Planning

___

KRR for Agents in Dynamic Environments
Overview

- Last week discussed **Decision Making** in some very general settings: Markov process, MDP, HMM, POMDP.

- This week look at a practical application of these ideas in a more restricted setting.

- **Planning** (or **AI Planning**) is about agents that execute actions to reach goals (e.g., a robot delivering an item).

- Note: ties closely to **Reasoning about Actions** (Week 9).
Some Dictionary Definitions of “Plan”

plan n.

1. A scheme, program, or method worked out beforehand for the accomplishment of an objective: a plan of attack.

2. A proposed or tentative project or course of action: had no plans for the evening.

[a representation] of future behaviour … usually a set of actions, with temporal and other constraints on them, for execution by some agent or agents.

– Austin Tate, MIT Encyclopaedia of the Cognitive Sciences, 1999
Classical Planning

- Deterministic environment; complete information
- Representations for classical planning
- Solving planning problems
  - Modern heuristics for state-space planning
  - Answer Set Programming and Graphplan

Background reading

*Automated Planning*, Malik Ghallab, Dana Nau, Paolo Traverso, Morgan Kaufmann 2004. Chapters 1, 2, 4 & 6


Note: I think Chapter 10 for 3rd Edition of Russell and Norvig.
Planning Overview

- Dynamic environment.
- One or more agents: (virtual) agents, robots.
- Agents take actions that change the environment.
- Agents have goals that they want to achieve.
- What sequence of actions will allow the agent to achieve its goals?
- Blocksworld is a prototypical example of a classical planning problem.
Planning for an Agent/Robot in a Dynamic World

\[ \Sigma = (S, A, \gamma) \]
- \( S = \{ \text{states} \} \)
- \( A = \{ \text{actions} \} \)
- \( \gamma = \text{state-transition function} \)

\( \Sigma \) is an abstraction that deals only with the aspects that the planner needs to reason about.
Example

Example $\Sigma = (S,A,\gamma)$:

- $S = \{s_0, \ldots, s_5\}$
- $A = \{\text{move1, move2, putdown, pickup, load, unload}\}$
- $\gamma$: see the arrows

Dock Worker Robots (DWR) example
Example

- **Classical plan**: a sequence of actions
  \[
  \langle \text{pickup, move1, load, move2} \rangle
  \]
Domain-Specific Planners

- Many successful real-world planning systems work this way
  - Mars exploration, sheet-metal bending, playing bridge, etc.
- Often use problem-specific techniques that are difficult to generalise to other planning domains
- For example, encodes the knowledge of domain experts (e.g., computer poker player)
Domain-Independent Planners

- No domain-specific knowledge except the description of the system $\Sigma$
- In practice,
  - Not feasible to make domain-independent planners work well in all possible planning domains
- Make simplifying assumptions to restrict the set of domains
  - **Classical planning**
    - Historical focus of most research on automated planning
Reduces to the following problem:
Given $\Sigma$, initial state $s_0$, and goal states $S_g$,
find a sequence of actions $(a_1, a_2, \ldots a_n)$ that produces
a sequence of state transitions $(s_0, s_1, s_2, \ldots, s_n)$ such that $s_n \in S_g$

Is this trivial?

- Generalise the earlier example:
  - Five locations, three robot carts, 100 containers, three piles
    - $10^{277}$ states

- Automated-planning research has been heavily dominated by classical planning. There are dozens of different algorithms.
Representations for Classical Planning
Classical Representations: Motivation

- In most problems, far too many states to try to represent all of them explicitly as $s_0, s_1, s_2, \ldots$
  - Represent each state as a set of **atomic features**
    - Example feature, $light(on)$ or $light(off)$; the light can be on or off.

- Define a set of **operators** that can be used to compute state-transitions
  - Example operator, $switch(on)$; turn the switch on.

- Don’t give all of the states explicitly
  - Just give the initial state
  - Use the operators to generate the other states as needed
Classical Representation

- Language of first-order logic but without function symbols
  - finitely many predicate symbols and constant symbols
- Classical planning problems often described using the STRIPS action language (developed in 1970s), or PDDL (a more modern language).
- We use STRIPS syntax, but for our purposes can think of STRIPS and PDDL as being used to represent the same sorts of problems.

