CMU 15-781
Lecture 6: Planning II

Teacher:
Gianni A. Di Caro
Recap: Classical Planning

• **Factored representation:** A state of the world is represented by a *collection of variables* → Exploit structure, sub-goaling / divide-and-conquer, domain-independent heuristics

• **PDDL / STRIPS:** Language expressive enough to describe a wide variety of problems, but restrictive enough to allow efficient algorithms to operate over it

• **State:** Conjunction of *literals*
**Recap: Classical Planning**

- **State:** Conjunction of *literals*
  - *Propositional literals:* Poor $\land$ Unknown
  - *Ground first order literals:* $\text{At(Plane}_1, \text{Rome}) \land \text{At(Plane}_2, \text{Tokyo})$
    $\text{At}(x, \text{Rome}) \land \text{At}(y, \text{Tokyo})$
  - *Function-free:* $\text{At(\text{Father(Tom)}, \text{NY})}$
    $\rightarrow \text{At(\text{Alex}, \text{NY})} \land \text{Father(\text{Alex, Tom})}$
  - *Closed-world assumption:* Any condition which is not mentioned in the state is assumed to be *false*

The world is represented through a set of features/objects (e.g., planes, people, cities) and each literal states a *fact* that attributes “values” to features
**Recap: Classical Planning**

- **Goals:** A conjunction of literals, $\text{At}(P_1, \text{JFK}) \land \text{At}(P_2, \text{SFO})$, that may also contain variables, such as $\text{At}(p, \text{JFK}) \land \text{Plane}(p)$, meaning that the goal is to have any plane at JFK.

- The aim is to reach a state that *entails* a goal: $\text{OnTable}(A) \land \text{OnTable}(B) \land \text{OnTable}(D) \land \text{On}(C, D) \land \text{Clear}(A) \land \text{Clear}(B) \land \text{Clear}(C)$ satisfies the goal to stack C on D.

- → A goal $g$ is a conjunction of *sub-goals*!
  $$g = g_1 \land g_2 \land \ldots \land g_n$$

- Goals are reached through *sequence of actions* (the plan).
Recap: Classical Planning

• **Actions**: Preconditions + Effects (Postconditions)
• **Action schema**: a number of different actions that can be derived by universal quantification of the variables, e.g., an action schema to fly a plane from one location to another:

  \[
  \text{Action}(\text{Fly}(p, \text{from}, \text{to}),
  \]

  \[
  \text{PRECOND: } \text{At}(p, \text{from}) \land \text{Plane}(p) \land \text{Airport(from)} \land \text{Airport(to)}
  \]

  \[
  \text{EFFECT: } \neg \text{At}(p, \text{from}) \land \text{At}(p, \text{to})
  \]

• An action is applicable in state \( s \) if \( s \) entails the preconditions
• The literals negated by the effect of \( a \) are removed from \( s \), while the positive literals resulting from \( a \) are added to \( s \)
Recap: Classical Planning

- \( \text{RESULT}(s, a) = (s - \text{DELETE}(a)) \cup \text{ADD}(a) \)

- Action schema:

\[
\text{Action}(\text{Name}(p_1, p_2, \ldots, p_n),
\]

\[
\text{PRECONDITIONS: } L_1(p) \land L_2(p) \land \ldots \land L_m(p)
\]

\[
\text{ADD-LIST: } \{A_1(p), A_2(p), \ldots, A_q(p)\}
\]

\[
\text{DELETE-LIST: } \{L_i(p), L_j(p) \land \ldots \land L_k(p)\}
\]
ReCap: Classical Planning

• **Planning domain**: Set of Action schemas (+ Set of Predicates)

• **Planning problem (instance)**: Planning domain + Initial state + Goal + Set of Objects (world features)

• **Solution of the planning problem**: A sequence of actions that, starting from the initial state, end in a state \( s \) that entails the goal

Air cargo transportation problem (from R&N)

• **Predicates**: At, Cargo, Plane, Airport, In

• **Objects**: \( C_1, C_2, P_1, P_2, SFO, JFK \)

