UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

Artificial Intelligence

Deterministic Planning

LESSON 19

prof. Antonino Staiano

M.Sc. In ''Machine Learning e Big Data'' - University Parthenope of Naples

Introduction

- Planning is deciding what to do based on an agent's ability, goals, and the state of the world
  - It is finding a sequence of actions to solve a goal
- Planning combines the two major areas of AI we have covered so far: search and logic
  - The combination enables planners to progress from toy problems, limited to around a dozen actions and states, to real-world industrial applications involving millions of states and thousands of actions
- Assumptions
  - The world is deterministic
  - There are no exogenous events outside the agent's control that change the state of the world
  - The agent knows what state it is in
  - Time progresses discretely from one state to the next
  - Goals are predicates of states that need to be achieved or maintained

Definition of Classical Planning

- Classical planning is defined as the task of finding a sequence of actions to accomplish a goal in a discrete, deterministic, static, fully observable environment
- Planning Domain Definition Language (PDDL) is a factored representation
  - Basic PDDL can handle classical planning domains, and extensions can handle non-classical domains that are continuous, partially observable, concurrent, and multi-agent
  - State: represented as a conjunction of ground atomic fluents
    - Recall
      - Ground -> no variables
      - fluent -> an aspect of the world that changes over time
      - Ground atomic -> there is a single predicate, with possible arguments being constants
  - uses database semantics: the closed-world assumption means that any fluents that are not mentioned are false
    - A semantic used in logic to allow straightforward logical sentences
  - Example
    - Poor ∧ Unknown (a state of a hapless agent?)
    - *At(Truck1, Melbourne) ∧ At(Truck2, Sydney)* (a state in a package delivery problem)

Action Schema

- An action schema represents a family of ground actions
- The schema consists of the action name, a list of all the variables used in the schema, a precondition (what must be true before the action) , and an effect (what becomes true after the action)

  Action(Fly(p, from, to), *// action schema for flying a plane from one location to another*

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

  EFFECT: ¬At(p, from) ∧ At(p, to))

- The precondition and the effect are each conjunctions of literals, that is, positive or negated atomic sentences
- It is also possible to choose constants to instantiate the variables (ground action)

  Action(Fly(P_1, SFO, JFK),

  PRECOND: At(P_1, SFO) ∧ Plane(P_1) ∧ Airport(SFO) ∧ Airport(JFK)

  EFFECT: ¬At(P_1, SFO) ∧ At(P_1, JFK))

Action Schema

- A ground action a is applicable in state s if s entails the precondition of a
  - that is, if every positive literal in the precondition is in s and every negated literal is not
- Executing an applicable action a in state s defines a state s'
  - represented by the set of fluents formed by starting with s, removing the fluents that appear as negative literals in the action's effect (delete list, i.e. DEL(a)), and adding the fluents that are positive literals in the action's effect (add list, ADD(a))
  - $s' = RESULT(s,a) = (s - DEL(a)) \cup ADD(a)$
- Example
  - the action Fly(P1,SFO,JFK) would remove the fluent At(P1,SFO) and add the fluent At(P1,JFK)

$$Action(Fly(P_1, SFO, JFK),$$
$$PRECOND: At(P_1, SFO) \wedge Plane(P_1) \wedge Airport(SFO) \wedge Airport(JFK)$$
$$EFFECT: \neg At(P_1, SFO) \wedge At(P_1, JFK))$$

Planning Domain

- A set of action schemas serves as a definition of a planning domain
  - A specific problem within the domain is defined with the addition of an initial state and a goal
  - The initial state is a conjunction of ground fluents (*Init*)
  - The goal is just like a precondition: a conjunction of literals (positive or negative) that may contain variables
    - Example
      - $At(C_1, SFO) \land \neg At(C_2, SFO) \land At(p, SFO)$
        - Any state in which cargo C_1 is at SFO but C_2 is not, and in which there is a plane at SFO

Example Domain: Air Cargo Transport

- Air cargo transport problem involving loading and unloading cargo and flying it from place to place
  - The problem can be defined with three actions: *Load*, *Unload*, and *Fly*

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\quad \land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\quad \land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
$\quad$ PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $\neg At(c, a) \land In(c, p))$
$Action(Unload(c, p, a),$
$\quad$ PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
$\quad$ EFFECT: $At(c, a) \land \neg In(c, p))$
$Action(Fly(p, from, to),$
$\quad$ PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
$\quad$ EFFECT: $\neg At(p, from) \land At(p, to))$