Example: the DWR domain
- Locations: l1, l2, ...
- Containers: c1, c2, ...
- Piles: p1, p2, ...
- Robot carts: r1, r2, ...
- Cranes: crane1, crane2, ...
Example (cont'd)

- **Fixed (static) relations**: same in all states
  - adjacent($l,l'$)  
  - attached($p,l$)  
  - belong($k,l$)

- **Dynamic relations (fluents)**: differ between states
  - occupied($l$)  
  - at($r,l$)
  - loaded($r,c$)  
  - unloaded($r$)
  - holding($k,c$)  
  - empty($k$)
  - in($c,p$)  
  - on($c,c'$)
  - top($c,p$)  
  - top(pallet,$p$)

- **Actions**:
  - pickup($c,k,p$)  
  - putdown($c,k,p$)
  - load($r,c,k$)  
  - unload($r$)  
  - move($r,l,l'$)
States

A state is a set $s$ of ground atoms

- The atoms represent the things that can be true in some states
- Only finitely many ground atoms, so only finitely many possible states

$$s_1 = \{\text{attached}(p1,\text{loc1}), \text{in}(c1,p1), \text{in}(c3,p1), \text{top}(c3,p1), \text{on}(c3,c1), \text{on}(c1,\text{pallet}), \text{attached}(p2,\text{loc1}), \text{in}(c2,p2), \text{top}(c2,p2), \text{on}(c2,\text{pallet}), \text{belong}(\text{crane1,loc1}), \text{empty}(\text{crane1}), \text{adjacent}(\text{loc1,loc2}), \text{adjacent}(\text{loc2,loc1}), \text{at}(r1,\text{loc2}), \text{occupied}(\text{loc2}), \text{unloaded}(r1)\}$$
Operators

An operator is a triple $o = (\text{name}(o), \text{precond}(o), \text{effects}(o))$

- **name**($o$): a syntactic expression of the form $n(x_1,\ldots,x_k)$
  - $(x_1,\ldots,x_k)$ is a list of every variable symbol (parameter) that appears in $o$
- **precond**($o$): **preconditions**
  - literals that must be true in order to use the operator
- **effects**($o$): **effects**
  - literals the operator will make true

**Example**

```
pickup(k,l,c,d,p)
;; crane k at location l takes c off of d in pile p
precond: belong(k,l), attached(p,l), empty(k), top(c,p), on(c,d)
effects: holding(k,c), ¬empty(k), ¬in(c,p), ¬top(c,p), ¬on(c,d), top(d,p)
```
Actions

An **action** is a ground instance (via a substitution) of an operator

<table>
<thead>
<tr>
<th>pickup(k,l,c,d,p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>;; crane k at location l takes c off of d in pile p</td>
</tr>
<tr>
<td>precond: belong(k,l), attached(p,l), empty(k), top(c,p), on(c,d)</td>
</tr>
<tr>
<td>effects: holding(k,c), ¬empty(k), ¬in(c,p), ¬top(c,p), ¬on(c,d), top(d,p)</td>
</tr>
</tbody>
</table>

- Let $\sigma = \{k/crane1, l/loc1, c/c3, d/c1, p/p1\}$
- Then $\text{pickup}(k,l,c,d,p)\sigma$ is the following action:

<table>
<thead>
<tr>
<th>pickup(crane1,loc1,c3,c1,p1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>precond: belong(crane1,loc1), attached(p1,loc1), empty(crane1), top(c3,p1), on(c3,c1)</td>
</tr>
<tr>
<td>effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1), ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)</td>
</tr>
</tbody>
</table>
Applicability and Result of Actions

- Let $S$ be a set of literals. Then
  \[
  S^+ = \{\text{atoms that appear positively in } S\} \\
  S^- = \{\text{atoms that appear negatively in } S\}
  \]

- Let $a$ be an operator or action. Then
  \[
  \text{precond}^+(a) = \{\text{atoms that appear positively in } a\text{'s preconditions}\} \\
  \text{precond}^-(a) = \{\text{atoms that appear negatively in } a\text{'s preconditions}\} \\
  \text{effects}^+(a) = \{\text{atoms that appear positively in } a\text{'s effects}\} \\
  \text{effects}^-(a) = \{\text{atoms that appear negatively in } a\text{'s effects}\}
  \]