• **Actions**: Load, Unload, Fly

\[
\begin{align*}
\text{Init} &: \text{At}(C_1, SFO) \land \text{At}(C_2, JFK) \land \text{At}(P_1, SFO) \land \text{At}(P_2, JFK) \\
& \land \text{Cargo}(C_1) \land \text{Cargo}(C_2) \land \text{Plane}(P_1) \land \text{Plane}(P_2) \\
& \land \text{Airport}(JFK) \land \text{Airport}(SFO)) \\
\text{Goal} &: \text{At}(C_1, JFK) \land \text{At}(C_2, SFO)) \\
\text{Action} &: \text{Load}(c, p, a), \\
& \quad \text{PRECOND: } \text{At}(c, a) \land \text{At}(p, a) \land \text{Cargo}(c) \land \text{Plane}(p) \land \text{Airport}(a) \\
& \quad \text{EFFECT: } \lnot \text{At}(c, a) \land \text{In}(c, p)) \\
\text{Action} &: \text{Unload}(c, p, a), \\
& \quad \text{PRECOND: } \text{In}(c, p) \land \text{At}(p, a) \land \text{Cargo}(c) \land \text{Plane}(p) \land \text{Airport}(a) \\
& \quad \text{EFFECT: } \text{At}(c, a) \land \lnot \text{In}(c, p)) \\
\text{Action} &: \text{Fly}(p, \text{from}, \text{to}), \\
& \quad \text{PRECOND: } \text{At}(p, \text{from}) \land \text{Plane}(p) \land \text{Airport}(\text{from}) \land \text{Airport}(\text{to}) \\
& \quad \text{EFFECT: } \lnot \text{At}(p, \text{from}) \land \text{At}(p, \text{to}))
\end{align*}
\]
PLANNING AS SEARCH

• (Forward) Search from initial state to goal
• Can use *standard search techniques*, including heuristic search

\[
\text{At}(P_1, A) \quad \text{At}(P_2, A) \\
\text{At}(P_1, B) \quad \text{At}(P_2, A) \\
\text{At}(P_1, A) \quad \text{At}(P_2, B) \\
\text{Fly}(P_1, A, B) \quad \text{Fly}(P_2, A, B)
\]
(FORWARD) STATE-SPACE SEARCH

• In absence of function symbols, the state space of a planning problem is finite → Any graph search algorithm that is complete will be a complete planning algorithm

• Irrelevant action problem: All applicable actions are considered at each state!

• The resulting branching factor $b$ is typically large and the state space is exponential in $b$ → Needs for good heuristics!

At home →
get milk, bananas and a cordless drill → return home
(Forward) State-space search

- Air Cargo Example
- Initial state: 10 airports, each airport has 5 planes and 20 pieces of cargo
- Goal: transport all the cargos at airport A to airport B
- Solution: load the 20 pieces of cargo at A into one of the planes at A and fly it to B
- Avg Branching factor $b$: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded), or loaded into any plane at its airport (if it is unloaded)
- Number of states to explore: $O(b^d) \sim 2000^{41}$
FIND A HEURISTIC: RELAX THE PROBLEM

• Define a Relaxed problem:
  o (Potentially) Easy to solve
  o The solution gives admissible heuristics for A*

• Relaxation: Remove all preconditions from actions

• → Every action will always be applicable, and any literal (sub-goal) can be achieved in one step

• → Adding edges to the graph: including forbidden actions

• → $h(x) =$ The number of steps required to get to the goal is the number of unsatisfied goals from current state $x$?
Domain-Independent Heuristic

- \( h(x) = \) The number of steps required to solve a conjunction of goals is the number of unsatisfied goals from current state \( x \)?

- Impossible to derive such a heuristic with atomic states! The successor function is a black box, here we exploit the structure of the representation.

- The heuristic is domain-independent!

- With atomic states, in general only domain-specific heuristics are possible.
HEURISTIC: IGNORE PRECONDITIONS

• Complications, that could made the heuristic function $h(x)$ not admissible:
  a. Some operations achieve multiple goals
  b. Some operations undo the effects of others

• Poll 1: To get an admissible heuristic, ignore preconditions and, in addition ignore:
  1. Just a
  2. Just b
  3. Both a and b
**IGNORE PRECONDITIONS & NON-GOAL EFFECTS**

- To avoid b. remove all the effects of actions, except those that are literals \( g_i, \ i=1, \ldots, n, \) in the goal \( g \) (i.e., sub-goals) → Exploit factored structure

- \( h(x) = \) the min number of actions such that the union of their effects contains all \( n \) sub-goals \( g_i \) → Admissible

- Computing \( h(x) = \) solving a SET COVER problem: NP-hard!

