Introduction to Artificial Intelligence

Lecture 19: Sequential decision-making and Markov decision processes I

November 10, 2025

Cooperative games

- A cooperative game is typically defined as a pair (N, v), where
  - $N = \{1,2, \dots, n\}$ is the set of players
  - $v: 2^N \to \mathbb{R}$ is the characteristic function, which assigns a value to every coalition

- Applications:
  - **Resource allocation**: given a budget (e.g., one million dollars) and a list of public projects, allocate the budget on a subset of projects to maximize social welfare
  - **Robotics**: multi-agent cooperation

Solution concepts in cooperative games

- **Core**: A payoff vector $x = (x_1, x_2, \dots, x_n)$ is in the core if $\sum_{i \in N} x_i = v(N)$, and $\sum_{i \in S} x_i \geq v(S), \forall S \subseteq N$
  - Example: in resource allocation games, this corresponds to solving a linear program
  - Consider a 3-player game, $N = \{1,2,3\}$. Define the characteristic function as

$$v(S) = \begin{cases} 0, if\ |S| \leq 1 \\ 100, if\ S = \{1,2\} \\ 80, if\ S = \{1,3\} \\ 60, if\ S = \{2,3\} \\ 120, if\ S = \{1,2,3\} \end{cases}$$

- **Shapley value**: ensures fair division https://en.wikipedia.org/wiki/Shapley_value

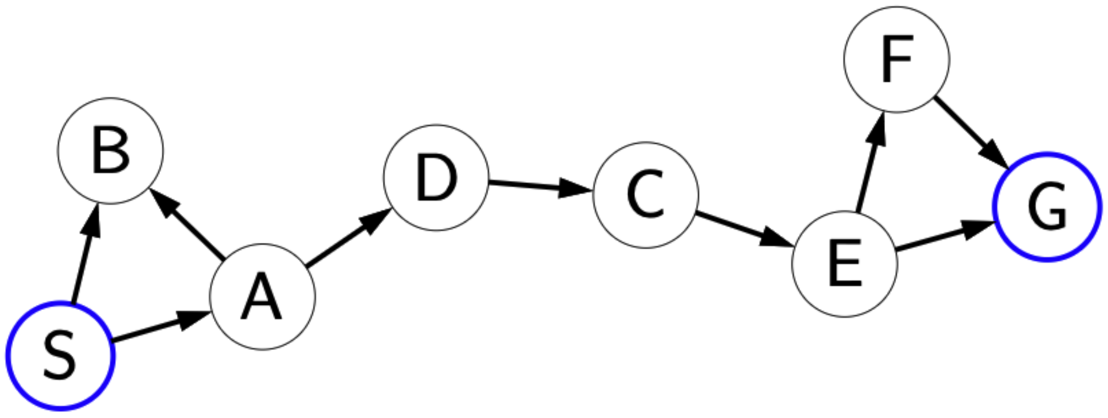
Lecture plan

- Markov Decision Processes
  - **Overview**
  - Modeling
  - Policy evaluation
  - Value iteration

Recap: search problems

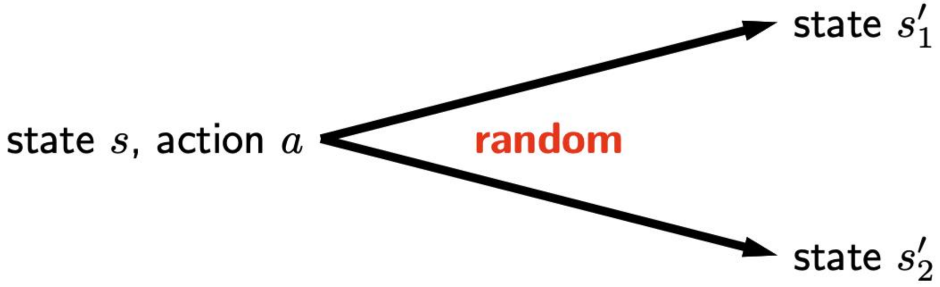
- In previous lectures, we introduced **search** problems, a powerful paradigm that can solve problems ranging from word segmentation to route finding

- However, search problems assume that an action a from states s results deterministically in a unique successor state $Succ(s, a)$



