Planning and search

- Previously, problem solving by search A*
- Idea – get a state space, initial stage, goal state come up with a plan
- Everything is off-line
- What if things change? (e.g. walking blind folded)

- Need to interleave planning and executing
- Environment is stochastic (driving example)
- Partially observable, there can be many agents
- Model of the world is unknown, plans are hierarchical
Example: Grid World

- A maze-like problem
- The agent lives in a grid
- Walls block the agent’s path
- Noisy movement: actions do not always go as planned
- 80% of the time, the action North takes the agent North (if there is no wall there)
- 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)
- Goal: maximize sum of rewards

Grid World Actions

Deterministic Grid World

Stochastic Grid World
Markov Decision Processes

• An MDP is defined by:
  – A set of states \( s \in S \)
  – A set of actions \( a \in A \)
  – A transition function \( T(s, a, s') \)
    - Probability that \( a \) from \( s \) leads to \( s' \), i.e., \( P(s' | s, a) \)
    - Also called the model or the dynamics
  – A reward function \( R(s, a, s') \)
    - Sometimes just \( R(s) \) or \( R(s') \)
  – A start state
  – Maybe a terminal state

• MDPs are non-deterministic search problems
  – One way to solve them is with expectimax search
  – We’ll have a new tool soon

[Demo – gridworld manual intro (L8D1)]

What is Markov about MDPs?

• “Markov” generally means that given the present state, the future and the past are independent

• For Markov decision processes, “Markov” means action outcomes depend only on the current state
  \[ P(S_{t+1} = s' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, \ldots, S_0 = s_0) = \]

  \[ P(S_{t+1} = s' | S_t = s_t, A_t = a_t) \]

• This is just like search, where the successor function could only depend on the current state (not the history)
Policies

- In deterministic single-agent search problems, we wanted an optimal plan, or sequence of actions, from start to a goal.

- For MDPs, we want an optimal policy \( \pi^* : S \rightarrow A \):
  - A policy \( \pi \) gives an action for each state.
  - An optimal policy is one that maximizes expected utility if followed.
  - An explicit policy defines a reflex agent.

- Expectimax didn’t compute entire policies:
  - It computed the action for a single state only.

Optimal Policies

- Optimal policy when \( R(s, a, s') = -0.03 \) for all non-terminals \( s \).

- Examples of optimal policies for different rewards:
  - \( R(s) = -0.01 \)
  - \( R(s) = -0.4 \)
  - \( R(s) = -0.03 \)
  - \( R(s) = -2.0 \)
Example: Racing

- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: Slow, Fast
- Going faster gets double reward

Racing Search Tree
MDP Search Trees

- Each MDP state projects an expectimax-like search tree

Utilities of Sequences

- What preferences should an agent have over reward sequences?
  - More or less?  
    
    \[ [1, 2, 2] \text{ or } [2, 3, 4] \]
  - Now or later?  
    
    \[ [0, 0, 1] \text{ or } [1, 0, 0] \]
Discounting

- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially

\[
\begin{align*}
1 & \quad \text{Worth Now} \\
\gamma & \quad \text{Worth Next Step} \\
\gamma^2 & \quad \text{Worth In Two Steps}
\end{align*}
\]

Discounting

- How to discount?
  - Each time we descend a level, we multiply in the discount once

- Why discount?
  - Sooner rewards probably do have higher utility than later rewards
  - Also helps our algorithms converge

- Example: discount of 0.5
  - \( U(\{1,2,3\}) = 1 \cdot 1 + 0.5 \cdot 2 + 0.25 \cdot 3 \)
  - \( U(\{1,2,3\}) < U(\{3,2,1\}) \)
Stationary Preferences

• Theorem: if we assume stationary preferences:

\[ [a_1, a_2, \ldots] \succ [b_1, b_2, \ldots] \]
\[ \iff \]
\[ [r, a_1, a_2, \ldots] \succ [r, b_1, b_2, \ldots] \]

• Then: there are only two ways to define utilities
  – Additive utility: \( U([r_0, r_1, r_2, \ldots]) = r_0 + r_1 + r_2 + \cdots \)
  – Discounted utility: \( U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots \)

Quiz: Discounting

• Given:

<table>
<thead>
<tr>
<th>10</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

  – Actions: East, West, and Exit (only available in exit states a, e)
  – Transitions: deterministic

• Quiz 1: For \( \gamma = 1 \), what is the optimal policy? 10 1
• Quiz 2: For \( \gamma = 0.1 \), what is the optimal policy? 10 1
• Quiz 3: For which \( \gamma \) are West and East equally good when in state d?
Infinite Utilities?!

