Reinforcement Learning:
Dynamic Programming and Monte-Carlo Methods
The Bellman Equations

Andrei Cozma and Landon Harris

Test Questions

1. What does the “model” refer to in the terms “model-based” and “model-free”?

2. What are some of the limitations of Dynamic Programming methods?

3. How can you handle the exploration/exploitation trade-off in Monte Carlo methods?

Presenters – Andrei Cozma

Program: Computer Science M.S. (maybe PhD)

- Intelligent Systems & Machine Learning
- Advisor: Dr. Hairong Qi
- AICIP Lab Group

Undergraduate Degree:

- Computer Science B.S., May 2022
- Minors: Cybersecurity & Business Administration

Interests

- Machine Learning & Deep Learning
  - Computer Vision
  - Natural Language Processing
  - Reinforcement Learning
- Signal & Information Processing

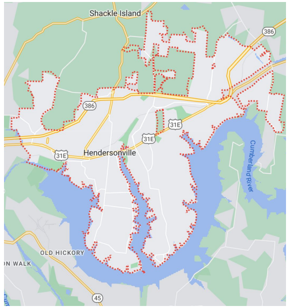
Goals

- Publish some papers summer or fall
- Build a strong network of connections

Presenters – Andrei Cozma

Deva, Romania

Hendersonville, TN

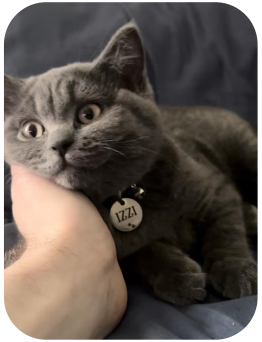
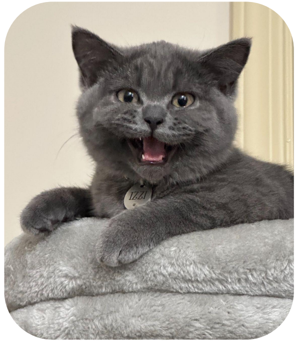
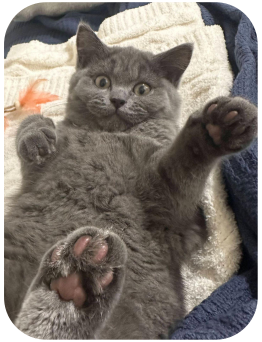


Presenters – Andrei Cozma

- Traveling
- Exploring
- Photography
- Music & Concerts
  - Rock, Alternative, Blues, etc etc.
- Ice Skating, Biking, Swimming





Presenters – Landon Harris

Program: Computer Science M.S.

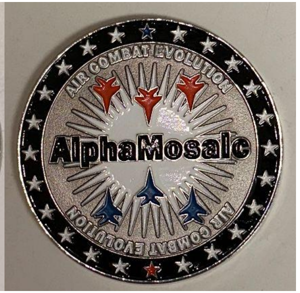
- Advisor: Dr. Hairong Qi
- AICIP Lab Group

Undergraduate Degree:

- Computer Science B.S., May 2022
- Minors: Mathematics

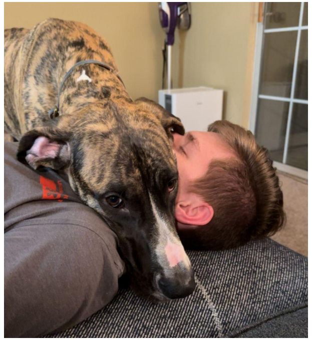
Interests

- Machine Learning & Deep Learning
  - Computer Vision
  - Reinforcement Learning
- Foundational AI
  - Dynamical Systems analysis
  - Training Policies / Continual Learning



Presenters – Landon Harris

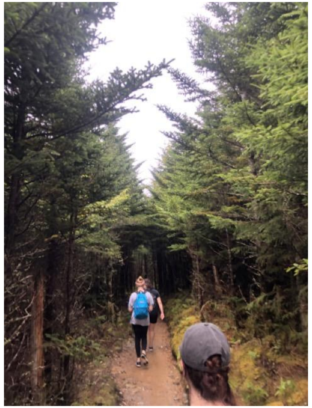
- Hometown: Franklin, TN



Presenters – Landon Harris

- Traveling
- Hiking
- Music







Outline

1. Terms & Overview
2. History
3. Background
4. Algorithms
    - Dynamic Programming
    - Monte Carlo
5. Applications
6. Implementation
7. Open Issues
8. Discussion

Overview – Part 1

Reinforcement Learning (RL)

- Subfield of Machine Learning
- Agents learn optimal actions through interaction with an environment.

Markov Decision Process (MDP)

- Mathematical framework for modeling decision-making processes in stochastic, discrete-time, and finite-state environments
- <u>Markov Property</u>: future states depend only on the current state, not on the sequence of past states.