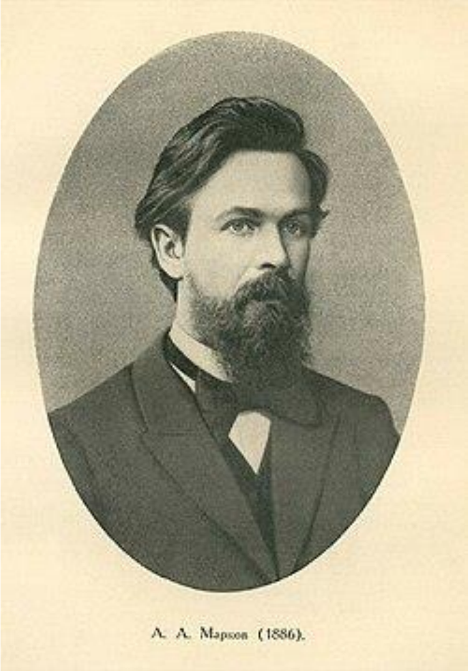
Stochastic search problems

- In real world the deterministic successor is often unrealistic. Taking an action might lead to any one of many possible states
- The question is how we can cope with randomness, *optimally*?
- We will introduce Markov Decision Processes to tackle this



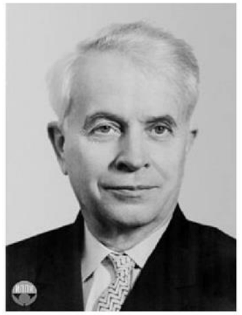
History

- Markov Decision Processes (MDPs) is a mathematical model for decision making under uncertainty
- MDPs, also called a stochastic dynamic program, were first introduced in 1950s-60s
- The term “Markov” refers to Andrey Markov as MDPs are extensions of Markov Chains, and they allow making decisions (taking actions or making choices)



Andrey Markov, 1856 - 1922

Андрей Андреевич Марков (1903-1979)



Lecture Plan

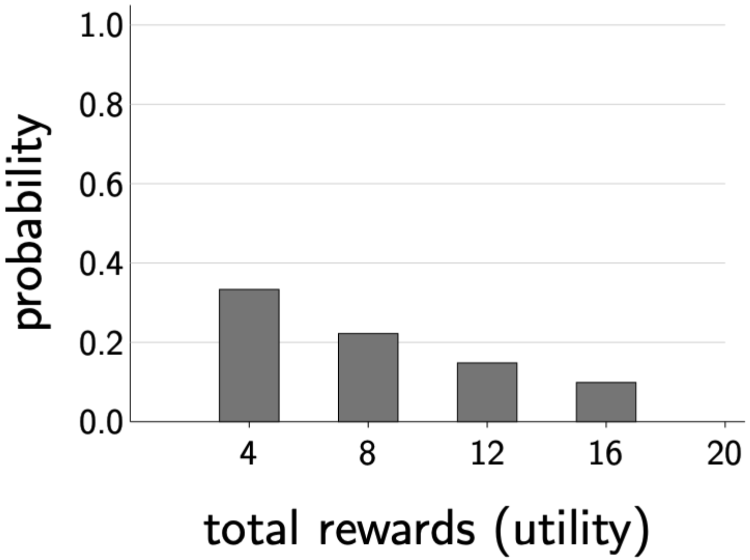
- Markov Decision Processes
  - Overview
  - **Modeling**
  - Policy evaluation
  - Value iteration

Dice game

- For each round $r = 1,2, \ldots$
  - You choose stay or quit
  - If quit, you get \$10 and we end the game
  - If stay, you get \$4 and then Player I rolls a 6-sided dice
    - If the dice results in 1 or 2, we end the game
    - Otherwise, continue to the next round

Expected (discounted) rewards

- If the player follow policy "stay":



- Expected utility:

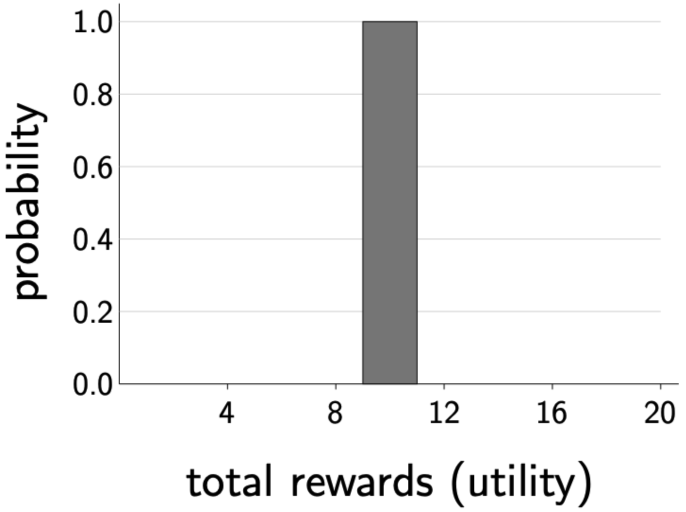
$$\frac{1}{3}(4) + \frac{2}{3} \cdot \frac{1}{3}(8) + \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3}(12) + \cdots = \sum_{i=1}^{\infty} \frac{1}{3} \times \left(\frac{2}{3}\right)^{i-1} \times 4i = 12$$

Expected (discounted) rewards

- Use calculus:
  - Set A as the series
  - Write down $\frac{2}{3}A = \frac{4}{3}\sum_{i=1}^{\infty}\left(\frac{2}{3}\right)^{i} \times\ i$
  - Thus, $A - \frac{2}{3}A = \frac{4}{3}\sum_{i=0}^{\infty}\left(\frac{2}{3}\right)^{i} = \frac{4}{3} \times \frac{1}{1-\frac{2}{3}} = 4$

Instantaneous reward

- If the player follow policy "quit":

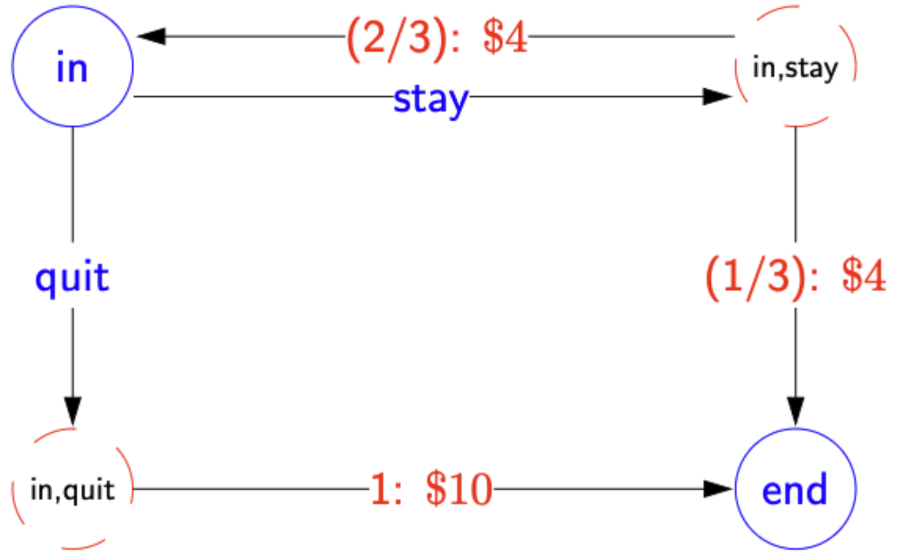


- (Expected) utility:

$$1 \times (10) = 10$$

MDP for dice game

- Formalize the dice game as an MDP (Markov decision process)
- An MDP can be represented as a graph
  - The nodes in this graph include both **states** and **chance nodes**
  - Edges coming out of a chance nodes are the possible outcomes of that action
  - Label these chance-to-state edges with the probability of a particular **transition** and the associated reward for traversing that edge



Definition of an MDP

- Markov decision process
  - $States$: the set of states
  - $s_{start} \in States$: starting state
  - $Actions(s)$: possible actions from state s
  - $T(s, a, s')$: probability of s' if action a is taken in state s
  - $Reward(s, a, s')$: reward for the transition (s, a, s')
  - $IsEnd(s)$: whether the process has reached the end of the game
  - $0 \leq \gamma \leq 1$: time discount factor (default: 1)

Transitions

- **Definition** (transition probabilities): The transition probabilities $T(s, a, s')$ specify the probability of being in state s' if action a is taken in state s
  - For each state s and action a, the transition probabilities specifies a distribution over successor states s'
  - This means that each given s and a, if we sum up the transition probability $T(s, a, s')$ over all possible successor states s', we get 1

s	a	s'	$T(s, a, s')$
in	quit	end	1
in	stay	in	$\frac{2}{3}$
in	stay	end	$\frac{1}{3}$