Figure 11.1 A PDDL description of an air cargo transportation planning problem

At predicates are maintained properly

  - The following plan is a solution to the problem:
    - [Load(C_1,P_1,SFO), Fly(P_1,SFO,JFK), Unload(C_1,P_1,JFK), Load(C_2,P_2,JFK), Fly(P_2,JFK,SFO), Unload(C_2,P_2,SFO)]

Example Domain: The Spare Tire Problem

- Consider the problem of changing a flat tire
  - The goal is to have a good spare tire properly mounted onto the car's axle
    - the initial state has a flat tire on the axle and a good spare tire in the trunk

$$
\begin{aligned}
&Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))\\
&Goal(At(Spare, Axle))\\
&Action(Remove(obj, loc),\\
&\quad \text{PRECOND: } At(obj, loc)\\
&\quad \text{EFFECT: } \neg At(obj, loc) \land At(obj, Ground))\\
&Action(PutOn(t, Axle),\\
&\quad \text{PRECOND: } Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle) \land \neg At(Spare, Axle)\\
&\quad \text{EFFECT: } \neg At(t, Ground) \land At(t, Axle))\\
&Action(LeaveOvernight,\\
&\quad \text{PRECOND:}\\
&\quad \text{EFFECT: } \neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)\\
&\qquad \land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))
\end{aligned}
$$

Figure 11.2 The simple spare tire problem.

- The following plan is a solution to the problem:
  - [Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)]

Example Domain: The Block World

- The famous blocks world planning domain
  - It consists of a set of cube-shaped blocks sitting on an arbitrarily large table
- On(b,x)
  - block b is on x, where x is either another block or the table
- Move(b, x, y)
  - block b from the top of x to the top of y
  - One of the preconditions on moving b is that no other block be on it
    - In first-order logic, this would be $\neg\exists x\ On(x,b)$ or $\forall x\ \neg On(x,b)$
    - Basic PDDL does not allow quantifiers, so we introduce a predicate Clear(x)
- Clear(x)
  - Nothing on x (true when nothing is on x)

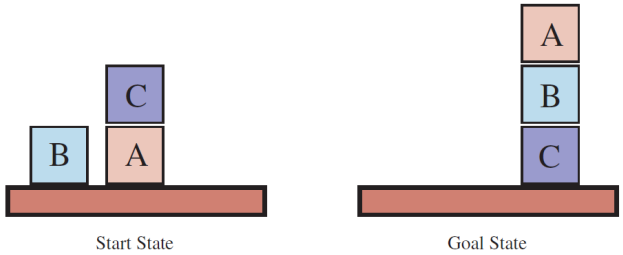


Figure 11.3 Diagram of the blocks-world problem in Figure 11.4.

- The action *Move* moves a block *b* from *x* to *y* if both *b* and *y* are clear, after, *b* is still clear but *y* is not
  - A first attempt at the *Move* schema is

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

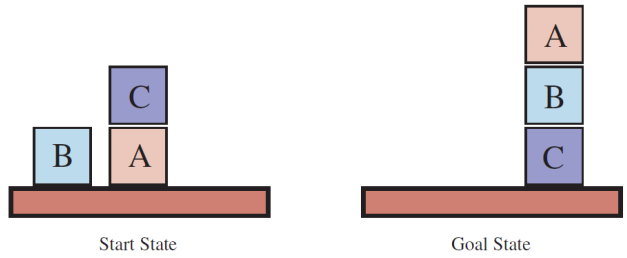
The Block World

Move schema

Action(MoveToTable(b, x),
PRECOND:On(b,x) ∧ Clear(b),
EFFECT:On(b,Table) ∧ Clear(x) ∧ ¬On(b,x))

Clear(x) is interpreted as "there is a clear space on x to hold a block."
Under this interpretation, Clear(Table) will always be true

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



ıre 11.3 Diagram of the blocks-world problem in Figure 11.4.

Figure 11.4 A planning problem in the blocks world: building a three-block tower. One solution is the sequence $[MoveToTable(C,A), Move(B,Table,C), Move(A,Table,B)]$.