- Action $a$ is **applicable** to (or **executable** in) $S$ if
  \[
  \text{precond}^+(a) \subseteq S \\
  \text{precond}^-(a) \cap S = \emptyset
  \]

- The **result** of applying action $a$ to state $S$ is
  \[
  \gamma(s,a) = (S \setminus \text{effects}^-(a)) \cup \text{effects}^+(a)
  \]
Example: Applicability

- An action:
  \[\text{pickup}(\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1})\]
  - precond: \[\text{belong}(\text{crane}, \text{loc1}), \text{attached}(\text{p1}, \text{loc1}), \text{empty}(\text{crane1}), \text{top}(\text{c3}, \text{p1}), \text{on}(\text{c3}, \text{c1})\]
  - effects: \[\text{holding}(\text{crane1}, \text{c3}), \neg \text{empty}(\text{crane1}), \neg \text{in}(\text{c3}, \text{p1}), \neg \text{top}(\text{c3}, \text{p1}), \neg \text{on}(\text{c3}, \text{c1}), \text{top}(\text{c1}, \text{p1})\]

- A state it’s applicable to
  \[s_1 = \{\text{attached}(\text{p1}, \text{loc1}), \text{in}(\text{c1}, \text{p1}), \text{in}(\text{c3}, \text{p1}), \text{top}(\text{c3}, \text{p1}), \text{on}(\text{c3}, \text{c1}), \text{on}(\text{c1}, \text{pallet}), \text{attached}(\text{p2}, \text{loc1}), \text{in}(\text{c2}, \text{p2}), \text{top}(\text{c2}, \text{p2}), \text{on}(\text{c2}, \text{pallet}), \text{belong}(\text{crane1}, \text{loc1}), \text{empty}(\text{crane1}), \text{adjacent}(\text{loc1}, \text{loc2}), \text{adjacent}(\text{loc2}, \text{loc1}), \text{at}(\text{r1}, \text{loc2}), \text{occupied}(\text{loc2}, \text{unloaded}(\text{r1}))\}\]
Example: Result

\[
\text{pickup}(\text{crane1}, \text{loc1}, \text{c3}, \text{c1}, \text{p1})
\]

\text{precond:} \quad \begin{align*}
&\text{belong}(\text{crane}, \text{loc1}), \\
&\text{attached}(\text{p1}, \text{loc1}), \\
&\text{empty}(\text{crane1}), \text{top}(\text{c3}, \text{p1}), \\
&\text{on}(\text{c3}, \text{c1})
\end{align*}

\text{effects:} \quad \begin{align*}
&\text{holding}(\text{crane1}, \text{c3}), \\
&\neg \text{empty}(\text{crane1}), \\
&\neg \text{in}(\text{c3}, \text{p1}), \neg \text{top}(\text{c3}, \text{p1}), \\
&\neg \text{on}(\text{c3}, \text{c1}), \text{top}(\text{c1}, \text{p1})
\end{align*}

\[s_2 = \{\text{attached}(\text{p1}, \text{loc1}), \text{in}(\text{c1}, \text{p1}), \text{in}(\text{c3}, \text{p1}), \text{top}(\text{c3}, \text{p1}), \text{on}(\text{c3}, \text{c1}), \text{on}(\text{c1}, \text{pallet}), \text{attached}(\text{p2}, \text{loc1}), \text{in}(\text{c2}, \text{p2}), \text{top}(\text{c2}, \text{p2}), \text{on}(\text{c2}, \text{pallet}), \text{belong}(\text{crane1}, \text{loc1}), \text{empty}(\text{crane1}), \text{adjacent}(\text{loc1}, \text{loc2}), \text{adjacent}(\text{loc2}, \text{loc1}), \text{at}(\text{r1}, \text{loc2}), \text{occupied}(\text{loc2}, \text{unloaded}(\text{r1}), \text{holding}(\text{crane1}, \text{c3}), \text{top}(\text{c1}, \text{p1})\}
Exercise
Exercise: The Blocks World