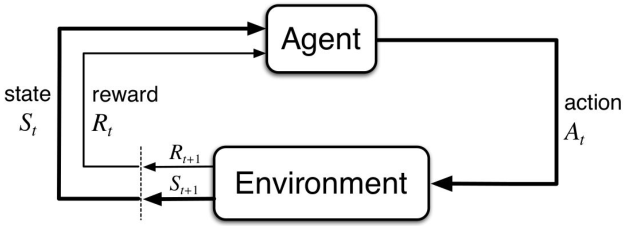
Model-Based Methods

- Methods that use knowledge of the environment's dynamics

Model-Free Methods

- Learn directly from experience, no model required

(more on these later)



Agent-environment interaction in a MDP

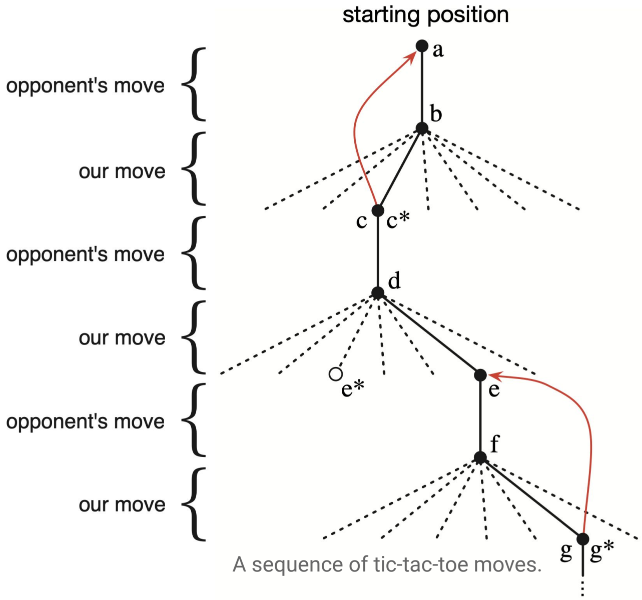
Applications:

- Robotics
- Game AI
- Recommendation Systems
- Finance, Healthcare, etc

Overview - Part #2

Policy (π)

- Defining the agent's behavior: mapping from state to action
- Stochastic:
  - **π(a|s)** is the probability of choosing action a while in state s
- Deterministic:
  - **π(s)** is the action taken while in state s



A sequence of tic-tac-toe moves.

Return (G)

- We try to maximize the sum of all future rewards
  - This holds for episodic/terminal (finite duration) problems

$$G_t = \sum_{\{t=t+1\}}^{T} R_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{\{i=0\}}^{\infty} \gamma^i R_{t+i+1}$$

- Discounted Returns:
  - For non-episodic problems, exploit the concept of discounting
  - Gamma, 0 < γ < 1, is the discount rate

recursive form: $G_t = R_{t+1} + \gamma G_{t+1}$

Overview - Part #3

Almost all Reinforcement Learning algorithms are based on estimating Value Functions

- **Expected Return -> Value Functions**
  - Value of a state is the expected return when starting in that state and then following policy π thereafter
  - Can only be calculated given a policy π

$$V(s) = E\left[G_t \mid S_t = s\right]$$

- **State-Value Function**
  - *How good is it to be in a given state?*
  - Maintain an average of actual returns for each state.
  - Avg. will converge to the state's value.

$$V^{\pi}(s) = E_{\pi}\left[\mathrm{G_t} \mid \mathrm{S_t} = \mathrm{s}\right] = \mathrm{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right], \forall s \in \mathcal{S}$$

- **Action-Value Function**
  - *How good is it to take a specific action while in a given state?*
  - Maintain separate averages for each action taken in a state
  - Avgs. will converge to the state-action values.

$$Q^{\pi}(s,a) = E_{\pi}\left[\mathrm{G_t} \mid \mathrm{S_t} = \mathrm{s}, A_t = a\right] = \mathrm{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right], \forall s \in \mathcal{S}$$

History

Early Beginnings: Pioneering Ideas (1950s-1960s)

- Bellman - Introduced Dynamic Programming, Optimal Control, Markov Decision Processes (MDPs), and Value Functions (1957)
- Samuel - Developed a checkers-playing program that learned value functions through approximation (1959)
- Howard - Proposed the Policy Iteration method for solving MDPs (1960)
- Minsky - Established the connection between DP and RL. Discussed Trial-and-Error Learning, Prediction, and Expectation. (1961)
- Michie, Chambers - Early use of Monte Carlo methods in RL (1968)

Golden Age of Theoretical Foundations (1970s-1980s)

- Witten - Published the earliest work on a Temporal-Difference (TD) Learning rule (1977)
- Barto, Sutton, Anderson - Developed the Actor-Critic Architecture for combining TD learning with trial-and-error learning (1981)
- Narendra, Wheeler - Introduced the Every-Visit Monte Carlo method (1986)
- Sutton - Separated TD learning from control, introduced the TD(λ) algorithm, and proved some of its convergence properties (1988)
- Watkins - Developed Q-Learning, which extended and integrated prior work in RL research (1989)

History

Model-Based and Model-Free Era (1990s-2000s)