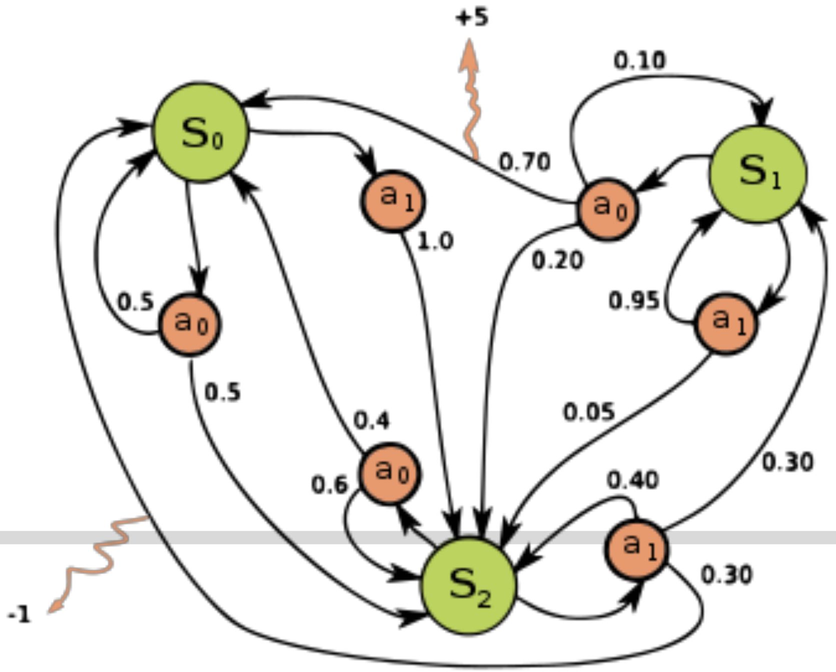
Transition probabilities

- If a transition to a particular s' is not possible, then $T(s,a,s') = 0$. We refer to the s' for which $T(s,a,s') > 0$ as the successors
- Generally, the number of successors of a given (s,a) is much smaller than the total number of states
  - In a search problem, each (s,a) has exactly one successor

s	a	s'	$T(s,a,s')$
in	quit	end	1
in	stay	in	$\frac{2}{3}$
in	stay	end	$\frac{1}{3}$

Policy

- **Search problem**: path (sequence of actions)
- **Policy**: specify what the agent should do for any state that the agent might reach
  - A policy π is a mapping from each state $s \in States$ to an action $a \in Actions(s)$
  - An illustrative example
- **Optimal policy**: a policy that yields the highest expected utility



Lecture Plan

- Markov Decision Processes
  - Overview
  - Modeling
  - **Policy evaluation**
  - Value iteration

Evaluating a policy

- Definition: **utility function**
  - Following a policy yields a **random** path
  - The utility of a policy is the (time-discounted) sum of the rewards on the solution path (which is a random variable)

Path (dice game)	Utility
[in; stay, 4, end]	4
[in; stay, 4, in; stay, 4, in; stay, 4, end]	12
[in; stay, 4, in; stay, 4, end]	8
[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]	16
...	...

- Definition: **value** (**expected utility**)
  - The value of a policy at a state is the **expected utility**

Time-discounted utility function

- While there's a fact that a reward today might be worth more than the same reward tomorrow, so we introduce a new aspect: **discounting**
- Definition: **utility function**
  - Path: $s_0, a_1 r_1 s_1, a_2 r_2 s_2, \ldots$ (action, reward, new state)
  - The utility with discount γ is

$$u_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \cdots$$

  - Note: this type of time-discounted reward function is also used in reinforcement learning. The point is to emphasize immediate payoffs and differentiate that from future payoffs

Examples of time discount factors

- Discount $\gamma = 1$ (save for the future):
  - $[stay, stay, stay, stay]$: $4 + 4 + 4 + 4 = 16$
- Discount $\gamma = 0$ (live in the moment):
  - $[stay, stay, stay, stay]$: $4 + 0 + 0 + \cdots = 4$
- Discount $\gamma = 0.5$ (balanced life):
  - $[stay, stay, stay, stay]$: $4 + \frac{1}{2} \cdot 4 + \frac{1}{4} \cdot 4 + \frac{1}{8} \cdot 4 = 7.5$

