Planning

COMP3431 Robot Software Architectures
Planning

A planner finds sequences of actions that will cause transitions from an initial state through intermediates states to a goal state
Actions

- Transitions from one state to the next are achieved by *actions*.
- Must specify how actions work
- Must work out correct sequence of actions to reach goal
Action Models

- Action action(<parameters>)
  - PRECOND: <conditions that must be true to apply this actions>
  - EFFECTS: <conditions that become true or false after executing the action>
Action Example

**Action** Fly(p, from, to)

**PRECOND:** Plane(p) ∧ At(p, from) ∧ Airport(from) ∧ Airport(to)

**EFFECT:** ¬At(p, from) ∧ At(p, to))

- positive and negative literals in effects can be separated into an *add list* and an *delete list*
Example

Init:  
Airport(MEL) \land Airport(SYD) \land Plane(P_1) \land Plane(P_2) \land Cargo(C_1) \land Cargo(C_2) \land 
At(C_1, SYD) \land At(C_2, MEL) \land At(P_1, SYD) \land At(P_2, MEL)

Goal:  
At(C_1, MEL) \land At(C_2, SYD)

Action  
Load(c, p, a)
  PRECOND:  
At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)
  EFFECT:  
\neg At(c, a) \land In(c, p)

Action  
Unload(c, p, a)
  PRECOND:  
In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)
  EFFECT:  
At(c, a) \land \neg In(c, p)

Action  
Fly(p, from, to)
  PRECOND:  
At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)
  EFFECT:  
\neg At(p, from) \land At(p, to)
Progression and Regression

- Forward Search

- Backward Search
Backward Regression

\[ g' = (g - \text{Add}(a)) \cup \text{Precond}(a) \]

- \( g' \) is the regression from goal \( g \) over action \( a \)
- I.e. going backwards from \( g \), we look for an action, \( a \), that has preconditions and effects that satisfy \( g' \)
Planning and TR Programs

Action :-
- goal → do_nothing
- precondition → action
- ....
- start → action

- TR Programs list actions from a plan, keeping preconditions
- Each rule below should be the regression of the rule above
Sussman’s Anomaly

• Goal: On(A, B) ∧ On(B, C)

• Try achieving On(A, B) first

[move(c,a,floor), move(a,floor,b), **move(a,b,floor)**, move(b,floor,c)]

• Trying On(B, C) first

[move(b,floor,c), **move(b,c,floor)**, move(c,a,floor), move(a,floor,b)]

• Should be:

[move(c,a,floor), move(b,floor,c), move(a,floor,b)]
WARPLAN


• WARPLAN tries to interleave actions by protecting goals.
  • Achieve on(A,B): [move(c,a,floor), move(a,floor,b)]
  • Protect on(A,B)
  • Now try on(B,C) by appending actions to end of plan
    • If it tries to undo a protected goal, move backwards through plan and try to slot new plan in.
Warplan

- \([\text{move}(c,a,\text{floor}), \text{move}(a,\text{floor},b), \text{move}(a,b,\text{floor}), \ldots]\)

- \([\text{move}(c,a,\text{floor}), \ldots, \text{move}(a,\text{floor},b)]\)

  Try inserting plan for on(B,C) here

- check that goals before and after are preserved
Partially Ordered Plans

Partial Order Plan:

Start

Left Sock

Right Sock

LeftSockOn

RightSockOn

Left Shoe

Right Shoe

LeftShoeOn, RightShoeOn

Finish

Total Order Plans:

Start

Start

Start

Start

Start

Start

Start

Right Sock

Right Sock

Left Sock

Left Sock

Right Sock

Right Sock

Left Sock

Left Sock

Right Shoe

Right Shoe

Right Shoe

Right Shoe

Left Shoe

Left Shoe

Left Shoe

Left Shoe

Right Sock

Right Sock

Right Sock

Right Sock

Left Sock

Left Sock

Left Sock

Left Sock

Finish

Finish

Finish

Finish

Finish

Finish

Finish

Finish
Partial-Order Planning

Init: Tire(Flat) ∧ Tire(Spare) ∧ At(Flat, Axle) ∧ At(Spare, Boot)

Goal: At (Spare, Axle)

Action Remove(obj, loc)
   PRECOND: At(obj, loc)
   EFFECT: ¬ At(obj, loc) ∧ At(obj, Ground)

Action PutOn(t, Axle)
   PRECOND: Tire(t) ∧ At(t, Ground) ∧ ¬ At(Flat, Axle)
   EFFECT: ¬ At(t, Ground) ∧ At(t, Axle)
Partial-Order Planning

Start

At(Spare,Boot)
At(Flat,Axle)

At(Spare,Axle)
Finish

At(Spare,Boot)

Remove(Spare,Boot)

Start

At(Spare,Boot)
At(Flat,Axle)

At(Spare,Axle)

PutOn(Spare,Axle)
Finish

¬ At(Flat,Axle)

At(Spare,Boot)

Remove(Spare,Boot)

Start

At(Spare,Boot)
At(Flat,Axle)

At(Spare,Axle)

PutOn(Spare,Axle)
Finish

¬ At(Flat,Axle)

At(Spare,Boot)

Remove(Spare,Boot)

Start

At(Spare,Boot)
At(Flat,Axle)

At(Spare,Axle)

PutOn(Spare,Axle)
Finish

¬ At(Flat,Axle)

At(Spare,Boot)

Remove(Spare,Boot)

Start

At(Spare,Boot)
At(Flat,Axle)

At(Spare,Axle)

PutOn(Spare,Axle)
Finish

¬ At(Flat,Axle)

At(Spare,Boot)

Remove(Spare,Boot)
Forward Planning

- Forward planners are now among the best.
- Use heuristics to estimate costs
- Possible to use heuristic search, like $A^*$, to reduce branching factor.
Planning graphs

• Used to achieve better heuristic estimates.
  • A solution can also directly extracted using GRAPHPLAN.