- Infinitely wide table, finite number of children’s blocks
- Ignore where a block is located on the table
- A block can sit on the table or on another block
- There’s a robot gripper that can hold at most one block

Want to move blocks from one configuration to another

 e.g.,

 initial state  goal

```
<table>
<thead>
<tr>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>
```

```
| a |
| b |
| c |
```
Exercise: Classical Representation – Symbols

- Constant symbols:
  - The blocks: a, b, c, d, e
- Dynamic relations?
Exercise: Classical Operators

- Preconditions and effects?
Given a planning domain (language $L$, operators $O$)

- **Representation** of a planning problem: a triple $P = (O, s_0, g)$
  - $O$ is the collection of operators
  - $s_0$ is a state (the initial state)
  - $g$ is a set of literals (the goal formula)
Plans and Solutions

Let $P = (O, s_0, g)$ be a planning problem

- **Plan**: any sequence of actions $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ such that each $a_i$ is an instance of an operator in $O$
- Plan $\pi$ is a **solution** for $P = (O, s_0, g)$ if it is executable and achieves $g$

  i.e., if there are states $s_0, s_1, \ldots, s_n$ such that

  $\gamma(s_0, a_1) = s_1$

  $\gamma(s_1, a_2) = s_2$

  $\vdots$

  $\gamma(s_{n-1}, a_n) = s_n$

  $s_n$ satisfies $g$
Example: The 5 DWR Operators

\textit{move}(r,l,m)

\begin{itemize}
  \item \textit{precond}: \text{adjacent}(l,m), \text{at}(r,l), \neg\text{occupied}(m)
  \item \textit{effects}: \text{at}(r,m), \text{occupied}(m), \neg\text{occupied}(l), \neg\text{at}(r,l)
\end{itemize}

\textit{load}(k,l,c,r)

\begin{itemize}
  \item \textit{precond}: \text{belong}(k,l), \text{holding}(k,c), \text{at}(r,l), \text{unloaded}(r)
  \item \textit{effects}: \text{empty}(k), \neg\text{holding}(k,c), \text{loaded}(r,c), \neg\text{unloaded}(r)
\end{itemize}

\textit{unload}(k,l,c,r)

\begin{itemize}
  \item \textit{precond}: \text{belong}(k,l), \text{at}(k,l), \text{loaded}(r,c), \text{empty}(k)
  \item \textit{effects}: \neg\text{empty}(k), \text{holding}(k,c), \text{unloaded}(r), \neg\text{loaded}(r,c)
\end{itemize}

\textit{putdown}(k,l,c,d,p)

\begin{itemize}
  \item \textit{precond}: \text{belong}(k,l), \text{attached}(p,l), \text{holding}(k,c), \text{top}(d,p)
  \item \textit{effects}: \neg\text{holding}(k,c), \text{empty}(k), \text{in}(c,p), \text{top}(c,p), \text{on}(c,d), \neg\text{top}(d,p)
\end{itemize}

\textit{pickup}(k,l,c,d,p)

\begin{itemize}
  \item \textit{precond}: \text{belong}(k,l), \text{attached}(p,l), \text{empty}(k), \text{top}(c,p), \text{on}(c,d)
  \item \textit{effects}: \text{holding}(k,c), \neg\text{empty}(k), \neg\text{in}(c,p), \neg\text{top}(c,p), \neg\text{on}(c,d), \text{top}(d,p)
\end{itemize}
Example

- Let $P = (O, s_0, g)$, where
  - $O = \{\text{the 5 DWR operators}\}$
  - $s_0 = \{\text{attached}(p1,\text{loc1}), \text{in}(c1,p1), \text{in}(c3,p1), \text{top}(c3,p1), \text{on}(c3,c1), \text{on}(c1,\text{pallet}), \text{attached}(p2,\text{loc1}), \text{in}(c2,p2), \text{top}(c2,p2), \text{on}(c2,\text{pallet}), \text{belong}(\text{crane1,loc1}), \text{empty}(\text{crane1}), \text{adjacent}(\text{loc1,loc2}), \text{adjacent}(\text{loc2,loc1}), \text{at}(\text{r1,loc2}), \text{occupied}(\text{loc2}), \text{unloaded}(\text{r1})\}$
  - $g = \{\text{loaded}(\text{r1,c3}), \text{at}(\text{r1,loc2})\}$
Two *redundant* solutions (can remove actions and still have a solution):

