CS 188: Artificial Intelligence
Spring 2011

Lecture 9: MDPs
2/16/2011

Pieter Abbeel – UC Berkeley
Many slides over the course adapted from either Dan Klein, Stuart Russell or Andrew Moore

Outline

- Markov Decision Processes (MDPs)
  - Formalism
  - Value iteration
- Expectimax Search vs. Value Iteration
  - Value Iteration:
    - No exponential blow-up with depth [cf. graph search vs. tree search]
    - Can handle infinite duration games
- Policy Evaluation and Policy Iteration

Reinforcement Learning

- Basic idea:
  - Receive feedback in the form of rewards
  - Agent’s utility is defined by the reward function
  - Must learn to act so as to maximize expected rewards

Grid World

- The agent lives in a grid
- Walls block the agent’s path
- The agent’s 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
- Small “living” reward each step
- Big rewards come at the end
- Goal: maximize sum of rewards

Grid Futures

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 \( T(s, a, s') \)
    - Prob that a from \( s \) leads to \( s' \)
    - i.e., \( P(s' | s, a) \)
  - Also called the model
  - A reward function \( R(s, a, s') \)
    - Sometimes just \( R(s) \) or \( R(s') \)
  - A start state (or distribution)
  - Maybe a terminal state

- MDPs are a family of non-deterministic search problems
  - Reinforcement learning: MDPs where we don’t know the transition or reward functions

What is Markov about MDPs?

- Andrey Markov (1856-1922)
- “Markov” generally means that given the present state, the future and the past are independent
- For Markov decision processes, “Markov” means:

\[
P(S_{t+1} = s' | S_t = s, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1}, \ldots, S_0 = s_0) = \\
P(S_{t+1} = s' | S_t = s_t, A_t = a_t)
\]

Solving MDPs

- In deterministic single-agent search problems, want an optimal plan, or sequence of actions, from start to a goal
- In an MDP, we want an optimal policy \( \pi^* : S \to A \)
  - A policy \( \pi \) gives an action for each state
  - An optimal policy maximizes expected utility if followed
  - Defines a reflex agent

Example: High-Low

- Three card types: 2, 3, 4
- Infinite deck, twice as many 2’s
- Start with 3 showing
- After each card, you say “high” or “low”
- New card is flipped
- If you’re right, you win the points shown on the new card
- Ties are no-ops
- If you’re wrong, game ends

- Differences from expectimax:
  - #1: get rewards as you go
  - #2: you might play forever!
Example: High-Low

<table>
<thead>
<tr>
<th>T = 0.25, R = 2</th>
<th>T = 0.25, R = 3</th>
<th>T = 0, R = 4</th>
<th>T = 0.25, R = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 High Low</td>
<td>3 Low</td>
<td>3 High</td>
<td>4 High Low</td>
</tr>
</tbody>
</table>

Utilities of Sequences

- In order to formalize optimality of a policy, need to understand utilities of sequences of rewards
- Typically consider stationary preferences:
  \[ [r_0, r_1, r_2, \ldots] > [r_0', r_1', r_2', \ldots] \]
  \[ [r_0', r_1', r_2', \ldots] > [r_0'', r_1'', r_2'', \ldots] \]

- Theorem: only two ways to define stationary utilities
  - Additive utility:
    \[ U([r_0, r_1, r_2, \ldots]) = r_0 + r_1 + r_2 + \ldots \]
  - Discounted utility:
    \[ U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots \]

MDP Search Trees

- Each MDP state gives an expectimax-like search tree

Discounting

- Typically discount rewards by \( \gamma < 1 \) each time step
  - Sooner rewards have higher utility than later rewards
  - Also helps the algorithms converge

Infinite Utilities?!

- Problem: infinite state sequences have infinite rewards

- Solutions:
  - Finite horizon: terminate episodes after a fixed T steps (e.g. life)
  - Gives nonstationary policies (\( \pi \) depends on time left)
  - Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like “done” for High-Low)
  - Discounting: for \( 0 < \gamma < 1 \)
    \[ U([r_0, \ldots, r_T]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\text{max}}/(1 - \gamma) \]
    - Smaller \( \gamma \) means smaller “horizon” – shorter term focus

Recap: Defining MDPs

- Markov decision processes:
  - States \( S \)
  - Start state \( s_0 \)
  - 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 (or return) = sum of discounted rewards
**Optimal Utilities**

- Fundamental operation: compute the values (optimal expectimax utilities) of states $s$.
- Why? Optimal values define optimal policies!
- Define the value of a state $s$: $V^*(s) = \text{expected utility starting in } s \text{ and acting optimally}$
- Define the value of a q-state $(s,a)$: $Q^*(s,a) = \text{expected utility starting in } s, \text{ taking action } a, \text{ and thereafter acting optimally}$
- Define the optimal policy: $\pi^*(s) = \text{optimal action from state } s$

**Value Estimates**

- Calculate estimates $V_k^*(s)$. Not the optimal value of $s$!
- The optimal value considering only next $k$ time steps ($k$ rewards)
- As $k \to \infty$, it approaches the optimal value
- Almost solution: recursion (i.e. expectimax)
- Correct solution: dynamic programming

