Announcements

- **Homework 2**
  - Due 2/11 at 11:59pm
  - Electronic HW2
  - Written HW2

- **Project 1**
  - Due Friday 2/8 at 4:00pm

- **Mini-contest 1 (optional)**
  - Due 2/11 at 11:59pm
CS 188: Artificial Intelligence

Expectimax & Markov Decision Processes

Instructors: Sergey Levine and Stuart Russell

University of California, Berkeley

[slides adapted from Dan Klein and Pieter Abbeel http://ai.berkeley.edu]
Worst-Case vs. Average Case
Expectimax Search

- Why wouldn’t we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Actions can fail: when moving a robot, wheels might slip

- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes

- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
  - I.e. take weighted average (expectation) of children

- Later, we’ll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes
**Expectimax Pseudocode**

```python
def value(state):
    if the state is a terminal state: return the state’s utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)

def max-value(state):
    initialize v = -\infty
    for each successor of state:
        v = max(v, value(successor))
    return v

def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v

v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10
Expectimax Example

3 -> 12 -> 9
2 -> 4 -> 6
15 -> 6 -> 0
Depth-Limited Expectimax

Estimate of true expectimax value (which would require a lot of work to compute)
Probabilities
Reminder: Probabilities

- A random variable represents an event whose outcome is unknown
- A probability distribution is an assignment of weights to outcomes

Example: Traffic on freeway
- Random variable: $T =$ whether there’s traffic
- Outcomes: $T$ in \{none, light, heavy\}
- Distribution: $P(T=\text{none}) = 0.25$, $P(T=\text{light}) = 0.50$, $P(T=\text{heavy}) = 0.25$

Some laws of probability:
- Probabilities are always non-negative
- Probabilities over all possible outcomes sum to one
The expected value of a function of a random variable is the average, weighted by the probability distribution over outcomes.

Example: How long to get to the airport?

<table>
<thead>
<tr>
<th>Time</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 min</td>
<td>0.25</td>
</tr>
<tr>
<td>30 min</td>
<td>0.50</td>
</tr>
<tr>
<td>60 min</td>
<td>0.25</td>
</tr>
</tbody>
</table>

$35 \text{ min}$
In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state:

- Model could be a simple uniform distribution (roll a die)
- Model could be sophisticated and require a great deal of computation
- We have a chance node for any outcome out of our control: opponent or environment
- The model might say that adversarial actions are likely!

For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes.

Having a probabilistic belief about another agent’s action does not mean that the agent is flipping any coins!
Let’s say you know that your opponent is actually running a depth 2 minimax, using the result 80% of the time, and moving randomly otherwise.

Question: What tree search should you use?

Answer: Expectimax!

- To figure out EACH chance node’s probabilities, you have to run a simulation of your opponent.
- This kind of thing gets very slow very quickly.
- Even worse if you have to simulate your opponent simulating you...
- ... except for minimax, which has the nice property that it all collapses into one game tree.
Other Game Types
Mixed Layer Types

- E.g. Backgammon
- Expectiminimax
  - Environment is an extra “random agent” player that moves after each min/max agent
  - Each node computes the appropriate combination of its children
Example: Backgammon

- Dice rolls increase $b$: 21 possible rolls with 2 dice
  - Backgammon $\approx$ 20 legal moves
  - Depth $2 = 20 \times (21 \times 20)^3 = 1.2 \times 10^9$

- As depth increases, probability of reaching a given search node shrinks
  - So usefulness of search is diminished
  - So limiting depth is less damaging
  - But pruning is trickier...

- Historic AI: TDGammon uses depth-2 search + very good evaluation function + reinforcement learning: world-champion level play

- 1st AI world champion in any game!
Multi-Agent Utilities

- What if the game is not zero-sum, or has multiple players?

- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically...

```
1, 6, 6  7, 1, 2  6, 1, 2  7, 2, 1  5, 1, 7  1, 5, 2  7, 7, 1  5, 2, 5
```
Non-Deterministic Search

[Diagram showing a robot facing a cliff with a diamond, and a decision tree with max and chance nodes and numerical values.]
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:

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

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

Andrey Markov (1856-1922)
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

\[ R(s) = -2.0 \]

\[ R(s) = -0.4 \]

\[ R(s) = -0.03 \]

\[ R(s) = -0.01 \]
Example: Racing
Example: Racing

- A robot car wants to travel far, quickly
- Three states: Cool, Warm, Overheated
- Two actions: Slow, Fast
- Going faster gets double reward
Each MDP state projects an expectimax-like search tree as a state.

(s, a) is a q-state

(s, a, s’) called a transition

T(s, a, s’) = P(s’ | s, a)

R(s, a, s’)
Utilities of Sequences
Utilities of Sequences

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

\[
\begin{align*}
\text{Worth Now} & : 1 \\
\text{Worth Next Step} & : \gamma \\
\text{Worth In Two Steps} & : \gamma^2
\end{align*}
\]
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**
  - $U([1,2,3]) = 1 \times 1 + 0.5 \times 2 + 0.25 \times 3$
  - $U([1,2,3]) < U([3,2,1])$
Theorem: if we assume stationary preferences:

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

\[ \Downarrow \]

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

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

Quiz 1: For $\gamma = 1$, what is the optimal policy?

Quiz 2: For $\gamma = 0.1$, what is the optimal policy?

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 left)
  - Discounting: use $0 < \gamma < 1$
    \[
    U([r_0, \ldots, r_\infty]) = \sum_{t=0}^{\infty} \gamma^t r_t \leq R_{\text{max}}/(1 - \gamma)
    \]
    - 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
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

VALUES AFTER 100 ITERATIONS

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
  - This is just what expectimax computed!

- 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
- 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 $\gamma < 1$
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 $s$
$k = 1$

VALUES AFTER 1 ITERATIONS

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

VALUES AFTER 2 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
Values after 3 iterations

k=3

Noise = 0.2
Discount = 0.9
Living reward = 0
<table>
<thead>
<tr>
<th></th>
<th>0.37</th>
<th>0.66</th>
<th>0.83</th>
<th>1.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.51</td>
<td>-1.00</td>
<td></td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.31</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

VALUES AFTER 4 ITERATIONS

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

VALUES AFTER 5 ITERATIONS

- Top left cell: 0.51
- Top middle cell: 0.72
- Top right cell: 0.84
- Right cell: 1.00

- Middle left cell: 0.27
- Middle middle cell: 0.55
- Middle right cell: -1.00

- Bottom left cell: 0.00
- Bottom middle cell: 0.22
- Bottom right cell: 0.13

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

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
k=9

VALUES AFTER 9 ITERATIONS

0.64  0.74  0.85  1.00
0.55  0.57  -1.00
0.46  0.40  0.47  0.27

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

VALUES AFTER 10 ITERATIONS

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

VALUES AFTER 100 ITERATIONS

Noise = 0.2
Discount = 0.9
Living reward = 0
Computing Time-Limited Values

$V_4(\text{car}) \quad V_4(\text{car}) \quad V_4(\text{car})$

$V_3(\text{car}) \quad V_3(\text{car}) \quad V_3(\text{car})$

$V_2(\text{car}) \quad V_2(\text{car}) \quad V_2(\text{car})$

$V_1(\text{car}) \quad V_1(\text{car}) \quad V_1(\text{car})$

$V_0(\text{car}) \quad V_0(\text{car}) \quad V_0(\text{car})$
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 step 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

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

Assume no discount!
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
Next Time: Policy-Based Methods