- Greedy log \( n \) approximation:
  - Admissibility is lost!
**Ignore (Specific) Preconditions**

- Ignore specific preconditions to derive *domain-specific* heuristics.
- Sliding block puzzle, move\((t,s_1,s_2)\) action:
  \[\text{On}(t,s_1) \land \text{Blank}(s_2) \land \text{Adjacent}(s_1,s_2) \Rightarrow \text{On}(t,s_2) \land \text{Blank}(s_1) \land \neg \text{On}(t,s_1) \land \neg \text{Blank}(s_2)\]
- Consider two options for removing specific preconditions from move():
  a. Removing \(\text{Blank}(s_2) \land \text{Adjacent}(s_1,s_2)\)
  b. Removing \(\text{Blank}(s_2)\)

- **Poll 2:** Match option to heuristic:
  1. a\(\leftrightarrow\)+\(\sum\)Manhattan, b\(\leftrightarrow\)#misplaced tiles
  2. a\(\leftrightarrow\)#misplaced tiles, b\(\leftrightarrow\)+\(\sum\)Manhattan
  3. b\(\leftrightarrow\)#misplaced tiles, a is inadmissible
  4. b\(\leftrightarrow\)+\(\sum\)Manhattan, a is inadmissible

---

**Example state**

```
5  2
6  1  3
7  8  4
```

**Goal state**

```
1  2  3
4  5  6
7  8
```
**Backward State-space Search**

- Searching from a goal state to the initial state (*regression*)
- We only need to consider actions that are relevant to the goal (or current state) → **Relevant-state search**
- This can make a strong reduction in branching factor, such that it could be more efficient than forward (progression) search
- “Imagine trying to figure out how to get to some small place with few traffic connections from somewhere with a lot of traffic connections”
BACKWARD STATE-SPACE SEARCH

• Regression from a (goal) state $g$ over the action $a$ gives state $g'$
  \[ g' = (g - \text{ADD}(a)) \cup \text{Preconditions}(a) \]

• DEL($a$) doesn’t appear: we don’t know whether the literals negated by DEL($a$) were true or not before $a$, therefore nothing can be said about them

• Variables can be included, such that a set of states is defined:
  \[ \text{Goal At(C}_2, \text{SFO}) \rightarrow \text{Unload(C}_2, p, \text{SFO}) \rightarrow g' = \text{In(C}_2,p) \land \text{At(p, SFO)} \land \text{Cargo(C}_2) \land \text{Plane(p)} \land \text{Airport(SFO)} \]
BACKWARD STATE-SPACE SEARCH

• How to select actions?

• Relevant actions only
  • Have an effect which is in the set of (current) goal literals
    Goal: $\text{At}(C_1, \text{JFK}) \land \text{At}(C_2, \text{SFO}) \rightarrow \text{Unload}(C_2, p, \text{SFO})$ is relevant, $\text{Fly}(p, \text{JFK}, \text{SFO})$ is not relevant

• Consistent actions only
  • Have no effect which negates an element of the goal
    Goal: $A \land B \land C$, action $a$ with effect $A \land B \land \neg C$ is not relevant
PLANNING GRAPHS

• Graph-based data structure representing a polynomial-size/time approximation of the exponential search tree

• Can be used to automatically produce good heuristic estimates (e.g., for A*)

• Can be used to search for a solution using the GRAPHPLAN algorithm
PLANNING GRAPHS

• Leveled graph: vertices organized into levels/stages, with edges only between levels

• Two types of vertices on alternating levels:
  o Conditions
  o Operations

• Two types of edges:
  o Precondition: from condition to operation
  o Postcondition: from operation to condition
GENERIC PLANNING GRAPH

* Slide based on Brafman
PLANNING GRAPH CONSTRUCTION

• $S_0$ contains all the conditions that hold in initial state
• Add operation to level $O_i$ if its preconditions appear in level $S_i$
• Add condition to level $S_i$ if it is the effect of an operation in level $O_{i-1}$ (*no-op action* also possible)
• **Idea:** $S_i$ contains all conditions that *could* hold at stage $i$; $O_i$ contains all operations that *could* have their preconditions satisfied at time $i$
• Can *optimistically estimate* how many steps it takes to reach a goal: it includes all possible operations and preconditions that could hold, multiple actions could be executed (in parallel) at each stage (time step)
Mutual exclusion links

• The graph also records conflicts between actions or conditions: two operations or conditions are mutually exclusive (mutex) if no valid plan can contain both at the same time.