- Problem: What if the game lasts forever? Do we get infinite rewards?

- Solutions:
  - Finite horizon: (similar to depth-limited search)
    - Terminate episodes after a fixed T steps (e.g. life)
    - Gives nonstationary policies ($\pi$ depends on time $T$)
  - Discounting: use $0 < \gamma < 1$
    - Smaller $\gamma$ means smaller “horizon” – shorter term focus
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like “overheated” for racing)

Recap: Defining MDPs

- Markov decision processes:
  - Set of states $S$
  - Start state $s_0$
  - Set of actions $A$
  - Transitions $P(s'|s,a)$ (or $T(s,a,s')$)
  - Rewards $R(s,a,s')$ (and discount $\gamma$)

- MDP quantities so far:
  - Policy = Choice of action for each state
  - Utility = sum of (discounted) rewards
Solving MDPs

- **The value (utility) of a state** \( s \):
  \[ V^*(s) = \text{expected utility starting in } s \text{ and acting optimally} \]

- **The value (utility) of a q-state** \((s,a)\):
  \[ Q^*(s,a) = \text{expected utility starting out having taken action } a \text{ from state } s \text{ and (thereafter) acting optimally} \]

- **The optimal policy**:
  \[ \pi^*(s) = \text{optimal action from state } s \]

Optimal Quantities

- **The value (utility) of a state** \( s \):
  \[ V^*(s) = \text{expected utility starting in } s \text{ and acting optimally} \]

- **The value (utility) of a q-state** \((s,a)\):
  \[ Q^*(s,a) = \text{expected utility starting out having taken action } a \text{ from state } s \text{ and (thereafter) acting optimally} \]

- **The optimal policy**:
  \[ \pi^*(s) = \text{optimal action from state } s \]
Snapshot of Demo – Gridworld V Values

Noise = 0.2  
Discount = 0.9  
Living reward = 0

Snapshot of Demo – Gridworld Q Values

Noise = 0.2  
Discount = 0.9  
Living reward = 0
Values of States

- Fundamental operation: compute the (expectimax) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards

- Recursive definition of value:

\[
V^*(s) = \max_a Q^*(s, a)
\]
\[
Q^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right]
\]
\[
V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right]
\]

Racing Search Tree
Time-Limited Values

- Key idea: time-limited values
- Define $V_k(s)$ to be the optimal value of $s$ if the game ends in $k$ more time steps
  - Equivalently, it's what a depth-$k$ expectimax would give from $s$

[Demo – time-limited values (L8D6)]

\[ k = 0 \]

Noise = 0.2
Discount = 0.9
Living reward = 0
$k=1$

Noise = 0.2
Discount = 0.9
Living reward = 0

$k=2$

Noise = 0.2
Discount = 0.9
Living reward = 0
k=3

Noise = 0.2
Discount = 0.9
Living reward = 0

k=4

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

Values after 5 iterations:

Noise = 0.2
Discount = 0.9
Living reward = 0

$k=6$

Values after 6 iterations:

Noise = 0.2
Discount = 0.9
Living reward = 0
**k=7**

Noise = 0.2
Discount = 0.9
Living reward = 0

**k=8**

Noise = 0.2
Discount = 0.9
Living reward = 0
k=9

Noise = 0.2
Discount = 0.9
Living reward = 0

k=10

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

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

- Start with $V_0(s) = 0$: no time steps left means an expected reward sum of zero
- Given vector of $V_k(s)$ values, do one ply of expectimax from each state:
  \[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]
- Repeat until convergence
- Complexity of each iteration: $O(S^2A)$
- Theorem: will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do
Example: Value Iteration

\[ V_2 \]
\[
\begin{array}{ccc}
3.5 & 2.5 & 0 \\
\end{array}
\]

\[ V_1 \]
\[
\begin{array}{ccc}
2 & 1 & 0 \\
\end{array}
\]

\[ V_0 \]
\[
\begin{array}{ccc}
0 & 0 & 0 \\
\end{array}
\]