---

**Value Iteration: $V^*_1$**

**Value Iteration: $V^*_2$**

**Value Iteration $V^*_{i+1}$**

**Value Iteration**

- Idea:
  - $V^*_i(s)$: the expected discounted sum of rewards accumulated when starting from state $s$ and acting optimally for a horizon of $i$ time steps.
- Start with $V_0^*(s) = 0$, which we know is right (why?)
- Given $V^*_i$, calculate the values for all states for horizon $i+1$:
  
  \[ V^*_{i+1}(s) \leftarrow \max_{a,a'} \sum_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma V^*_i(s') \right] \]

- This is called a value update or Bellman update
- Repeat until convergence

- Theorem: will converge to unique optimal values
- Basic idea: approximations get refined towards optimal values
- Policy may converge long before values do
Example: Bellman Updates

Example: Bellman Updates

<table>
<thead>
<tr>
<th></th>
<th>V1</th>
<th>V2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ V_{i+1}(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_i(s') \right] \]

\[ V_2((3, 3)) = \sum_{s'} T((3, 3), \text{right}, s') \left[ R((3, 3)) + 0.9 V_1(s') \right] = 0.9 \left[ 0.8 \cdot 1 + 0.1 \cdot 0 + 0.1 \cdot 0 \right] \]

Convergence*

- Define the max-norm: \[ ||U|| = \max_s |U(s)| \]

- Theorem: For any two approximations \( U \) and \( V \)

\[ ||U_{i+1} - V_{i+1}|| \leq \gamma ||U_i - V_i|| \]

- I.e. any distinct approximations must get closer to each other, so, in particular, any approximation must get closer to the true \( U \) and value iteration converges to a unique, stable, optimal solution

- Theorem:

\[ ||U_{i+1} - U_i|| < \epsilon \Rightarrow ||U_{i+1} - U|| < 2\epsilon/(1 - \gamma) \]

- I.e. once the change in our approximation is small, it must also be close to correct

At Convergence

- At convergence, we have found the optimal value function \( V^* \) for the discounted infinite horizon problem, which satisfies the Bellman equations:

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

The Bellman Equations

- Definition of “optimal utility” leads to a simple one-step lookahead relationship amongst optimal utility values:

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

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

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

Practice: Computing Actions

- Which action should we chose from state \( s \):
  - Given optimal values \( V \)
    
    \[ \arg \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V^*(s') \right] \]
  
  - Given optimal q-values \( Q \)
    
    \[ \arg \max_a Q^*(s, a) \]
  
  - Lesson: actions are easier to select from Q’s!

Complete Procedure

- 1. Run value iteration (off-line)

\( \Rightarrow \) Returns \( V \), which (assuming sufficiently many iterations is a good approximation of \( V^* \))

- 2. Agent acts. At time \( t \) the agent is in state \( s_t \) and takes the action \( a_t \):

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

Outline

- Markov Decision Processes (MDPs)
  - Formalism
  - Value iteration
- Expectimax Search vs. Value Iteration
  - Value Iteration:
    - No exponential blow-up with depth [cf. graph search vs. tree search]
    - Can handle infinite duration games
- Policy Evaluation and Policy Iteration

Why Not Search Trees?

- Why not solve with expectimax?
- Problems:
  - This tree is usually infinite (why?)
  - Same states appear over and over (why?)
  - We would search once per state (why?)
- Idea: Value iteration
  - Compute optimal values for all states all at once using successive approximations
  - Will be a bottom-up dynamic program similar in cost to memoization
  - Do all planning offline, no replanning needed!

Expectimax vs. Value Iteration: $V_1^*$

Expectimax vs. Value Iteration: $V_2^*$

Outline

- Markov Decision Processes (MDPs)
  - Formalism
  - Value iteration
- Expectimax Search vs. Value Iteration
  - Value Iteration:
    - No exponential blow-up with depth [cf. graph search vs. tree search]
    - Can handle infinite duration games
- Policy Evaluation and Policy Iteration
Utilities for Fixed Policies

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

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

- 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 $V$'s for a fixed policy?

  Idea one: modify Bellman updates
  \[
  V_0^\pi(s) = 0 \\
  V_{k+1}^\pi(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_k^\pi(s')] 
  \]

  Idea two: it's just a linear system, solve with Matlab (or whatever)

Policy Iteration

- Alternative approach:
  - 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

- This is policy iteration
  - It's still optimal!
  - Can converge faster under some conditions

Comparison

- In value iteration:
  - Every pass (or "backup") updates both utilities (explicitly, based on current utilities) and policy (possibly implicitly, based on current policy)

- In policy iteration:
  - Several passes to update utilities with frozen policy
  - Occasional passes to update policies

- Hybrid approaches (asynchronous policy iteration):
  - Any sequences of partial updates to either policy entries or utilities will converge if every state is visited infinitely often

Asynchronous Value Iteration*

- In value iteration, we update every state in each iteration

  Actually, any sequences of Bellman updates will converge if every state is visited infinitely often

  In fact, we can update the policy as seldom or often as we like, and we will still converge

  Idea: Update states whose value we expect to change:
  - If $|P_{\pi_0}(s) \rightarrow T(s)|$ is large then update predecessors of $s$
MDPs recap

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

- **Solution methods:**
  - Value iteration (VI)
  - Policy iteration (PI)
  - Asynchronous value iteration

- **Current limitations:**
  - Relatively small state spaces
  - Assumes $T$ and $R$ are known