Non-Deterministic Search


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

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 (or reward)
  - An explicit policy defines a reflex agent

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

Discounting
- How to discount?
  - Each time we descend a level, we multiply by the discount factor
- Why discount?
  - Think of it as a gamma chance of ending the process at every step
  - Also helps our algorithms converge
- Example: discount of 0.5
  - \( U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3 \)
  - \( U([1,3,2]) = 1*1 + 0.5*3 + 0.25*2 \)

Solving MDPs
We're doing way too much work with expectimax!

- Problem: States are repeated
  - Idea: Only compute needed quantities once

- Problem: Tree goes on forever
  - Idea: Do a depth-limited computation, but with increasing depths until change is small
  - Note: Deep parts of the tree eventually don’t matter if $\gamma < 1$

**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
- Recursive definition of value:
  
  $V^*(s) = \max_a Q^*(s, a)$
  
  $Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$
  
  $V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$

Bellman Equations
- Recursive definition of value:
  
  $V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$

- Bellman Equation:
  Necessary condition for optimality in optimization problems formulated as Dynamic Programming

- Dynamic Programming:
  Process to simplify an optimization problem by breaking it down into an optimal substructure.

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 $s$.

Noise = 0.2
Discount = 0.9
Living reward = 0
<table>
<thead>
<tr>
<th>$k$</th>
<th>Noise</th>
<th>Discount</th>
<th>Living reward</th>
<th>Values after 3 iterations</th>
<th>Values after 4 iterations</th>
<th>Values after 5 iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0.2</td>
<td>0.9</td>
<td>0</td>
<td>$0.00$ $0.32$ $0.76$ $1.00$</td>
<td>$0.00$ $0.45$ $1.00$</td>
<td>$0.51$ $0.73$ $0.84$ $1.00$</td>
</tr>
<tr>
<td>4</td>
<td>0.2</td>
<td>0.9</td>
<td>0</td>
<td>$0.00$ $0.45$ $1.00$</td>
<td>$0.00$ $0.45$ $1.00$</td>
<td>$0.51$ $0.73$ $0.84$ $1.00$</td>
</tr>
<tr>
<td>5</td>
<td>0.2</td>
<td>0.9</td>
<td>0</td>
<td>$0.00$ $0.45$ $1.00$</td>
<td>$0.00$ $0.45$ $1.00$</td>
<td>$0.51$ $0.73$ $0.84$ $1.00$</td>
</tr>
<tr>
<td>6</td>
<td>0.2</td>
<td>0.9</td>
<td>0</td>
<td>$0.00$ $0.45$ $1.00$</td>
<td>$0.00$ $0.45$ $1.00$</td>
<td>$0.51$ $0.73$ $0.84$ $1.00$</td>
</tr>
<tr>
<td>7</td>
<td>0.2</td>
<td>0.9</td>
<td>0</td>
<td>$0.00$ $0.45$ $1.00$</td>
<td>$0.00$ $0.45$ $1.00$</td>
<td>$0.51$ $0.73$ $0.84$ $1.00$</td>
</tr>
<tr>
<td>8</td>
<td>0.2</td>
<td>0.9</td>
<td>0</td>
<td>$0.00$ $0.45$ $1.00$</td>
<td>$0.00$ $0.45$ $1.00$</td>
<td>$0.51$ $0.73$ $0.84$ $1.00$</td>
</tr>
</tbody>
</table>
\( k = 9 \)

VALUES AFTER 9 ITERATIONS

\[
\begin{array}{cccc}
0.64 & 0.74 & 0.85 & 1.00 \\
0.52 & 0.51 & & 1.00 \\
0.44 & 0.40 & 0.47 & 0.27 \\
\end{array}
\]

\( k = 10 \)

VALUES AFTER 10 ITERATIONS

\[
\begin{array}{cccc}
0.64 & 0.74 & 0.85 & 1.00 \\
0.58 & 0.57 & & 1.00 \\
0.48 & 0.44 & 0.47 & 0.27 \\
\end{array}
\]

\( k = 11 \)

VALUES AFTER 11 ITERATIONS

\[
\begin{array}{cccc}
0.64 & 0.74 & 0.85 & 1.00 \\
0.58 & 0.57 & & 1.00 \\
0.48 & 0.44 & 0.47 & 0.27 \\
\end{array}
\]