Assume no discount!

\[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]

Gridworld Values V*
Gridworld: $Q^*$

The Bellman Equations

- Definition of “optimal utility” via expectimax recurrence gives a simple one-step lookahead relationship amongst optimal utility values

\[
V^*(s) = \max_a Q^*(s, a)
\]

\[
Q^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right]
\]

\[
V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right]
\]

- These are the Bellman equations, and they characterize optimal values in a way we’ll use over and over
Value Iteration

- Bellman equations characterize the optimal values:
  \[ V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]

- Value iteration computes them:
  \[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]

- Value iteration is just a fixed point solution method
  - ... though the \( V_k \) vectors are also interpretable as time-limited values

Convergence*

- How do we know the \( V_k \) vectors are going to converge?

- Case 1: If the tree has maximum depth \( M \), then \( V_M \) holds the actual untruncated values

- Case 2: If the discount is less than 1
  - Sketch: For any state \( V_k \) and \( V_{k+1} \), can be viewed as depth \( k+1 \) expectimax results in nearly identical search trees
  - The difference is that on the bottom layer, \( V_{k+1} \) has actual rewards while \( V_k \) has zeros
  - That last layer is at best all \( r_{\text{MAX}} \)
  - It is at worst \( r_{\text{MIN}} \)
  - But everything is discounted by \( \gamma^k \) that far out
  - So \( V_k \) and \( V_{k+1} \) are at most \( \gamma^k \max |R| \) different
  - So as \( k \) increases, the values converge
Policy Methods

Fixed Policies

- Expectimax trees max over all actions to compute the optimal values
- If we fixed some policy $\pi(s)$, then the tree would be simpler – only one action per state
  - ... though the tree’s value would depend on which policy we fixed
Utilities for a Fixed Policy

- Another basic operation: compute the utility of a state $s$ under a fixed (generally non-optimal) policy

- Define the utility of a state $s$, under a fixed policy $\pi$:
  $V^\pi(s) =$ expected total discounted rewards starting in $s$ and following $\pi$

- Recursive relation (one-step look-ahead / Bellman equation):
  $V^\pi(s) = \sum_{s'} T(s, \pi(s), s') \{ R(s, \pi(s), s') + \gamma V^\pi(s') \}$

Example: Policy Evaluation

Always Go Right

Always Go Forward
Example: Policy Evaluation

Always Go Right

Always Go Forward

Policy Evaluation

• How do we calculate the $V$'s for a fixed policy $\pi$?

• Idea 1: Turn recursive Bellman equations into updates (like value iteration)

  \[
  V_0^\pi(s) = 0
  \]

  \[
  V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^\pi(s')]
  \]

  • Efficiency: $O(S^2)$ per iteration

• Idea 2: Without the maxes, the Bellman equations are just a linear system
  – Solve with Matlab (or your favorite linear system solver)
Policy Extraction

Let's imagine we have the optimal values $V^*(s)$.

How should we act?
- It's not obvious!

We need to do a mini-expectimax (one step)

This is called policy extraction, since it gets the policy implied by the values.

$$\pi^*(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$
Computing Actions from Q-Values

- Let’s imagine we have the optimal q-values:

- How should we act?
  - Completely trivial to decide!

\[ \pi^*(s) = \arg\max_a Q^*(s, a) \]

- Important lesson: actions are easier to select from q-values than values!