- Tesauro - Developed TD-Gammon, a successful Backgammon player using TD-Learning (1992)
- Williams - Policy Gradient methods gain popularity, optimizing policy directly (1992)
- Rummery, Niranjan - SARSA: On-policy TD Control (1994)
- John - Expected SARSA: Off-policy TD Control (1994)
- Bradtke, Barto - Developments on Sample-Based Learning with Monte Carlo methods (1996)
- Sutton, Barto - Popular RL book with a comprehensive overview of Model-Free and Model-Based methods (1998)

Revolutionary Breakthroughs (2010s-Present)

- DeepMind - Deep Q-Network (DQN) capable of learning from high-dimensional sensory inputs (2013)
- DeepMind - DQN demonstrates human-level performance in Atari games (2015)
- DeepMind - AlphaGo, combines Deep Neural Networks and Monte Carlo Tree Search (MCTS), defeats world champion (2016)
- OpenAI - Proximal Policy Optimization (PPO) sample-efficiency and balances exploration and exploitation (2017)
- DeepMind - AlphaStar achieves grandmaster level in StarCraft II (2019)

Algorithms Intro – The Bellman Equations

From the definitions of Value Function: $V(s) = E\left[G_t \mid S_t = s\right]$ and Return: $\mathrm{G_t} = R_{t+1} + \gamma \mathrm{G_{t+1}}$

We can derive: $V^\pi(s) = E_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right]$

State-Value Function

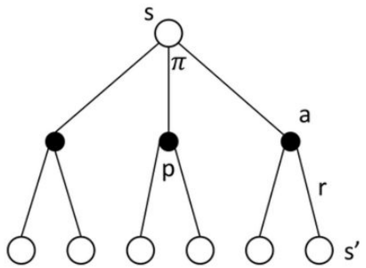
$$V^\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma V^\pi\left(s'\right)\right] =$$

$$= \sum_a \pi(a \mid s) Q^\pi(s, a)$$

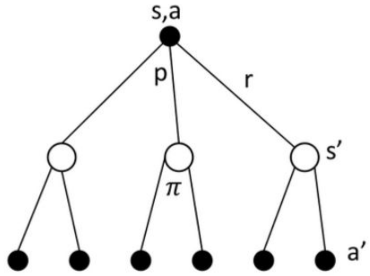
Action-Value Function

$$Q^\pi(s, a) = \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma \sum_{a'} \pi\left(a' \mid s'\right) Q^\pi\left(s', a'\right)\right] =$$

$$= \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma V^\pi\left(s'\right)\right]$$



Backup Diagrams



Algorithms Intro – Optimal Policy & Value Functions

1. A **policy** π* is defined to be better than or equal to a **policy** π if
   for all states, its expected return is greater than or equal to that of π

$$\pi^* \geq \pi \quad \Leftrightarrow \quad V^{\pi^*}(s) \geq V^{\pi}(s) \quad \forall s \in S$$

2. There is always at least one policy (a.k.a optimal policy) that is better than or equal to all other policies

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad \forall s \in S$$

$$V^*(s) = \max_{a} E\left[r_{t+1} + \gamma V^*\left(s_{t+1}\right) \mid S_t = s, A_t = a\right] = \max_{a} \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma V^*\left(s'\right)\right]$$

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \quad \forall s \in S, a \in A(s)$$

$$Q^*(s,a) = \mathrm{E}\left[\mathrm{r}_{t+1} + \gamma V^*\left(s_{t+1}\right) \mid s_t = s, a_t = a\right] = \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma \max_{a'} Q^*\left(s', a'\right)\right]$$

Intuitively, it expresses the fact that the value of a state under an optimal policy
must equal *the expected return for the best action from that state.*

Algorithms Intro - Bellman Optimality Equations

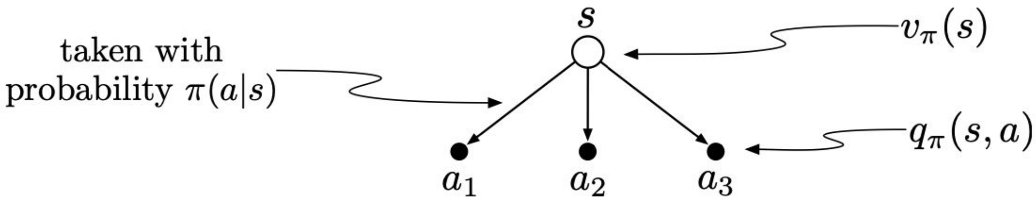
1. Since **V*(s)** and **Q*(s, a)** are value functions for a policy, they must satisfy the Bellman Equation.
   a. Any policy which is greedy with respect to the optimal value function, is an optimal policy.
   b. <u>Principle of Optimality</u>: “An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.” (Bellman, 1957)