Algorithms for Classical Planning

- Forward state-space search for planning
  - Start at initial state
  - To determine the applicable actions we unify the current state against the preconditions of each action schema
  - For each unification that successfully results in a substitution, we apply the substitution to the action schema to yield a ground action with no variables
- Backward search for planning
  - Start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state
  - Consider relevant actions at each step
  - Reduces branching factor
  - A relevant action is one with an effect that unifies with one of the goal literals, but with no effect that negates any part of the goal

Forward state-space search for planning

- Planning problems can be solved by using heuristic search algorithms
  - The states in the search space are ground states where the fluents are either true or not
  - The goal state has all positive fluents in the problem's goal and none of the negative fluents
  - The applicable actions in a state s, Actions(s), are grounded instantiations of the action schemas
    - That is, constant values have replaced variables
    - They are determined by unifying the current state against the preconditions of each action schema
    - The identified substitution is applied to the action schema providing a ground action with no variables
  - Each schema may unify in multiple ways
    - If an action has multiple literals in the precondition, then each of them can potentially be matched against the current state in several ways
    - E.g., in spare tire, the Remove action, the precondition At(obj, loc) matches against the initial state in two ways, resulting in the two substitutions {obj/Flat,loc/Axle} and {obj/Spare,loc/Trunk}
    - This can lead to search graphs whose depth to the solution has an unfeasible number of nodes
      - An accurate heuristics is needed to make forward search feasible

Backward search for planning

- Starts at the goal and works backward, applying actions to find a sequence of steps reaching the initial state
- Focuses on relevant actions, i.e., with an effect that unifies with one of the goal literals, but with no effect that negates any part of the goal
  - For example, the goal $\neg Poor \wedge Famous$ considers actions with the sole effect Famous but not those with $Poor \wedge Famous$
- Regression Process:
- Given a goal g and an action a, the regression from g over a, yields a state g′ whose positive and negative literals are given by
  - POS(g′) = (POS(g) - ADD(a)) ∪ POS(Precond(a))
  - NEG(g′) = (NEG(g) - DEL(a)) ∪ NEG(Precond(a))
- Preconditions must hold before action execution, but added/deleted literals need not be true before

Backward search for planning

- Some care is required when there are variables in g and a
  - Suppose the goal is to deliver a specific piece of cargo to SFO: $At(C_2,SFO)$
  - The *Unload* action schema has the effect *At(c,a)*
  - The unification with the goal provides the substitution $\{c/C_2,a/SFO\}$
    - applying that substitution to the schema gives us a new schema that captures the idea of using any plane that is at SFO:

      Action(Unload(C_2, p',SFO),

      PRECOND:In(C_2,p') ∧ At(p',SFO) ∧ Cargo(C_2) ∧ Plane(p') ∧ Airport(SFO)

      EFFECT:At(C_2,SFO) ∧ ¬In(C_2,p'))
    - p is replaced with p' (standardizing apart variable names) to avoid conflicts between different variables with the same name
  - The regressed state description gives the new goal
    - g' =In(C_2,p') ∧ At(p',SFO) ∧ Cargo(C_2) ∧ Plane(p') ∧ Airport(SFO)
- For most problem domains backward search keeps the branching factor lower than forward search

Algorithms for Classical Planning

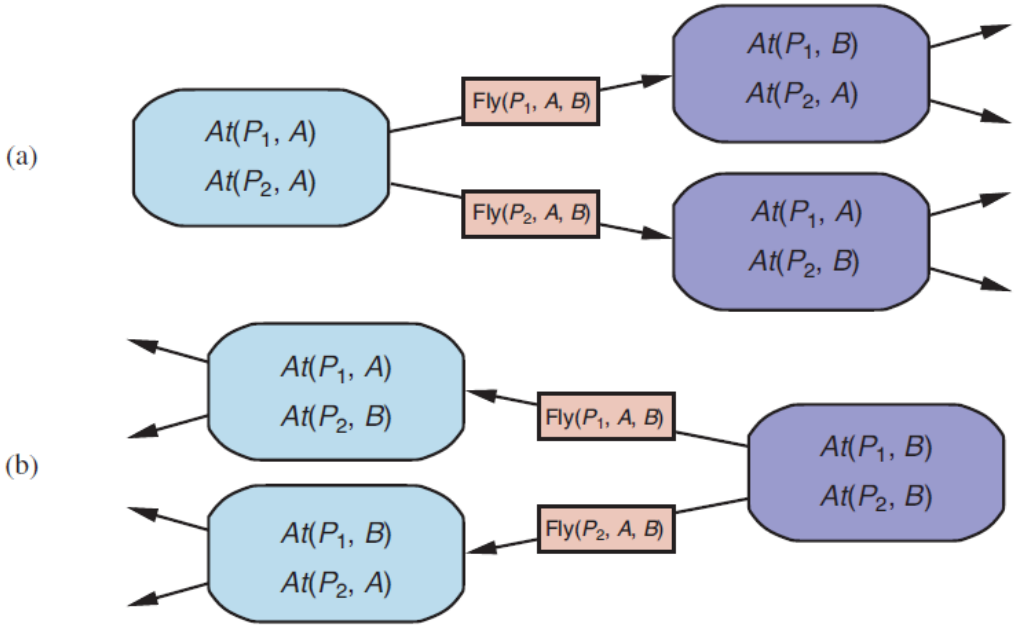


Figure 11.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of ground states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through state descriptions, starting at the goal and using the inverse of the actions to search backward for the initial state.

