Solving MDPs

- Value Iteration
- Policy Iteration
- Reinforcement Learning

Optimal Quantities

- The value (utility) of a state $s$: $V^*(s)$ = expected utility starting in $s$ and acting optimally
- The value (utility) of a q-state $(s,a)$: $Q^*(s,a)$ = expected utility starting out having taken action $a$ from state $s$ and (thereafter) acting optimally
- The optimal policy: $\pi^*(s)$ = optimal action from state $s$

Values of States

- Fundamental operation: compute the (expectmax) value of a state
  - Expected utility under optimal action
  - Average sum of (discounted) rewards
  - This is just what expectimax computed!
  
  Recursive definition of value:
  
  \[
  V^*(s) = \max_a \sum_{s'} T(s,a,s') [r(s,a,s') + \gamma V^*(s')] \\
  Q^*(s,a) = \sum_{s'} T(s,a,s') [r(s,a,s') + \gamma V^*(s')] \\
  \]

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
- Notes: 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 $s$

- Time-Limited Values
- Computing Time-Limited Values

Value Iteration

The Bellman Equations

How to be optimal:

Step 1: Take correct first action
Step 2: Keep being optimal

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:
  \[ V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]
- Value iteration is just a fixed point solution method
  - though the \( V_k \) vectors are also interpretable as time-limited values

Value Iteration Algorithm

- Start with \( V_0(s) = 0 \):
- Given vector of \( V_k(s) \) values, do one ply of expectimax from each state:
  \[ V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_k(s') \right] \]
- Repeat until convergence
- Complexity of each iteration: \( O(S^2 A) \)
- Number of iterations: \( \text{poly}(|S|, |A|, 1/(1-\gamma)) \)
- Theorem: will converge to unique optimal values

### k=0

VALUES AFTER 0 ITERATIONS

### k=1

VALUES AFTER 1 ITERATIONS

### k=2

VALUES AFTER 2 ITERATIONS

### k=3

VALUES AFTER 3 ITERATIONS
$k=4$

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

$k=6$

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

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

- How do we know the $V_k$ vectors will 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
  - Search: For any state $V_k$ and $V_{k+1}$ can be viewed as depth $k+1$ expectimax results in nearly identical search trees
  - The max difference happens if big reward at $k+1$ level
  - That last layer is at best all $R_{MAX}$
  - But everything is discounted by $\gamma_k$ that far out
  - So $V_k$ and $V_{k+1}$ are at most $\gamma^k$ (if $|R|$ different
  - So as $k$ increases, the values converge

Computing Actions from Values

- 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)
  \[ \pi^*(s) = \arg \max_a \sum_{a'} T(s, a, a') [R(s, a, a') + \gamma V^*(s')] \]
- This is called policy extraction, since it gets the policy implied by the values
Computing Actions from Q-Values

- Let’s imagine we have the optimal q-values:
  \[ \pi^*(s) = \arg \max_n Q^*(s, a) \]
- How should we act?
  - Completely trivial to decide!
- Important lesson: actions are easier to select from q-values than values!

Problems with Value Iteration

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

VI → Asynchronous VI

- Is it essential to back up all states in each iteration?
  - No!
- States may be backed up
  - many times or not at all
  - in any order
- As long as no state gets starved...
  - convergence properties still hold!!
Asynch VI: Prioritized Sweeping

- Why backup a state if values of successors same?
- Prefer backing a state
  - whose successors had most change

- Priority Queue of (state, expected change in value)
- Backup in the order of priority
- After backing a state update priority queue
  - for all predecessors