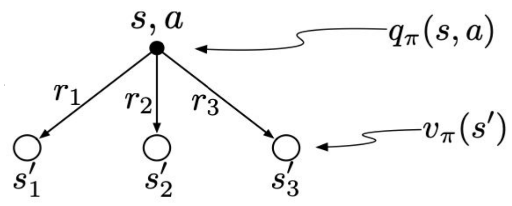
2. Given **V*(s)**, one-step-ahead search produces the long-term optimal actions
   a. This is a deterministic policy

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}} \sum_{s', r} p(s', r \mid s, a)\left[r + \gamma V^*(s')\right]$$

3. Given **Q*(s, a)**, the agent does not even need to do a one-step-ahead search

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}} Q^*(s, a)$$





Algorithms - Dynamic Programming (DP)

A theoretically optimal method to compute an optimal policy π*

- Done by solving the Bellman Optimality Equations

Requires:

- An accurate model of the environment (e.g. transition probabilities)
- The environment maintains the Markov property
- Enough space & time to perform the computation

Usually have to settle for approximation, offering several advantages:

- learn more effectively
- feature extraction can reduce noise
- can address relatively large scale problems

Algorithms - DP - Policy Evaluation

Task: finding the state-value function

Given: the current policy

Steps:

1. Choose an arbitrary V_0
2. At each step you use the update rule:

$$V_{k+1}^{\pi}(s) = \sum_{a} \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \alpha V_k^{\pi}\left(s'\right)\right]$$

$$V_0 \to V_1 \to V_2 \to \cdots \to V_k \to V_{k+1} \to \cdots \to V^{\pi}$$

3. Can be proven to converge to V^{π} as $k \to \infty$
4. In each iteration all states are updated
   a. scheme can be computationally heavy
5. When to stop?
   a. Since we cannot do infinite iterations, do until largest update is smaller then some threshold

$$\max s \in S \left|V_{k+1}(s) - V_k(s)\right|$$

Algorithms – DP – Policy Improvement

Task: Update the policy **Given**: the current state-value function

Steps:

1. Suppose we've computed the state-value function (using Policy Evaluation) for some policy π
2. Let π' be a deterministic policy that:
    - Selects an action a for each state s to maximize the first-step value

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V^\pi(s')\right]$$

3. What is the value if we first choose action a which is not necessarily $\pi(s)$?

$$Q^\pi(s, a) = \sum_{s',r} p(s', r \mid s, a)\left[r + \alpha V^\pi(s')\right]$$

    - If this is higher than $V^\pi(s)$, then it should be better to select action a and then follow π.
    - Should we change the policy? Yes
4. Policy Improvement Theorem:
    - if: $Q^\pi(s, \pi'(s)) \geq \mathrm{V}^\pi(s) \quad \forall s \in S$
    - then: $\mathrm{V}^{\pi'}(s) \geq \mathrm{V}^\pi(s)$

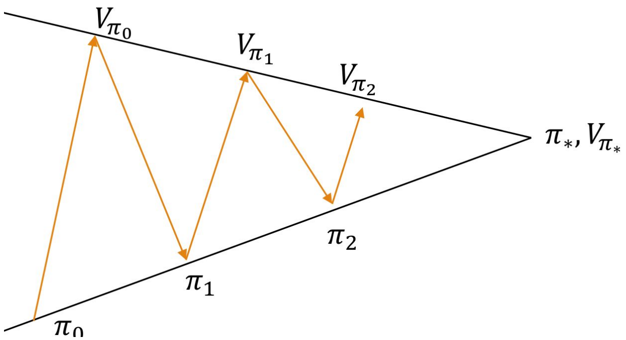
Algorithms – DP – Policy Iteration

Technique for obtaining the optimal policy

- Two complementary steps:
  a. Policy Evaluation: Finding the value function under a given policy
     - Updating the value function makes the policy not greedy anymore
  b. Policy Improvement: Updating the policy given the current value function
     - Finding a greedy policy for V^{π}_{i} makes the value function incorrect

Yet together they drive each other to the optimal solution: π^* , V^{π^*}

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} V_{\pi_*}$$



- It can be shown that π' is at least as good as π
  a. if they are equal they are both the optimal policy.

Algorithms – DP – Value Iteration

Policy Evaluation

$$V_{k+1}^{\pi}(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)\left[r + \alpha V_k^{\pi}(s')\right]$$

$$V_0 \to V_1 \to V_2 \to \cdots \to V_k \to V_{k+1} \to \cdots \to V^{\pi}$$

Policy Improvement

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V^{\pi}(s')\right]$$

Policy Iteration

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \ldots \xrightarrow{I} \pi_* \xrightarrow{E} V_{\pi_*}$$

Value Iteration

$$V_{k+1}(s) = \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \alpha V_k(s')\right]$$

Algorithms – DP – Conclusions

Key Takeaways

- Requires a complete model of the environment to compute state values (model-based)
  - State-transition probabilities
- Uses Bellman Equations to compute state-values for each time step, until convergence
- May not be practical for large problems
  - Today, we can get away with millions of states
- Dynamic programming is still more efficient than exhaustive search
  - DP is guaranteed to find the optimal policy in less than some polynomial combination of |A| and |S|
  - Exhaustive search is $|A|^{|S|}$

Algorithms - Monte Carlo (MC) Methods

A complete model of the environment is not always available.

