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

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')
    - Cost of breathing
    - R is also a Big Table!
- For now, we give this as input to the agent
**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 \( R(s') \)

<table>
<thead>
<tr>
<th>State</th>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_{32} )</td>
<td>-0.01</td>
</tr>
<tr>
<td>( s_{42} )</td>
<td>-1.01</td>
</tr>
<tr>
<td>( s_{43} )</td>
<td>0.99</td>
</tr>
</tbody>
</table>

**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:
  \[
P(S_{t+1} = s'| S_t = s, A_t = a, S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, \ldots, S_0 = s_0) = P(S_{t+1} = s'| S_t = a_t, A_t = a_t)
\]
- This is just like search, where the successor function could only depend on the current state (not the history).

**Optimal 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.

**Example: Racing**

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

<table>
<thead>
<tr>
<th>State</th>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>-0.01</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>-0.03</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>-0.4</td>
</tr>
<tr>
<td>( s_4 )</td>
<td>-2.0</td>
</tr>
</tbody>
</table>

**Policies**

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

Worth Now \( \frac{1}{(1-\gamma)} \)
Worth Next Step \( \frac{1}{1-\gamma} \)
Worth In Two Steps \( \frac{1}{(1-\gamma)^2} \)
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
  - \( U([1,2,3]) = 1*1 + 0.5*2 + 0.25*3 \)
  - \( U([1,2,3]) < U([3,2,1]) \)

Stationary Preferences

- Theorem: if we assume stationary preferences:
  - \([a_1, a_2, \ldots] \succ [b_1, b_2, \ldots] \)
  - \([c, a_1, a_2, \ldots] \succ [c, b_1, b_2, \ldots] \)

- 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: \( U([r_0, r_1, r_2, \ldots]) = r_0 + \gamma r_1 + \gamma^2 r_2 \ldots \)

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? 
  - 10 1 0

- Quiz 2: For \( \gamma = 0.1 \), what is the optimal policy? 
  - 10 1 0

- 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 (\( \gamma \) depends on time left)
  - 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

- Value Iteration
- Policy Iteration
- Reinforcement Learning
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: 
  \[ \pi^*(s) = \text{optimal action from state } s \]

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

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

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 \( \gamma < 1 \)
### 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 $s$

![Diagram](image)

### Computing Time-Limited Values

- **Value Iteration**
  - 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

![Diagram](image)

- **The Bellman Equations**
  - 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

![Diagram](image)
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: $O(S^2A)$
- Number of iterations: $\text{poly}(|S|, |A|, 1/(1-\gamma))$
- Theorem: will converge to unique optimal values

### Iterations

#### k=0

<table>
<thead>
<tr>
<th>State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0.00</td>
</tr>
<tr>
<td>0,1</td>
<td>0.00</td>
</tr>
<tr>
<td>0,2</td>
<td>0.00</td>
</tr>
<tr>
<td>1,0</td>
<td>0.00</td>
</tr>
<tr>
<td>1,1</td>
<td>0.00</td>
</tr>
<tr>
<td>1,2</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

#### k=1

<table>
<thead>
<tr>
<th>State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0.00</td>
</tr>
<tr>
<td>0,1</td>
<td>0.00</td>
</tr>
<tr>
<td>0,2</td>
<td>0.00</td>
</tr>
<tr>
<td>1,0</td>
<td>0.00</td>
</tr>
<tr>
<td>1,1</td>
<td>0.00</td>
</tr>
<tr>
<td>1,2</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

#### k=2

<table>
<thead>
<tr>
<th>State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0.00</td>
</tr>
<tr>
<td>0,1</td>
<td>0.00</td>
</tr>
<tr>
<td>0,2</td>
<td>0.00</td>
</tr>
<tr>
<td>1,0</td>
<td>0.00</td>
</tr>
<tr>
<td>1,1</td>
<td>0.00</td>
</tr>
<tr>
<td>1,2</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

#### k=3

<table>
<thead>
<tr>
<th>State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0.00</td>
</tr>
<tr>
<td>0,1</td>
<td>0.52</td>
</tr>
<tr>
<td>0,2</td>
<td>0.78</td>
</tr>
<tr>
<td>1,0</td>
<td>0.00</td>
</tr>
<tr>
<td>1,1</td>
<td>0.43</td>
</tr>
<tr>
<td>1,2</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

#### k=4

<table>
<thead>
<tr>
<th>State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td>0.37</td>
</tr>
<tr>
<td>0,1</td>
<td>0.66</td>
</tr>
<tr>
<td>0,2</td>
<td>0.83</td>
</tr>
<tr>
<td>1,0</td>
<td>0.00</td>
</tr>
<tr>
<td>1,1</td>
<td>0.51</td>
</tr>
<tr>
<td>1,2</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
### 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)
  \[
  \pi^*(s) = \arg \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] 
  \]
- 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:
- 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!

### 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
  - 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 max difference happens if big reward at $k+1$ level
  - That last layer is at best all $R_{\text{MAX}}$
  - But everything is discounted by $\gamma$ that far out
  - So $V_k$ and $V_{k+1}$ are at most $\gamma^{k+1} |R|$ different
  - So as $k$ increases, the values converge
Problems with Value Iteration

- Value iteration repeats the Bellman updates:
  \[ V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')] \]
- 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

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