Policy Iteration
Problems with Value Iteration

- Value iteration repeats the Bellman updates:

\[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]

- Problem 2: The “max” at each state rarely changes

- Problem 3: The policy often converges long before the values

[Demo: value iteration (L9D2)]

\[ k = 0 \]

Noise = 0.2
Discount = 0.9
Living reward = 0
k=1

Noise = 0.2
Discount = 0.9
Living reward = 0

k=2

Noise = 0.2
Discount = 0.9
Living reward = 0
k=3

Noise = 0.2
Discount = 0.9
Living reward = 0

k=4

Noise = 0.2
Discount = 0.9
Living reward = 0
\( k = 5 \)

Noise = 0.2  
Discount = 0.9  
Living reward = 0

\( k = 6 \)

Noise = 0.2  
Discount = 0.9  
Living reward = 0
\[ k = 7 \]

Noise = 0.2
Discount = 0.9
Living reward = 0

\[ k = 8 \]

Noise = 0.2
Discount = 0.9
Living reward = 0
k=9

Noise = 0.2
Discount = 0.9
Living reward = 0

k=10

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

Noise = 0.2
Discount = 0.9
Living reward = 0

k=12

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

- Alternative approach for optimal values:
  - **Step 1: Policy evaluation**: calculate utilities for some fixed policy (not optimal utilities!) until convergence
  - **Step 2: Policy improvement**: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values
  - Repeat steps until policy converges

- This is **policy iteration**
  - It's still optimal!
  - Can converge (much) faster under some conditions
Policy Iteration

• Evaluation: For fixed current policy $\pi$, find values with policy evaluation:
  – Iterate until values converge:
    $$V_{k+1}^{\pi_i}(s) \leftarrow \sum_{s'} T(s, \pi_i(s), s') \left[ R(s, \pi_i(s), s') + \gamma V_k^{\pi_i}(s') \right]$$

• Improvement: For fixed values, get a better policy using policy extraction
  – One-step look-ahead:
    $$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k^{\pi_i}(s') \right]$$

Comparison

• Both value iteration and policy iteration compute the same thing (all optimal values)

• In value iteration:
  – Every iteration updates both the values and (implicitly) the policy
  – We don’t track the policy, but taking the max over actions implicitly recomputes it

• In policy iteration:
  – We do several passes that update utilities with fixed policy (each pass is fast because we consider only one action, not all of them)
  – After the policy is evaluated, a new policy is chosen (slow like a value iteration pass)
  – The new policy will be better (or we’re done)

• Both are dynamic programs for solving MDPs
Summary: MDP Algorithms

• So you want to….
  – Compute optimal values: use value iteration or policy iteration
  – Compute values for a particular policy: use policy evaluation
  – Turn your values into a policy: use policy extraction (one-step lookahead)

• These all look the same!
  – They basically are – they are all variations of Bellman updates
  – They all use one-step lookahead expectimax fragments
  – They differ only in whether we plug in a fixed policy or max over actions

Double Bandits
Double-Bandit MDP

- Actions: Blue, Red
- States: Win, Lose

\[ \begin{array}{c|cc}
\text{State} & 0.75 & 0.25 \\
\hline
\text{Win} & \$2 & \$0 \\
\text{Lose} & \$2 & \$0 \\
\end{array} \]

No discount
100 time steps
Both states have the same value

Offline Planning

- Solving MDPs is offline planning
  - You determine all quantities through computation
  - You need to know the details of the MDP
  - You do not actually play the game!

<table>
<thead>
<tr>
<th>Value</th>
<th>Play Red</th>
<th>150</th>
<th>Play Blue</th>
<th>100</th>
</tr>
</thead>
</table>

No discount
100 time steps
Both states have the same value
Let’s Play!

Online Planning

- Rules changed! Red’s win chance is different.
Let’s Play!

$0$ $0$ $0$ $2$ $0$
$2$ $0$ $0$ $0$ $0$

What Just Happened?

• That wasn’t planning, it was learning!
  – Specifically, reinforcement learning
  – There was an MDP, but you couldn’t solve it with just computation
  – You needed to actually act to figure it out

• Important ideas in reinforcement learning that came up
  – Exploration: you have to try unknown actions to get information
  – Exploitation: eventually, you have to use what you know
  – Regret: even if you learn intelligently, you make mistakes
  – Sampling: because of chance, you have to try things repeatedly
  – Difficulty: learning can be much harder than solving a known MDP