( move(r1,loc2,loc1),
  pickup(crane1,loc1,c3,c1,p1),
  move(r1,loc1,loc2),
  move(r1,loc2,loc1),
  load(crane1,loc1,c3,r1),
  move(r1,loc1,loc2) )

( pickup(crane1,loc1,c3,c1,p1),
  putdown(crane1,loc1,c3,c2,p2),
  move(r1,loc2,loc1),
  pickup(crane1, loc1,c3,c2,p2),
  load(crane1,loc1,c3,r1),
  move(r1,loc1,loc2) )

A solution that is both *irredundant* and *shortest*:

( move(r1,loc2,loc1), pickup(crane1, loc1,c3,c1,p1),
  load(crane1,loc1,c3,r1), move(r1,loc1,loc2) )

Are there any other shortest solutions? Are irredundant solutions always the shortest?
Exercise
Exercise: Plans

initial state

Solution?
Alternative to the classical representation
- Use ground atoms for properties that do not change, e.g., adjacent(loc1,loc2)
- For properties that can change, assign values to **state variables**
  - Like fields in a record structure

```
move(r, l, m)
;; robot r at location l moves to an adjacent location m
precond: rloc(r) = l, adjacent(l, m)
effects: rloc(r) ← m
```

```
s_1 = {top(p1)=c3, cpos(c3)=c1, cpos(c1)=pallet, holding(crane1)=nil, rloc(r1)=loc2, loaded(r1)=nil, ...}
```
Expressive Power

- Any problem that can be represented in one representation can also be represented in the other
- Can convert in linear time and space

\[
P(x_1, \ldots, x_n) \quad \text{becomes} \quad f_p(x_1, \ldots, x_n) = 1
\]

\[
f(x_1, \ldots, x_n) = y \quad \text{becomes} \quad P_f(x_1, \ldots, x_n, y)
\]

Classical representation \quad State-variable representation
Comparison

- Classical representation
  - The most popular for classical planning, partly for historical reasons

- State-variable representation
  - Equivalent to classical representation in expressive power
  - Less natural for logicians, more natural for engineers and most computer scientists
  - Useful in non-classical planning problems as a way to handle numbers, functions, time
State-Space Search
Search Algorithms

**Search tree**
- nodes = states
- edges = actions
Search Algorithms

Search tree
- nodes = states
- edges = actions

Most common search method: **depth-first** search
- In general, sound but not complete
  - But classical planning has only finitely many states
  - can make depth-first search complete by doing loop-checking
Exercise
Exercise: Interchange Values of Variables

**Operator** assign(v,w,x,y)

;; assign the value of v (which is currently x)

;; to the value of w (which is currently y)

precond: value(v,x), value(w,y)

effects: ¬value(v,x), value(v,y)

**Initial state**  \( s_0 = \{ \text{value(a,3)}, \text{value(b,5)}, \text{value(c,0)} \} \)

**Goal**  \( g = \{ \text{value(a,5)}, \text{value(b,3)} \} \)

In the search tree for this planning problem,

- what is the length of the shortest path to a solution?

- what is the length of the longest path in the tree?
Algorithms/Technologies for Solving Planning Problems
How to Solve Classical Planning Problems

- Heuristic search
  - Heuristics aren't guaranteed to work - but work well as a guide
  - Many different heuristics
  - Is it possible to generate heuristics automatically?

- Planning with Answer Set Programming
  - Requires setting a maximum length to the path

- Graphplan (won't cover here)
  - Algorithm developed in 1995
  - At the time it set a new benchmark for planning!
Motivation

- A standard tree search may try lots of actions that are unrelated to the goal.