Policy evaluation

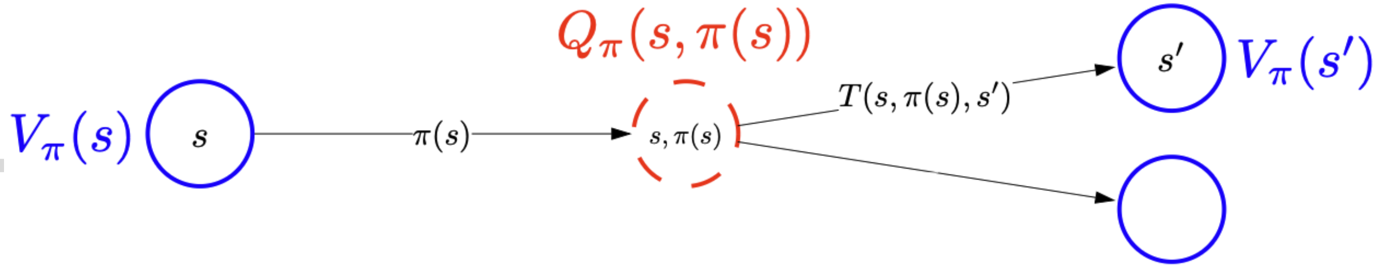
- Definition:
  - **Value of a policy**: Let $V_\pi(s)$ be the expected utility received by following policy π from state s
  - ***Q*-value of a policy**: Let $Q_\pi(s,a)$ be the expected utility of taking action a from state s and then following policy π

$$V_\pi(s) = \begin{cases} 0, & \text{if IsEnd}(s) \\ Q_\pi(s, \pi(s)), & \text{otherwise} \end{cases}$$

  - Then, we have the following identity

$$Q_\pi(s,a) = \sum_{s'} T(s,a,s') \cdot \big(Reward(s,a,s') + \gamma \cdot V_\pi(s')\big)$$

  - An illustration:



Back to the dice game

- Let π be the "stay" policy: $\pi(in) = stay, V_\pi(end) = 0$. Then we have

$$V_\pi(in) = \frac{1}{3}\big(4 + V_\pi(end)\big) + \frac{2}{3}\big(4 + V_\pi(in)\big)$$

- In this case, we can solve this linear system and get a closed-form solution:

$$V_\pi(in) = 12$$

Policy evaluation

- Key idea: Start with arbitrary policy values and repeatedly apply recurrences to converge to true values

- **Policy evaluation**
  - Initialize $V_\pi^{(0)}(s) := 0$ for all states s
  - For iteration $t = 1, \ldots, T$:
    - For each state s:

$$V_\pi^{(t)}(s) := \sum_{s'} T(s, \pi(s), s')[Reward(s, \pi(s), s') + \gamma V_\pi^{(t-1)}(s')]$$

Policy evaluation implementation

- How many iterations T? Repeat until values don't change much:

$$\max_{s \in States} |V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s)| \leq \epsilon$$

- Memory optimization: store only last two values

$$V_\pi^{(t)}, V_\pi^{(t-1)}$$

- Time complexity (can be further optimized for sparse transition models):

$$O(T \cdot |S|^2 \cdot |A|)$$

Dice game

- Let π be the "stay" policy: $\pi(in) = stay, V_\pi^{(t)}(end) = 0$

$$V_\pi^{(t)}(in) = \frac{1}{3}\left(4 + V_\pi^{(t-1)}(end)\right) + \frac{2}{3}\left(4 + V_\pi^{(t-1)}(in)\right)$$