• Consists of a sequence of levels that correspond to time steps in the plan.
  • Level 0 is the initial state.

• Each level consists of a set of literals and a set of actions.
  • Literals = all those that could be true at that time step, depending upon the actions executed at the preceding time step.
  • Actions = all those actions that could have their preconditions satisfied at that time step, depending on which of the literals actually hold.
Planning graphs

• Records only a restricted subset of possible negative interactions among actions

• They work only for propositional problems.
Example

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)
   EFFECT: Have(Cake)
Cake example

- Start at level S0 and determine action level A0 and next level S1.
  - A0 >> all actions whose preconditions are satisfied in the previous level.
  - Connect precond and effect of actions S0 --> S1
  - Inaction is represented by persistence actions.
- Level A0 contains the actions that could occur
  - Conflicts between actions are represented by mutex links
• Level S1 contains all literals that could result from picking any subset of actions in A0
  • Conflicts between literals that can not occur together (as a consequence of the selection action) are represented by mutex links.
  • S1 defines multiple states and the mutex links are the constraints that define this set of states.
• Continue until two consecutive levels are identical: leveled off
  • Or contain the same amount of literals (explanation follows later)
A mutex relation holds between two actions when:
- Inconsistent effects: one action negates the effect of another.
- Interference: one of the effects of one action is the negation of a precondition of the other.
- Competing needs: one of the preconditions of one action is mutually exclusive with the precondition of the other.

A mutex relation holds between two literals when (inconsistent support):
- If one is the negation of the other OR
- if each possible action pair that could achieve the literals is mutex.
PG and heuristic estimation

- PG's provide information about the problem
  - A literal that does not appear in the final level of the graph cannot be achieved by any plan.
    - Useful for backward search (cost = inf).
  - Level of appearance can be used as cost estimate of achieving any goal literals = level cost.
- Small problem: several actions can occur
  - Restrict to one action using serial PG (add mutex links between every pair of actions, except persistence actions).
  - Cost of a conjunction of goals? Max-level, sum-level and set-level heuristics.
- PG is a relaxed problem.
The GRAPHPLAN Algorithm

How to extract a solution directly from the PG

```
function GRAPHPLAN(problem) return solution or failure
    graph ← INITIAL-PLANNING-GRAPH(problem)
    goals ← GOALS[problem]
    loop
        if goals all non-mutex in last level of graph then
            solution ← EXTRACT-SOLUTION(graph, goals, LENGTH(graph))
            if solution ≠ failure then return solution
            else if NO-SOLUTION-POSSIBLE(graph) then return failure
        graph ← EXPAND-GRAPH(graph, problem)
```
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, Groundp) \land \neg At(Flat, Axle)
  EFFECT: At(Spare, Axle) \land \neg At(Spare, Ground))

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) )
Initially the plan consists of literals from the initial state and literals from the closed world assumption (S0).

Add actions whose preconditions are satisfied by EXPAND-GRAPH (A0).

Also add persistence actions and mutex relations.

Add the effects at level S1

Repeat until goal is in level Si
EXPAND-GRAPH also looks for mutex relations

- Inconsistent effects
  - E.g. Remove(Spare, Trunk) and LeaveOverNight due to At(Spare, Ground) and not At(Spare, Ground)

- Interference
  - E.g. Remove(Flat, Axle) and LeaveOverNight At(Flat, Axle) as PRECOND and not At(Flat, Axle) as EFFECT

- Competing needs
  - E.g. PutOn(Spare, Axle) and Remove(Flat, Axle) due to At(Flat, Axle) and not At(Flat, Axle)

- Inconsistent support
  - E.g. in S2, At(Spare, Axle) and At(Flat, Axle)
In S2, the goal literals exist and are not mutex with any other
  • Solution might exist and EXTRACT-SOLUTION will try to find it
  • EXTRACT-SOLUTION can use Boolean CSP to solve the problem or a search process:
    • Initial state = last level of PG and goal goals of planning problem
    • Actions = select any set of non-conflicting actions that cover the goals in the state
    • Goal = reach level S0 such that all goals are satisfied
    • Cost = 1 for each action.
Termination? YES

PG are monotonically increasing or decreasing:
  - Literals increase monotonically
  - Actions increase monotonically
  - Mutexes decrease monotonically

Because of these properties and because there is a finite number of actions and literals, every PG will eventually level off!
Extracting the Plan

- Heuristic forward search planners, like Lama, use A* to find path from start to goal.
  - Cost is based on level in graph.
- Answer Set Programming is a very efficient type of constraint solving that is fast but only works on propositional representations.