in S_{i+1}
7. If $S_{i+1} \neq S_i$, then increment i by 1 and go to 3

Mutexes

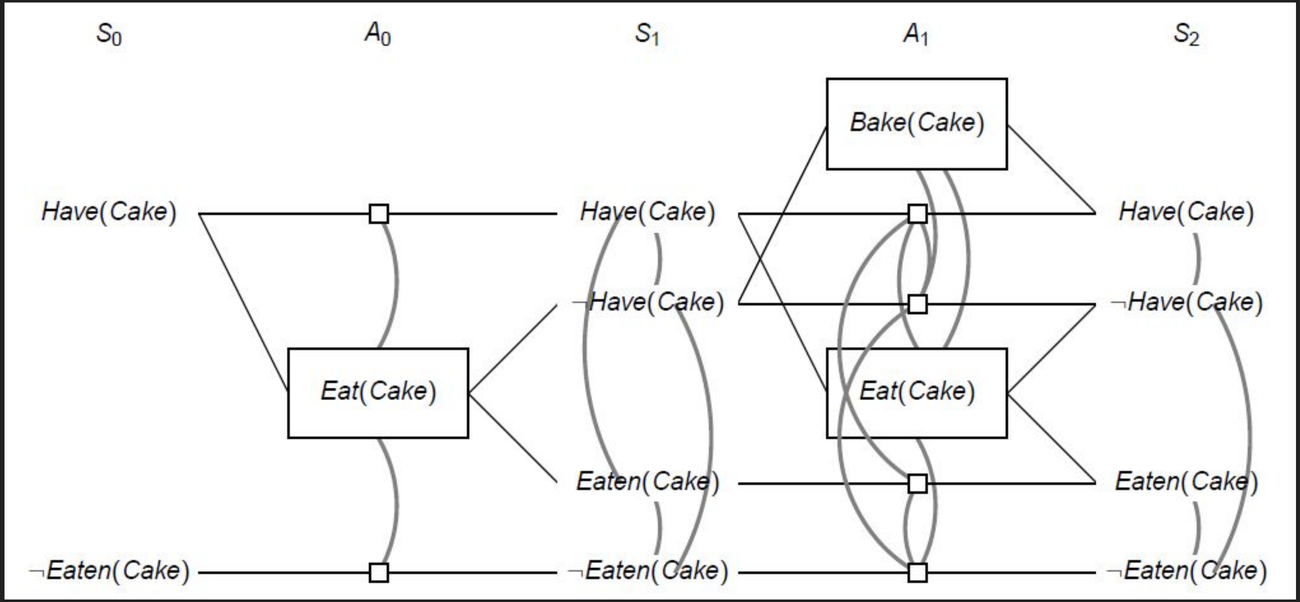
- A mutex between two actions indicates that it is impossible to perform these actions in parallel.
- A mutex between two literals indicates that it is impossible to have these both literals true at this stage.

How to compute mutexes

- Actions
  - Inconsistent effects: two actions that lead to inconsistent effects
  - Interference: an effect of the first action negates the precondition of the other action
  - Competing needs: a precondition of the first action is mutually exclusive with a precondition of the second action.
- Literals
  - one literal is the negation of the other one
  - Inconsistency support: each pair of action achieving the two literals are mutually exclusive.



- Inconsistent effects: Eat(Cake) & noop of Have(Cake) disagree on effect Have(Cake)
- Interference: Eat(Cake) negates precondition of the noop of Have(Cake)
- Competing needs: Bake(Cake) & Eat(Cake): compete on Have(Cake) precondition



- In S1, Have(Cake) & Eaten(Cake) are mutex
- In S2, they are not because Bake(Cake) & the noop of Eaten(Cake) are not mutex

Plan Graph Summary

- Continue until two consecutive levels are identical.
- Graph indicates which actions are not executable in parallel
- Construction polynomial
- No choice which action to take, only indicate which are forbidden to occur in parallel

Planning graph for heuristic search

- Using the planning graph to estimate the number of actions to reach a goal

- If a literal does not appear in the final level of the planning graph, then there is no plan that achieve this literal!
    - $h = \infty$

Heuristics

- **max-level:** take the maximum level where any literal of the goal first appears
  - admissible
- **level-sum:** take the sum of the levels where any literal of the goal first appears
  - not admissible, but generally efficient (specially for independent subplans)
- **set-level:** take the minimum level where all the literals of the goal appear and are free of mutex
  - admissible

Graphplan Algorithm

- Extracts a plan directly from the plan graph

```
GRAPHPLAN(problem) returns solution or failure
graph ← INITIALPLANNINGGRAPH(problem)
goals ← GOALS[problem]
loop do
    if goals all non-mutex in last level of graph then do
       solution ← EXTRACTSOLUTION(graph,goals,LENGTH(graph))
       if solution ≠ failure then return solution
       else if  NOSOLUTIONPOSSIBLE(graph) then return failure
    graph ← EXPANDGRAPH (graph,problem)
```

Questions?