<u>Learning through experience</u> (sample-based)

- Online interaction with an environment
  - do *not* require prior knowledge (state transition probabilities)
- Simulated interactions
  - *technically* require a kind of model of the environment, but *not* probability distributions as required for DP
  - sometimes it’s easier to only get samples!

Typically applied to <u>episodic tasks</u> (incremental in an episode-by-episode manner)

- In contrast, DP method focused on one-step transitions (step-by-step scheme)
- If task is not episodic, make it episodic (e.g. forced termination by max number of steps)

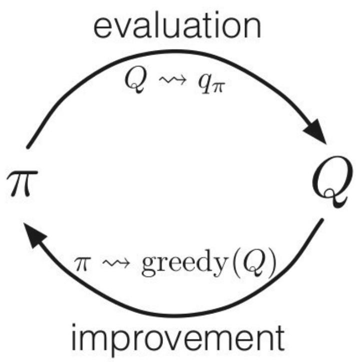
Algorithms – Monte Carlo – Control

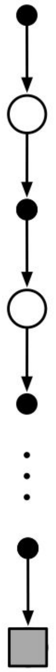
Generate an episode that follows policy π:

$$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T$$

Policy Iteration

- Policy Evaluation
  - averaging sample returns over many outcomes to update value function
- Policy Improvement
  - selecting greedy actions to get the new policy
- Alternating between Evaluation and Improvement *each episode*.
  - Only calculating returns and new policy *at the end of an episode*

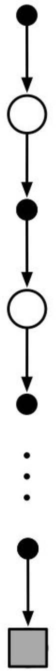




Converge to optimal policy over infinitely many episodes

- In practice, update only to a threshold desired level of performance (same as DP)

Algorithms - MC - Policy Evaluation



Generate an episode that follows policy π:

$$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T$$

Estimating $V^\pi(s)$ from experience

- Simply average many returns observed after visiting a certain state s
  - After each occurrence of state s in an episode
- A state's value is the expected return
  - So, this average can become a good approximation to the expected value

Two Types:

- First Visit: averaging only the returns following *first* visits to state s
- Every Visit: averaging returns following *all* the visits to state s

  Use the same MC approach. The agent records the rewards received after taking action a at state s

Algorithms – MC – Policy Improvement

So we have discussed a way find $V^{\pi}(s)$

Problem?

- We do not have the model of the environment (transition probabilities)
- Thus cannot easily do policy improvement (finding the new greedy policy π)

$$\pi'(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \alpha V^{\pi}(s')\right]$$

- Instead we should estimate $Q^{\pi}(s, a)$ directly:

$$\pi'(s) = \underset{a \in A}{\operatorname{argmax}} Q^{\pi}(s, a)$$

$$\pi_0 \xrightarrow{E} Q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} Q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} Q_{\pi_*}$$

Algorithms - MC - Exploration Trade-Off

$$\pi_0 \xrightarrow{E} Q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} Q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} Q_{\pi_*}$$

Now we have another problem:

- Need to force exploration or some actions will never be chosen.
- How can we fix this?

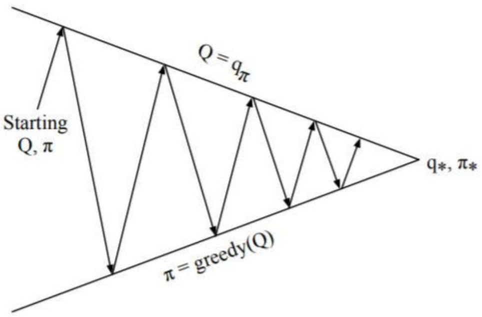
$$\pi'(s) = \underset{a \in A}{\text{argmax}} Q^\pi(s,a)$$

Some ways…

- Exploring starts (MC-ES)
  - Each state-action pair at the beginning of an episode has a non zero probability.
    - But cannot always be done.
- Soft policies (e.g.: ϵ-greedy)
  - Most of the time choose an action that has maximal estimated action value
  - With probability epsilon instead select an action at random
- Off-Policy learning

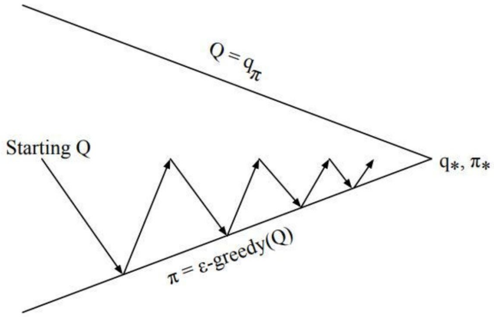
Algorithms - MC - On-Policy vs Off-Policy

Greedy Policy



$$\pi'(s) = \underset{a \in A}{\operatorname{argmax}} Q^{\pi}(s, a)$$

Soft Policy (ϵ-greedy)



$$\pi(a \mid s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(s)|} & \text{if } a^* = \underset{a \in A}{\operatorname{argmax}} Q^{\pi}(s, a) \\ \frac{\varepsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases}$$

