The Planning Problem

Finding a sequence of actions to achieve some goal.
Plans

It’s great when a plan just works …

… but the world doesn’t work like that.

To plan effectively we must take uncertainty seriously.
Probabilistic Planning

As before:
- Generalize deterministic logic to probabilities.
- Generalize deterministic planning to probabilistic planning.

This results in a harder planning problem.

In particular:
- Must model stochasticity.
- Plans can fail.
- Can no longer compute straight-line plans.
Stochastic Outcomes

\[ s' = T(s, a) \]

\[ C(s, a, s') \]

\[ R(s, a, s') \]

probability distribution over transitions
Probabilistic Planning

Recall - systems that change over time:

- *Problem has a state.*
- State has the Markov property.

\[
P(S_t | S_{t-1}, a_{t-1}, S_{t-2}, a_{t-2}, \ldots, S_0, a_0) = P(S_t | S_{t-1}, a_{t-1})
\]

only the previous state

but also the previous action *(controlled process)*
The Markov Property

Needs to be extended for planning:
• \( s_{t+1} \) depends only on \( s_t \) and \( a_t \),
• \( r_t \) depends only on \( s_t, a_t \), and \( s_{t+1} \)

Current state is a sufficient statistic of agent’s history.

This means that:
• Decision-making depends only on current state
• The agent does not need to remember its history
Probabilistic Planning

**Markov Decision Processes (MDPs):**

- *The* canonical decision making formulation.
- Problem has a set of states.
- Agent has available actions.

- Actions cause stochastic *transitions*.
- Transitions have *costs/rewards*.
  - Transitions, costs depend *only on previous state*.

- Agent must choose actions to maximize reward (minimize costs) *summed over time*. 
Markov Decision Processes

\( S \) : set of states
\( A \) : set of actions
\( \gamma \) : discount factor

\( R \) : reward function
\[ R(s, a, s') \] is the reward received taking action \( a \) from state \( s \) and transitioning to state \( s' \).

\( T \) : transition function
\[ T(s'|s, a) \] is the probability of transitioning to state \( s' \) after taking action \( a \) in state \( s \).

(some states are absorbing - execution stops)
Episodic Problems

Some problems end when you hit a particular state.

Model: transition to absorbing state.
In practice: reset the problem.
MDPs

Goal: choose actions to **maximizes return**: expected sum of discounted rewards.

$$R^\pi(s) = \mathbb{E} \left[ \sum_{i=0}^{\infty} \gamma^i r_i \right]$$

(equiv: min sum of costs)

due to stochasticity  
all future rewards  
now matters more  
rewards summed
Example

**States:** set of grid locations

**Actions:** up, down, left, right

**Transition function:** move in direction of action with \( p = 0.9 \)

**Reward function:** -1 for every step, 1000 for (absorbing) goal
Back to PDDL

MDPs do not contain the structure of PDDL.

• *PPDDL*: probabilistic planning domain definition language

Now operators have probabilistic outcomes:

```
(:action move_left
  :parameters (x, y)
  :precondition (not (wall(x-1, y)))
  :effect (probabilistic
      0.8 (and (at(x-1)) (not at(x)) (decrease (reward) 1))
      0.2 (and (at(x+1)) (not(at(x))(decrease (reward) 1))
  )
)
```
Example

\[
\begin{align*}
A & \quad B & \quad C \\
0.8 & \quad r=-2 & \\
0.2 & \quad r=-5
\end{align*}
\]
Our goal is to find a policy:

$$\pi : S \rightarrow A$$

... that maximizes return: expected sum of rewards. (equiv: min sum of costs)

$$R^\pi (s) = \mathbb{E} \left[ \sum_{i=0}^{\infty} \gamma^i r_i \right]$$
Policies and Plans

Compare a policy:
• An action for every state.

… with a plan:
• A sequence of actions.

Why the difference?
Policies
Planning

So our goal is to produce optimal policy.

\[ \pi^*(s) = \max_{\pi} R^\pi(s) \]

Note: we know \( T \) and \( R \).

Useful fact: such a policy always exists. (But there might be more than one.)
Planning

The key quantity is the return given by a policy from a state:

\[ R^\pi(s) \]

Define the value function to estimate this quantity:

\[ V^\pi(s) = \mathbb{E} \left[ \sum_{i=0}^{\infty} \gamma^i r_i \right] \]
Value Functions

$V$ is a useful thing to know. Maybe we can use it to improve $\pi$. How to find $V$?
Monte Carlo

Simplest thing you can do: sample $R(s)$.

Do this repeatedly, average:

$$V^\pi (s) = \frac{R_1(s) + R_2(s) + \ldots + R_n(s)}{n}$$
Monte Carlo Estimation

One approach:

- For each state $s$
- Repeat many times:
  - Start at $s$
  - Run policy forward until absorbing state (or $\gamma^t < \epsilon$)
  - Write down discount sum of rewards received
  - This is a sample of $V(s)$
  - Average these samples

This always works!