Algorithms for Classical Planning

- Other classical planning approaches
  - An alternative called partial-order planning represents a plan as a graph rather than a linear sequence:
    - each action is a node in the graph,
    - for each precondition of the action there is an edge from another action (or from the initial state) that indicates that the predecessor action establishes the precondition
      - So we could have a partial-order plan that says that actions Remove(Spare, Trunk) and Remove(Flat, Axle) must come before PutOn(Spare, Axle), without saying which of the two Remove actions should come first
    - We search in the space of plans rather than world-states, inserting actions to satisfy conditions

Heuristics for Planning

- Forward and backward search is inefficient without a good heuristic function h(s)
  - h(s) estimates the distance from a state s to the goal
  - if we can derive an admissible heuristic for this distance then we can use A∗ search to find optimal solutions
  - an admissible heuristic can be derived by defining a relaxed problem that is easier to solve
    - the exact cost of a solution to this easier problem then becomes the heuristic for the original problem
  - A search problem is a graph where the nodes are states, and the edges are actions
    - There are two main ways we can relax this problem to make it easier:
      - by adding more edges to the graph, making it easier to find a path,
      - or by grouping multiple nodes, forming an abstraction of the state space that has fewer states, and thus is easier to search

Heuristics for Planning

- Heuristics that add edges to the graph
  - Ignore preconditions heuristic: drops all preconditions from actions
    - Every action becomes applicable
    - Any single goal fluent can be achieved in one step (if there are any applicable actions)
    - It's possible to ignore only selected preconditions of actions, e.g., consider the 8-puzzle encoded as a planning problem with a single schema

      $Action(Slide(t,s_1,s_2),$

      $PRECOND:On(t,s_1) \wedge Tile(t) \wedge Blank(s_2) \wedge Adjacent(s_1,s_2)$

      $EFFECT:On(t,s_2) \wedge Blank(s_1) \wedge \neg On(t,s_1) \wedge \neg Blank(s_2))$
    - if we remove the preconditions $Blank(s_2) \wedge Adjacent(s_1,s_2)$ then any tile can move in one action to any blank space and we get the number-of-misplaced-tiles heuristic
    - If we remove only the $Blank(s_2)$ precondition then we get the Manhattan distance heuristic

Heuristics for Planning

- Heuristics that add edges to the graph
  - Ignore-delete-lists heuristic: removing the delete lists from all actions (i.e., removing all negative literals from effects)
  - That makes it possible to make monotonic progress toward the goal
    - no action will ever undo the progress made by another action
  - It turns out it is still NP-hard to find the optimal solution to this relaxed problem, but an approximate solution can be found in polynomial time by hill climbing

Heuristics for Planning

- Domain-independent pruning
  - Many states are just variants of other states
    - symmetry reduction: prune out of consideration all symmetric branches of the search tree except for one
  - For example, suppose we have a dozen blocks on a table, and the goal is to have block A on top of a three-block tower
    - The first step in a solution is to place some block x on top of block y (where x, y, and A are all different)
    - then, place A on top of x and we're done
      - There are 11 choices for x, and given x, 10 choices for y, and thus 110 states to consider
        - But all these states are symmetric: choosing one over another makes no difference, and thus a planner should only consider one of them

Heuristics for Planning

- State abstraction in planning
  - state abstraction: a many-to-one mapping from states in the ground representation of the problem to the abstract representation
  - relaxations that decrease the number of states
    - The easiest form of state abstraction is to ignore some fluents
  - decomposition: dividing a problem into parts, solving each part independently, and then combining the parts
  - Subgoal independence assumption: the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal independently
  - The subgoal independence assumption can be optimistic or pessimistic
    - optimistic: negative interactions between the subplans for each subgoal
      - That is, when an action in one subplan deletes a goal achieved by another subplan
    - pessimistic: inadmissible, when subplans contain redundant actions
      - That is, two actions that could be replaced by single action in the merged plan