On-policy: Policy being evaluated and improved is also used to generate episode (e.g.: ϵ-greedy)

Off-policy: Separate *target policy* (greedy) and *behavior policy* (ϵ-greedy)

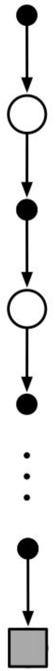
Algorithms – MC – Conclusions

Key Takeaways

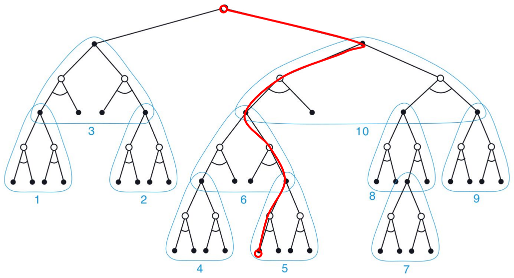
- Can learn directly from interaction with the environment
    - No need for full models
- No need to learn about all states
    - Computational complexity of updating one node is independent of $|S|$
    - So, optimal policy trajectory corresponds to a small state subset
    - Even if the environment's dynamics are known, MC is often more efficient than DP

One issue to watch out for: maintaining sufficient exploration

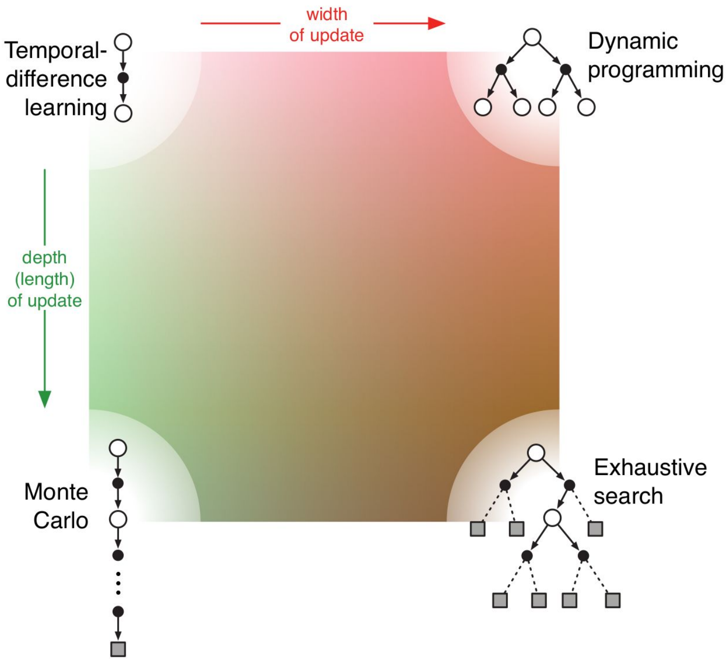
- Exploring starts, Soft policies.
- Off-Policy Methods

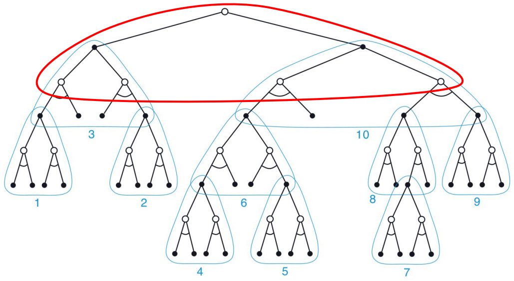


Algorithms – DP vs. MC Backup Diagrams



MC Update





DP Update

Applications

Robotics

- Robotic grasping and manipulation tasks
- Navigation in complex dynamic environments
- Autonomous driving, Industrial automation

Game AI

- Playing complex games like chess, Go, and poker
- Game AI for non-player characters (NPCs)
- Procedural content generation

Recommendation systems

- Personalized content recommendation
- Improving search engine results
- Dynamic pricing and demand forecasting

Finance

- Trading and portfolio optimization
- Fraud detection and prevention
- Risk management and forecasting

Healthcare

- Drug discovery and development
- Patient diagnosis and treatment planning
- Personalized medicine and treatment recommendations

Implementation - Dynamic Programming

```python
def train(self):
    while True:
        delta = 0
        V_prev = np.copy(self.V)
        # Update the value of each state. Make a copy so we don't get in self-loops
        for state in range(self.env.observation_space.n):
            # calculate the action-value for each possible action
            Q = np.zeros(self.env.action_space.n)
            for action in range(self.env.action_space.n):
                expected_value = 0
                # Loop through all possible resulting states from this action
                for probability, next_state, reward, done in self.env.P[state][action]:
                    expected_value += probability * (reward + self.gamma * self.V[next_state])
                Q[action] = expected_value
            action, value = np.argmax(Q), np.max(Q)
            # update the state-value function with the best performing action's value
            self.V[state] = value
            delta = max(delta, abs(V_prev[state] - self.V[state]))
        if delta < self.theta:
            break
```