- Converges to $V_\pi(in) = 12$ in 100 iterations

Lecture Plan

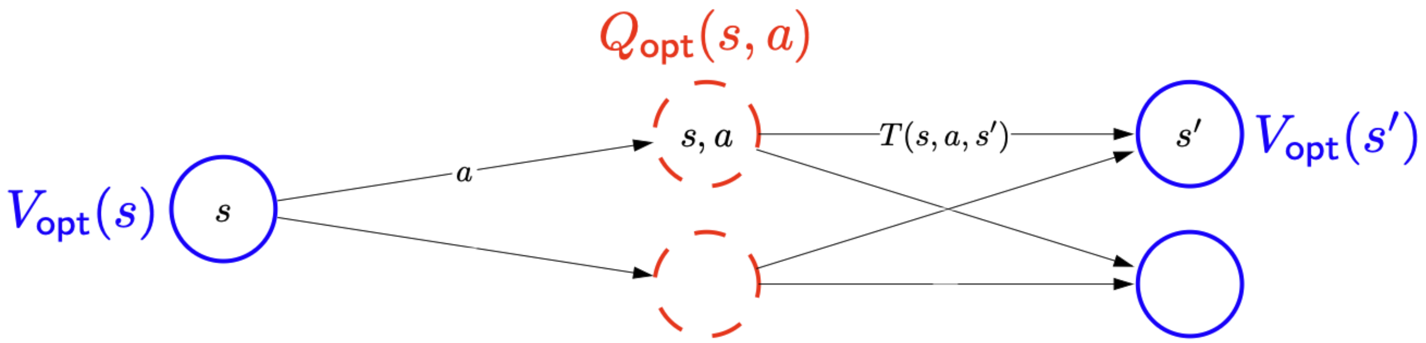
- Markov Decision Processes
  - Overview
  - Modeling
  - Policy evaluation
  - **Value iteration**

Optimal value and policy

- Definition: The optimal value $V_{opt}(s)$ is the maximum value attained by any policy

- Optimal value if action a is taken in state s:

$$Q_{opt}(s,a) = \sum_{s'} T(s,a,s')[Reward(s,a,s') + \gamma V_{opt}(s')]$$



Optimal values and Q-values

- Optimal value from state s:

$$V_{opt}(s) = \begin{cases} 0 \\ \max_{a \in Actions(s)} Q_{opt}(s,a) \end{cases}$$

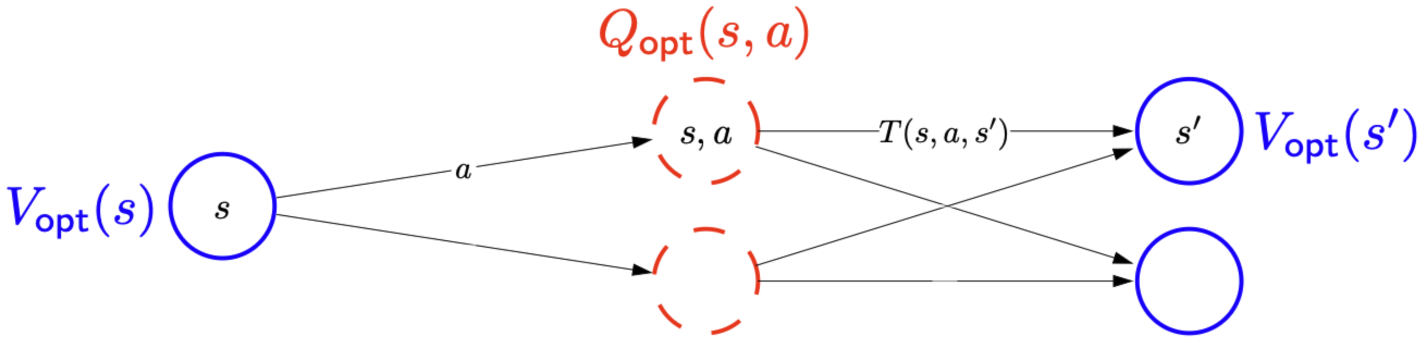
- Bellman equation (1957), after Richard Bellman: the utility of a state is the expected reward for the next transition plus the discounted utility of the next state, assuming that the agent chooses the optimal action

Richard Ernest Bellman

The value iteration algorithm

- Algorithm: value iteration [Bellman equation, 1957]
  - Initialize $V_\pi^{(0)}(s) \coloneqq 0$ for all states s
  - For iteration $t = 1, \ldots, T$:
    - For each state s:

$$V_{opt}^{(t)}(s) \coloneqq \max_{a \in \mathrm{Actions}(s)} \sum_{s'} T(s,a,s')\left[Reward(s,a,s') + \gamma V_{opt}^{(t-1)}(s')\right]$$



