What are Sequential Decision Problems?

• Until now, we mostly focused on “one-shot” decision making
  ▪ Outcome generally based on making one decision
  ▪ Even in repeated games we played, utility received in each round of repeated game is based on decision made in just that particular round

• Now, consider when agent’s (eventual) total utility depends on a sequence of decisions
  ▪ Example: extensive-form game
    ○ Winning or losing game depends on all moves till end of game
  ▪ Generally, agent wants to optimize utility over time
Sequential Decision Problem Set-Up

• Agent is situated in an environment (world)
  ▪ Agent is in a particular state, s, at any given time
    o For our discussion, we say environment is fully observable
      • I.e., agent can always recognize what state it is in
  ▪ In each state s, agent chooses an action, a, from set of available actions in that state, A(s)
    o Generally, not all action choices are available in all states
  ▪ Transition model: maps (state s, action a) to new state s’
    o Transition model is stochastic (i.e., probabilistic)
      • Taking an action in state may lead to different outcomes
        o More formally, transition model is given by P(s’ | s, a)
    o That is, after we take action a in state s, what is the probability that we are next in state s’
Markov Decision Process

- Note transition model: $P(s' \mid s, a)$
  - Assumption in model: the next state we move to, $s'$, only depends on current state $s$ and action we take
  - It does not depend on steps we previously took to get to state $s$
    - In other words, all that matters is where we are when we take the action, not where we were before
    - This is called the “Markov property”
Andrey Andreyevich Markov

- Andrey Andreyevich Markov (1856-1922) was a Russian mathematician
  - Markov property is named after him
  - He also invented Markov Chains…
    - …which are the basis for Google’s PageRank algorithm

That’s right, kids!
Markov Decision Process (Part I)

- Some states are *terminal states* in the sense that the decision problem is done when we reach them
  - i.e., Getting to winning or losing condition in a game
- A policy $\pi(s)$ maps from states to actions, specifying the agent’s recommended action in *any* state
  - We want to learn “good” policies of actions to take
  - Should be able to apply policy when in any state
Markov Decision Process (Part II)

- The agent receives a reward $R(s)$ when it enters a non-terminal state $s$
  - Rewards may be positive, zero, or negative
  - When agent enters a terminal state $s_T$, we determine “total” reward (utility $U$) based the sequence of states followed $(s_0, s_1, s_2, \ldots, s_T)$
    - This total reward may be additive:
      $$U(s_0, s_1, \ldots, s_T) = R(s_0) + R(s_1) + R(s_2) + \ldots + R(s_T)$$
    - Or, discounted at each step (where discount factor $\gamma \in (0, 1)$)
      $$U(s_0, s_1, \ldots, s_T) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots + \gamma^T R(s_T)$$
    - Generally, use discounted reward since “getting a dollar tomorrow is worth less than getting a dollar now”
Example: Grid World

- A maze-like problem
  - The agent lives in a grid
  - Walls block the agent’s path

- Potentially noisy movement: actions not always as planned
  - E.g., 80% of the time, the action North takes the agent North
  - 10% of the time, North takes the agent West; 10% East
  - If there is a wall in the direction the agent would have been taken, the agent stays put

- The agent receives rewards each time step
  - Small “living” reward each step (can be negative)
  - Big rewards come at the end (good or bad)

Thanks to Dan Klein and Pieter Abbeel
Expected Utility

- Say agent has policy $\pi(s)$ that it executes
  - Expected utility (with discounted rewards) of agent using policy $\pi$ where agent started in state $s$ is:
    \[
    U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]
    \]
  - Expectation is over state sequences determined by initial state $s$ and policy $\pi$ which determines actions agent takes
  - Want to find optimal policy (i.e., maximizes expected utility), which we denote $\pi^*$ ($s$ is starting state):
    \[
    \pi^*(s) = \arg\max_{\pi} U^\pi(s)
    \]
Determining Expected Utility

- Let $U(s)$ denote utility of executing $\pi^*(s)$
  - Then, by definition of expectation we have:
    \[ \pi^*(s) = \arg \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s') \]
  - Utility of state $U(s)$ is immediate reward $R(s)$ plus discounted expected utility of next state $s'$, assuming agent acts optimally
    \[ U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s') \]
  - This is called the Bellman Equation
Learning Policy: Value Iteration