Example Q-Table:

Resulting Policy:

Implementation – Monte Carlo

```python
def initialize(self):
    # The Q-Table holds the current expected return for each state-action pair
    self.Q = np.zeros((self.n_states, self.n_actions))
    # R keeps track of all the returns that have been observed for each state-action pair to update Q
    self.R = [[[] for _ in range(self.n_actions)] for _ in range(self.n_states)]
    # An arbitrary e-greedy policy:
    # With probability epsilon, sample an action uniformly at random
    self.Pi = np.full(
        (self.n_states, self.n_actions), self.epsilon / self.n_actions
    )
    # For the initial policy, we randomly select a greedy action for each state
    self.Pi[
        np.arange(self.n_states),
        np.random.randint(self.n_actions, size=self.n_states),
    ] = (
        1 - self.epsilon + self.epsilon / self.n_actions
    )
```

```python
def update_first_visit(self, episode_hist):
    G = 0
    # For each step of the episode, in reverse order
    for t in range(len(episode_hist) - 1, -1, -1):
        state, action, reward = episode_hist[t]
        # Updating the expected return
        G = self.gamma * G + reward
        # First-visit MC method:
        # Updating the expected return and policy only if this is the first visit to this state-action pair
        if (state, action) not in [(x[0], x[1]) for x in episode_hist[:t]]:
            self.R[state][action].append(G)
            self.Q[state, action] = np.mean(self.R[state][action])
            # Updating the epsilon-greedy policy.
            # With probability epsilon, sample an action uniformly at random
            self.Pi[state] = np.full(self.n_actions, self.epsilon / self.n_actions)
            # With probability 1-epsilon, select the greedy action
            self.Pi[state, np.argmax(self.Q[state])] = (
                1 - self.epsilon + self.epsilon / self.n_actions
            )
```

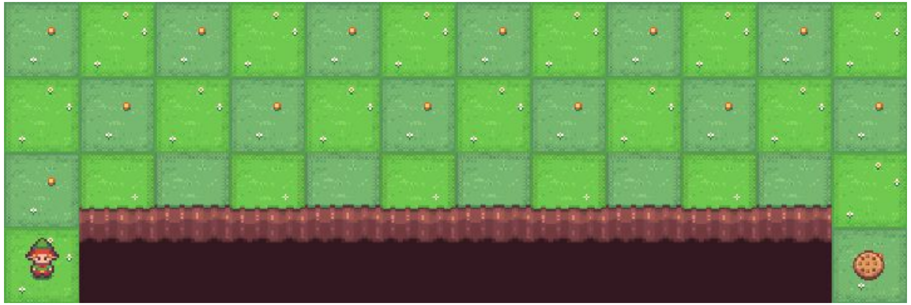
Example Q-Table:

Resulting Policy:

Implementation - Evaluation Environments

Gymnasium - CliffWalking-v0

- **# States:** 48
- **# Actions:** 4
- **Rewards:**
  - -100 falling into cliff
  - 1 for reaching goal
  - -1 otherwise



https://gymnasium.farama.org/environments/toy_text/cliff_walking/

Implementation - Evaluation Environments

Gymnasium - FrozenLake-v1

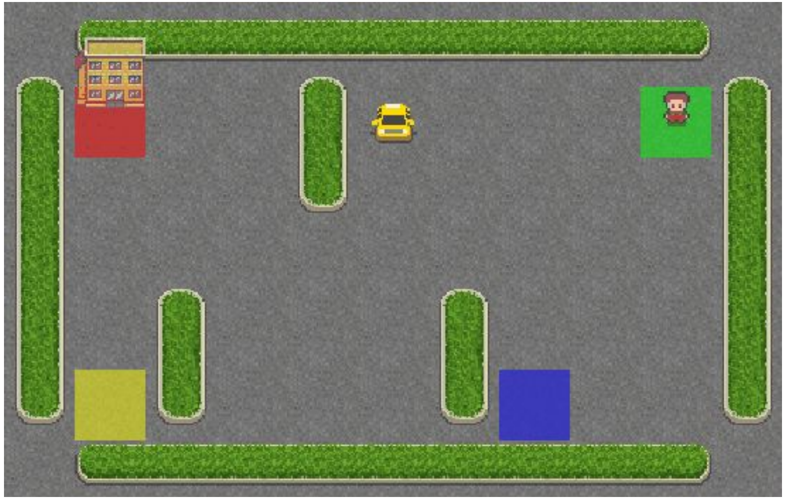
- **# States:** n^2 (variable)
- **# Actions:** 4
- **Rewards:**
  - 100 for reaching goal
  - -10 for falling into hole
  - -1 otherwise