- One way to reduce branching factor:
  - First create a **relaxed problem**
    - Remove some restrictions of the original problem
      - Want the relaxed problem to be easy to solve (polynomial time)
    - The solutions to the relaxed problem will include all solutions to the original problem
  - Then do a modified version of the original search
    - Restrict its search space to include only those actions that occur in solutions to the relaxed problem
Planning with Heuristic Search

- Explicitly search with heuristic $h(s)$ that estimates cost from $s$ to goal

- General idea:
  heuristic function = length of optimal plan for a relaxed problem

- Example:
  - Manhattan distance in 15-puzzle (sum of distances to correct positions)
  - Manhattan distance is an admissible heuristic (it never overestimates the cost).

- How to get such heuristics automatically?
Relaxation Heuristic - General-Purpose Heuristics for Classical Planning

- Automatic extraction of informative heuristic function from the problem P itself

- Most common relaxation in planning: ignore all negative effects of the operators.

Let $P^+$ be obtained from planning problem $P$ by dropping the negative effects. If $c^*(P^+, s)$ is optimal cost of $P^+$ with initial state $s$, then the heuristic is set to

$$h(s) = c^*(P^+, s)$$

- This heuristic is intractable in general, but easy to approximate

**Example.**

- **Operator** assign(v,w,x,y)
  - precond: value(v,x), value(w,y)
  - effects: $\neg$ value(v,x), value(v,y)

- $s_0 = \{ \text{value}(a,3), \text{value}(b,5), \text{value}(c,0) \}$, $g = \{ \text{value}(a,5), \text{value}(b,3) \}$

- Optimal relaxed plan: assign(a,b,3,5), assign(b,a,5,3), hence $h(s_0) = 2$
Planning with ASP – Simple Example

Cake-Example

• Two state properties: have, eaten
• Action eat, which is possible if have is true;
  effects: eaten, not have
• Action bake, which is possible if have is false;
  effect: have
• Initially, have is true
• The goal is to make eaten true

• Add to each state feature and action a time argument
  • p(T) – p is true at time T
  • a(T) – action a is taken at time T

• Initial state:
  have(0).
Planning with ASP – Preconditions & Effects

- Plan length ($\tau = \text{search depth}$):
  
  $$\text{time}(0..\tau).$$

- Generator: one action at a time
  
  $$1 \{ \text{bake}(T); \text{eat}(T) \} 1 :\text{-} \text{time}(T).$$

- Tester (1): Action preconditions
  
  $$:- \text{eat}(T), \text{not have}(T).$$  
  $$:- \text{bake}(T), \text{have}(T).$$

- Auxiliary rules: Action effects
  
  $$\text{have}(T+1) :\text{-} \text{bake}(T), \text{time}(T).$$  
  $$\text{have}(T+1) :\text{-} \text{have}(T), \text{not eat}(T), \text{time}(T).$$  
  $$\text{eaten}(T+1) :\text{-} \text{eat}(T), \text{time}(T).$$  
  $$\text{eaten}(T+1) :\text{-} \text{eaten}(T), \text{time}(T).$$

Stands for: \(\text{time}(0).\)  
\(\text{time}(1).\)  
\(\ldots\)  
\(\text{time}(\tau).\)  
where \(\tau\) a number $\geq 0$

eat possible if have true  
bake possible if have false
Condition under which have remains unchanged
Planning with ASP - Goal Conditions

• Tester (2):
  exclude models where the goal has \textit{not} been reached at time $\tau + 1$

% Goal
:- not eaten(\tau+1).

Remember:
the goal is to make eaten \texttt{true}
Planning with ASP - Plans

```
time(0..0). % equivalent to just "time(0)."
have(0).
1 { eat(T); bake(T) } 1 :- time(T).
:- eat(T), not have(T).
:- bake(T), have(T).
have(T+1) :- bake(T), time(T).
have(T+1) :- have(T), not eat(T), time(T).
eaten(T+1) :- eat(T), time(T).
eaten(T+1) :- eaten(T), time(T).
:- not eaten(1).
```

- Plans correspond to answer sets:
  - there is a stable model iff there is a valid sequence of n moves that leads to the goal
- A valid plan:
  - all action instances in the stable model. Here: eat(0)
Summary

- Representations for classical planning
  - Classical representation
  - State-variable representation

- State-space search
  - with heuristics

- Solving planning problems
  - With heuristics
  - ASP
  - Graphplan