- Value Iteration algorithm

Initialize $U'(s) = 0$ for all states $s$

While (not yet converged) {

Let $U(s) = U'(s)$ for all states $s$

for each state $s$ in $S$ {

$$U'(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'| s, a) U(s')$$

}

}

}
Learned Utilities

- Learned U(s) values after 100 iterations

Noise = 0
Discount = 1
Living reward = 0

Thanks to Dan Klein and Pieter Abbeel
Learned Values for Actions

- Learned values for actions after 100 iterations

Noise = 0
Discount = 1
Living reward = 0

Thanks to Dan Klein and Pieter Abbeel
Learned Utilities (Action Noise)

- Learned U(s) values after 100 iterations

Noise = 0.2
Discount = 1
Living reward = 0

Thanks to Dan Klein and Pieter Abbeel
Learned Values for Noisy Actions

- Learned values for actions after 100 iterations

![Grid with values](image)

- Noise = 0.2
- Discount = 1
- Living reward = 0

Thanks to Dan Klein and Pieter Abbeel
Learned Utilities (Action Noise + Discount)

- Learned U(s) values after 100 iterations

Noise = 0.2
Discount = 0.9
Living reward = 0

Thanks to Dan Klein and Pieter Abbeel
Learned Values for Noisy Actions+Discount

- Learned values for actions after 100 iterations

Noise = 0.2
Discount = 0.9
Living reward = 0

Thanks to Dan Klein and Pieter Abbeel
Noise = 0.2
Discount = 0.9
Living reward = 0
k=1

VALUES AFTER 1 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=2

Noise = 0.2
Discount = 0.9
Living reward = 0
\( k = 3 \)

VALUES AFTER 3 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=4

VALUES AFTER 4 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k = 5$

VALUES AFTER 5 ITERATIONS

Noise = 0.2  
Discount = 0.9  
Living reward = 0
Noise = 0.2
Discount = 0.9
Living reward = 0

VALUES AFTER 6 ITERATIONS

k=6
\(k=7\)

VALUES AFTER 7 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=8

VALUES AFTER 8 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
VALUES AFTER 9 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=10

Values after 10 iterations:

- Top row: 0.64, 0.74, 0.85, 1.00
- Second row: 0.56, 0.57, -1.00
- Bottom row: 0.48, 0.41, 0.47, 0.27

Noise = 0.2
Discount = 0.9
Living reward = 0
k = 11

VALUES AFTER 11 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=12$

VALUES AFTER 12 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
k=100

Noise = 0.2
Discount = 0.9
Living reward = 0

VALUES AFTER 100 ITERATIONS
Cost of Living

- Policies learned if we add negative reward $R(s)$ at each non-terminal state ("living cost")

\[
\begin{align*}
R(s) &= -0.03 \\
R(s) &= -0.4 \\
R(s) &= -2.0
\end{align*}
\]

Thanks to Dan Klein and Pieter Abbeel
Implications of Discounting

• Consider the following 3 x 101 grid world:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
<th>99</th>
<th>100</th>
<th>101</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>50</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>...</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-50</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>...</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
<tr>
<td>1</td>
<td>-50</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
<td>...</td>
<td>+1</td>
<td>+1</td>
<td>+1</td>
</tr>
</tbody>
</table>

- Values in grid denote rewards at each state
- First move is *Up* or *Down*, then make 100 *Right* moves
- Let $\gamma$ be discount factor
  - Utility of moving *Up* = $(\gamma^1)(50) + \sum_{t=2}^{101} (\gamma^t)(-1)$
  - Utility of moving *Down* = $(\gamma^1)(-50) + \sum_{t=2}^{101} (\gamma^t)(1)$
  - $\gamma = 1$: $U(\text{Up}) = -50$, $U(\text{Down}) = 50$ \(\Rightarrow\) Choose *Down*
  - $\gamma = 0.99$: $U(\text{Up}) = -12.6$, $U(\text{Down}) = 12.6$ \(\Rightarrow\) Choose *Down*
  - $\gamma = 0.98$: $U(\text{Up}) = 7.3$, $U(\text{Down}) = -7.3$ \(\Rightarrow\) Choose *Up*
  - ”Up” is really “dump waste in lake”
  - ”Down” is really “process waste, no dumping”
Exploration/Exploitation Trade-off

• Consider 100 x 100 grid:

<table>
<thead>
<tr>
<th></th>
<th>100</th>
<th>-1</th>
<th>-1</th>
<th>...</th>
<th>-1</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>-1</td>
<td>-1</td>
<td></td>
<td></td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>10</td>
<td></td>
<td></td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td></td>
<td></td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>...</td>
<td></td>
<td>99</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

• In real-world, we don’t have infinite time to try all actions in all states
  - Do we just “exploit” reward at (2, 2)?
  - Or do we “explore” hoping to find something better?
    - If so, how much exploring should we do?
    - How do we balance exploitation and exploration?