*But very high variance.* Why?
Monte Carlo Estimation

\[ R = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \ldots + \gamma^n r_n \]
Doing Better

We need an estimate of R that doesn’t grow in variance as the episode length increases.

Might there be some relationship between values that we can use as an extra source of information?

\[
R(s_0) = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \ldots + \gamma^n r_n
\]

\[
R(s_1) = \gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3 + \ldots + \gamma^{n-1} r_n
\]
Bellman’s equation is a condition that must hold for $V$:

$$V^\pi(s) = \mathbb{E}_{s'} \left[ r(s, \pi(s), s') + \gamma V^\pi(s') \right]$$

- Value of this state
- Reward
- Value of next state
Dynamic Programming

We can use this expression to update $V$:

$$V^{\pi}(s) \leftarrow \sum_{s'} [T(s'|s, \pi(s)) \times (r(s, \pi(s), s') + \gamma V^{\pi}(s'))]$$

This algorithm is called **dynamic programming**
Value Iteration

This gives us an algorithm for **computing the value function for a specific given fixed policy**:

Repeat:

- Make a copy of the VF.
- For each state in VF, assign value using BE.
- Replace old VF.

This is known as **value iteration**.
Value Iteration

\[ V[s] = 0, \forall s \]

do:
\[
V_{old} = \text{copy}(V)
\]
for each state \( s \):

\[
V[s] = \sum_{s'} T(s, \pi(s), s') [r(s, \pi(s), s') + \gamma V_{old}[s']] 
\]

until \( V \) converges.

Notes:

- Fixed policy \( \pi \).
- \( V[s'] = 0 \), definitionally, if \( s \) is absorbing.
Policy Iteration

Recall that we seek the policy that maximizes $V^\pi(s), \forall s$.

Therefore we know that, for the optimal policy $\pi^*$:

$$V^{\pi^*}(s) \geq V^\pi(s), \forall \pi, s$$

This means that any change to $\pi$ that increases $V^\pi$ anywhere obtains a better policy.
Policy Iteration

This leads to a general policy improvement framework:
1. Start with a policy $\pi$
2. Estimate $V^\pi$
3. Improve $\pi$
   a. $\pi(s) = \max_a \mathbb{E} [r + \gamma V^\pi(s')]$, $\forall s$

This is known as **policy iteration**.
It is guaranteed to converge to the optimal policy.

Steps 2 and 3 can be interleaved as rapidly as you like.
Policy Iteration

\[ V[s] = 0, \forall s \]

do:
\[ V_{old} = \text{copy}(V) \]

for each state \( s \):
\[ V[s] = \sum_{s'} T(s, \pi(s), s') [r(s, \pi(s), s') + \gamma V_{old}[s']] \]

for each state \( s \):
\[ \pi(s) = \text{argmax}_a \sum_{s'} T(s, a, s') [r(s, a, s') + \gamma V[s']] \]

while \( \pi \) changes.

Finds an optimal policy in time polynomial in \(|S|\) and \(|A|\).
(There are \(|A|^{|S|}\) possible policies.)
Policy Iteration
Improvements

Extensions to the basic algorithm largely deal with controlling the size of the state sweeps:

- Not all states are reachable.
- Not all states need to be updated at each iteration.
- Not all states are likely to be encountered from a start state.

DP algorithms can solve problems with millions of states.
Elevator Scheduling

Crites and Barto (1985):

- System with 4 elevators, 10 floors.
- Realistic simulator.
- 46 dimensional state space.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>AvgWait</th>
<th>SquaredWait</th>
<th>SystemTime</th>
<th>Percent &gt;60 secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECTOR</td>
<td>30.3</td>
<td>1643</td>
<td>59.5</td>
<td>13.50</td>
</tr>
<tr>
<td>HUFF</td>
<td>22.8</td>
<td>884</td>
<td>55.3</td>
<td>5.10</td>
</tr>
<tr>
<td>DLB</td>
<td>22.6</td>
<td>880</td>
<td>55.8</td>
<td>5.18</td>
</tr>
<tr>
<td>LQF</td>
<td>23.5</td>
<td>877</td>
<td>53.5</td>
<td>4.92</td>
</tr>
<tr>
<td>BASIC HUFF</td>
<td>23.2</td>
<td>875</td>
<td>54.7</td>
<td>4.94</td>
</tr>
<tr>
<td>FIM</td>
<td>20.8</td>
<td>685</td>
<td>53.4</td>
<td>3.10</td>
</tr>
<tr>
<td>ESA</td>
<td>20.1</td>
<td>667</td>
<td>52.3</td>
<td>3.12</td>
</tr>
<tr>
<td>RLD</td>
<td>18.8</td>
<td>593</td>
<td>45.4</td>
<td>2.40</td>
</tr>
<tr>
<td>RLP</td>
<td>18.6</td>
<td>585</td>
<td>45.7</td>
<td>2.49</td>
</tr>
</tbody>
</table>
“Drivers and Loads” (trucking), CASTLE lab at Princeton

“the model was used by 20 of the largest truckload carriers to dispatch over 66,000 drivers”