\( k = 12 \)

VALUES AFTER 12 ITERATIONS

\[
\begin{array}{cccc}
0.64 & 0.74 & 0.85 & 1.00 \\
0.57 & 0.51 & & 1.00 \\
0.48 & 0.42 & 0.47 & 0.28 \\
\end{array}
\]

\( k = 100 \)

VALUES AFTER 100 ITERATIONS

\[
\begin{array}{cccc}
0.64 & 0.74 & 0.85 & 1.00 \\
0.57 & 0.57 & & 1.00 \\
0.48 & 0.42 & 0.48 & 0.28 \\
\end{array}
\]

Value Iteration
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:
  \[ V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')] \]
- 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

<table>
<thead>
<tr>
<th>$V_0$</th>
<th>$V_1$</th>
<th>$V_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Assume no discount!

Example: Value Iteration

<table>
<thead>
<tr>
<th>$V_0$</th>
<th>$V_1$</th>
<th>$V_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Assume no discount!

Example: Value Iteration

<table>
<thead>
<tr>
<th>$V_0$</th>
<th>$V_1$</th>
<th>$V_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Assume no discount!
Convergence

- How do we know the Vk vectors are going to converge?

  Case 1: If the tree has maximum depth M, then Vk holds the actual untruncated values.

  Case 2: If the discount is less than 1

  Sketch: For any state Vk and Vk+1 can be viewed as depth k+1 expectimax results in nearly identical search trees.

  The difference is that on the bottom layer, Vk+1 has actual rewards while Vk has zeros.

  That last layer is at best all RMAX.

  It is at worst RMIN.

  But everything is discounted by γ that far out.

  So Vk and Vk+1 are at most γmax|R| different.

  So as k increases, the values converge.

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.

Computing Actions from Values

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

- How should we act?

  Completely trivial to decide!

  r*(s) = arg max_u Qk(s, u)

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

Policy Methods

- Let’s think.

  Take a minute, think about value iteration.

  Write down the biggest question you have about it.
Problems with Value Iteration

- Value iteration repeats the Bellman updates:
  \[ v_{k+1}(s) = \max_{a} \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma v_k(s') \right] \]
- Problem 1: It’s slow – \( O(S^2A) \) per iteration
- Problem 2: The “max” at each state rarely changes
- Problem 3: The policy often converges long before the values

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

Policy Evaluation

- Do the optimal action
- Do what \( \pi \) says to do

Fixed Policies

- Expectimax trees run 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 \( \pi \):
  \[ V^\pi(s) = \text{expected total discounted rewards starting in } s \text{ and following } \pi \]
- Recursive relation (one-step look-ahead / Bellman equation):
  \[ V^\pi(s) = \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V^\pi(s')] \]

Policy Evaluation

- How do we calculate the \( V^\pi \)'s for a fixed policy \( \pi \)?
  - Idea 1: Turn recursive Bellman equations into updates (like value iteration)
  - 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)

Example: Policy Evaluation

Always Go Right

Always Go Forward

Policy Iteration

- Evaluation: For fixed current policy \( \pi \), find values with policy evaluation
  - Iterate until values converge:
  \[ V^\pi_{k+1}(s) = \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V^\pi_k(s')] \]
- Improvement: For fixed values, get a better policy using policy extraction
  - One-step look-ahead:
  \[ \pi_{k+1}(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^\pi_k(s')] \]
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 (like a value iteration pass).
  - The new policy will be better (or we’re done).
- Both are dynamic programs for solving MDPs.

Summary: MDP Equations
- Value iteration equation:
  \[ V_{k+1}(s) = \max_u \sum_s T(u,s,s') [R(s,u,s') + \gamma V_k(s')] \]
- Policy evaluation equation:
  \[ V_k(s) \leftarrow \sum_s T(s,\pi_k(s),s') [R(s,\pi_k(s),s') + \gamma V_k(s')] \]
- Policy iteration equation:
  \[ \pi_{k+1}(s) = \arg\max_u \sum_s T(s,u,s') [R(s,u,s') + \gamma V_k(s')] \]

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 are 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.

The Bellman Equations

Next Time: Reinforcement Learning!