• A bit more formally:
  o Two operations are mutex if their preconditions or postconditions are mutex.
  o Two conditions are mutex if one is the negation of the other, or all actions that achieve them are mutex.

• Even more formally...
A RUNNING EXAMPLE

- “Have cake and eat cake too” problem

**Initial state:** \( \text{Have}(\text{Cake}) \)

**Goal:** \( \text{Have}(\text{Cake}) \land \text{Eaten}(\text{Cake}) \)

**Eat(Cake):**
- **Precond:** \( \text{Have}(\text{Cake}) \)
- **Effect:** \( \neg \text{Have}(\text{Cake}) \land \text{Eaten}(\text{Cake}) \)

**Bake(Cake):**
- **Precond:** \( \neg \text{Have}(\text{Cake}) \)
- **Effect:** \( \text{Have}(\text{Cake}) \)
A RUNNING EXAMPLE

Initial state: $\text{Have}(\text{Cake})$

Goal: $\text{Have}(\text{Cake}) \land \text{Eaten}(\text{Cake})$

$\text{Eat}(\text{Cake})$:
- **Precond:** $\text{Have}(\text{Cake})$
- **Effect:** $\neg\text{Have}(\text{Cake}) \land \text{Eaten}(\text{Cake})$

$\text{Bake}(\text{Cake})$:
- **Precond:** $\neg\text{Have}(\text{Cake})$
- **Effect:** $\text{Have}(\text{Cake})$
A running example

Initial state: $Have(Cake)$
Goal: $Have(Cake) \land Eaten(Cake)$

$Eat(Cake)$:
- **Precond:** $Have(Cake)$
- **Effect:** $\neg Have(Cake) \land Eaten(Cake)$

$Bake(Cake)$:
- **Precond:** $\neg Have(Cake)$
- **Effect:** $Have(Cake)$
**Mutex Cases**

- **Inconsistent postconditions** (two ops): one operation negates the effect of the other, *Eat(Cake)* and no-op *Have(Cake)*

- **Interference** (two ops): a postcondition of one operation negates a precondition of other, *Eat(Cake)* and no-op *Have(Cake)* (issue in parallel execution, the order should not matter but here it would)

* Slide based on Brafman
**Mutex Cases**

- **Competing needs** (two ops): a precondition of one operation is mutex with a precondition of the other, *Bake(Cake)* and *Eat(Cake)*

- **Inconsistent support** (two conditions): each possible pair of operations that achieve the two conditions is mutex, *Have(Cake)* and *Eaten(Cake)*, are mutex in $S_1$ but not in $S_2$ because they can be achieved by *Bake(Cake)* and *Eaten(Cake)*

* Slide based on Brafman
A Running Example

Initial state: $\text{Have(Cake)}$

Goal: $\text{Have(Cake)} \land \text{Eaten(Cake)}$

$\text{Eat(Cake)}$:
- Precond: $\text{Have(Cake)}$
- Effect: $\neg\text{Have(Cake)} \land \text{Eaten(Cake)}$

$\text{Bake(Cake)}$:
- Precond: $\neg\text{Have(Cake)}$
- Effect: $\text{Have(Cake)}$

Inconsistent postconditions

Negation of each other

Interference
A running example

Initial state: Have(Cake)

Goal: Have(Cake) \land Eaten(Cake)

Eat(Cake):
  PRECOND: Have(Cake)
  EFFECT: \neg Have(Cake) \land Eaten(Cake)

Bake(Cake):
  PRECOND: \neg Have(Cake)
  EFFECT: Have(Cake)

Inconsistent support

Competing needs
Planning Graphs

To be continued ...