Time complexity of the value iteration algorithm

- **Time complexity**:
  - One outer loop runs for T iterations
  - In each iterations, we update every state s, giving a factor $|S|$
  - For each state we take a max over all actions, and for each (s, a) we sum over possible next states, giving factors $|A|$ and $|S'|$
- The overall time complexity is $O(T \cdot |S|^2 \cdot |A|)$

Convergence rate

- Convergence rate: for a discounted, finite MDP with $\gamma \in [0,1)$, value iteration is a γ-contraction in the max norm
$$\|v_{k+1} - v^\star\|_\infty \leq \gamma\|v_k - v^\star\|$$

- Thus, the error decays geometrically by a rate of γ^T

- To guarantee ε error to $v^\star$, need $T = \frac{\log(\epsilon^{-1})}{1-\gamma}$ iterations

Value iteration for the dice game

- Initially, the optimal policy is "quit"
- But as we run value iteration longer, it switches to "stay"

s	end	in	
$V_{\text{opt}}^{(t)}$	0.00	12.00	($t = 100$ iterations)
$\pi_{\text{opt}}(s)$	-	stay	

- Recall that stay yields an expected reward is 12
- Quit yields an instantaneous reward of 10

Recap

- **Theorem**: convergence rate for policy iteration
  - Suppose the discount factor $\gamma < 1$, or MDP graph is acyclic
  - Then value iteration converges to the optimal policy value

- **Counterexample:** if we have a cyclic graph and $\gamma = 1$, value function will terminate immediately with the same value, because all paths are of infinite length

Summary

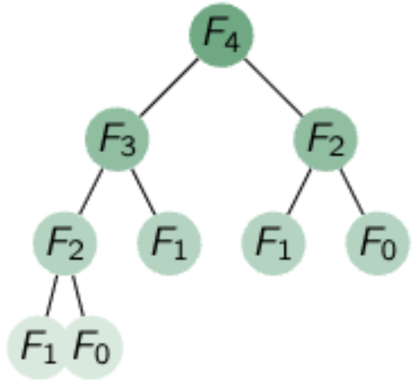
- Markov Decision Processes (MDPs) are a type of models for coping with uncertainty
- Solutions are represented by policies rather than solution paths
- Policy evaluation: $(MDP, \pi) \rightarrow V_{pi}$
- Value iteration: $MDP \rightarrow (Q_{opt}, \pi_{opt})$
- Algorithms:
  - Policy evaluation computes policy value $V_{\pi}(s)$
  - Value iteration computes optimal value $V_{opt}(s)$

Dynamic programming: Learning from subproblems

- **The value iteration algorithm is part of a broader computing paradigm called dynamic programming**
- Core insight: Solve each subproblem once and store the result for future use
  - **Optimal substructure:** Optimal solution contains optimal solutions to subproblems
  - **Overlapping Subproblems:** Same subproblems appear multiple times

Example: Computing Fibonacci numbers

- The Fibonacci sequence F_0, F_1, F_2, F_3, … is the sequence of numbers defined by
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$
- Question: given n, compute F_n
- Dynamic programming vs. brute-force search




```
def fib_dp(n):
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]

# Time: O(n), Space: O(n)
```

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

# Time: O(2^n)
# Recalculates same values!
```

Example: Value iteration network

- AlphaGo's value network uses a form of value iteration to evaluate board positions. It learns a value function $V(s)$ that corresponds to the probability of winning from state s

```
def minimax_dp(state, depth, memo={}):
    if state in memo:
        return memo[state]  # DP: reuse computed values

    if is_terminal(state) or depth == 0:
        return evaluate(state)

    if maximizing_player:
        value = max(minimax_dp(child, depth-1) for child in children(state)
    else:
        value = min(minimax_dp(child, depth-1) for child in children(state)

    memo[state] = value  # Store for future use
    return value
```

Other use cases

- Dynamic programming is good when
  ✓ The problem can be defined in terms of subproblems
  ✓ The optimal solution use optimal subproblems, and this can be specified by a recurrence relation

- **Applications**
  - **Shortest path:** Bellman-Ford
  - **Sequence alignment:** Edit distance, DNA matching
  - **Optimization:** Knapsack, scheduling
  - **Counting:** Number of ways to reach goal