https://gymnasium.farama.org/environments/toy_text/frozen_lake/

Implementation - Evaluation Environments

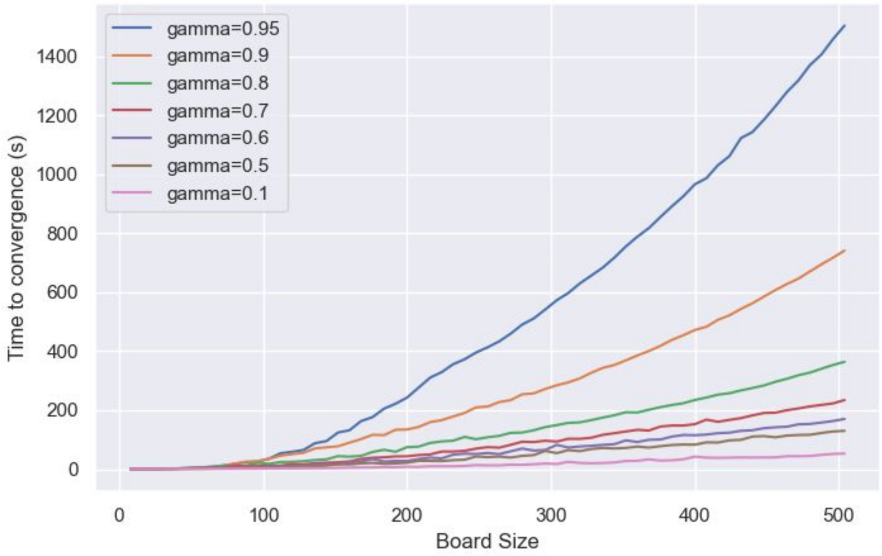
Gymnasium - Taxi-v3

- **# States:** 500 (404 reachable)
  - 25 taxi positions
  - 5 passenger locations
  - 4 destinations
- **# Actions:** 6
- **Rewards:**
  - 20 for delivering passengers
  - -10 for illegal pickup/dropoff
  - -1 otherwise



https://gymnasium.farama.org/environments/toy_text/taxi/

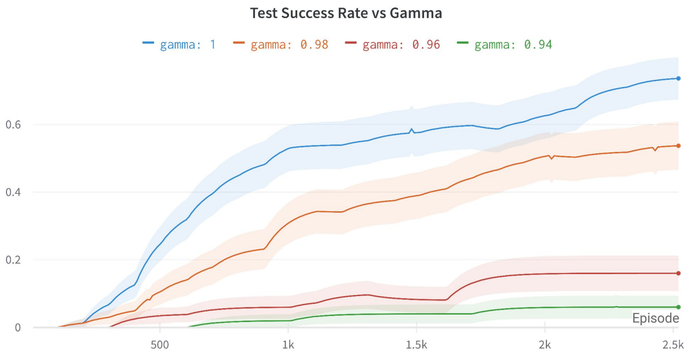
Implementation - Dynamic Programming Results

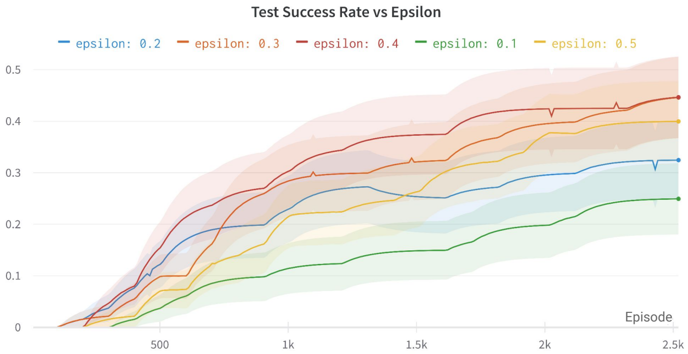


Environment: FrozenLake-v1

- Gammas tested: *[0.95, 0.9, 0.8, 0.7, 0.6, 0.5, 0.1]*
- Successful Gammas: [0.95, 0.9]

Implementation – Monte Carlo Results



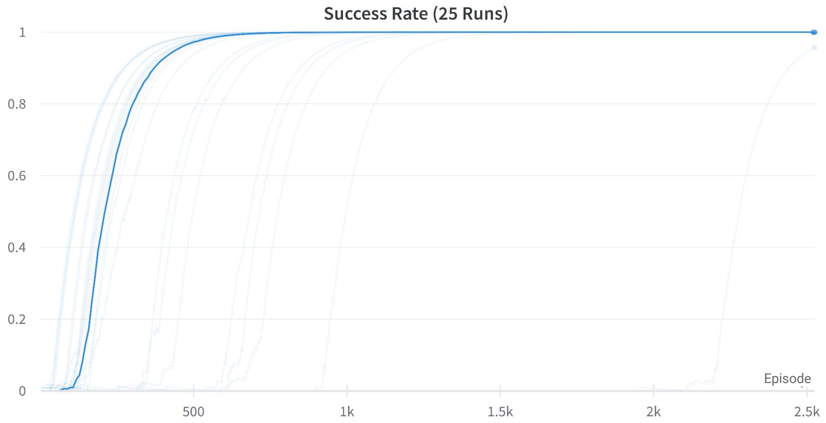


Environment: CliffWalking-v0

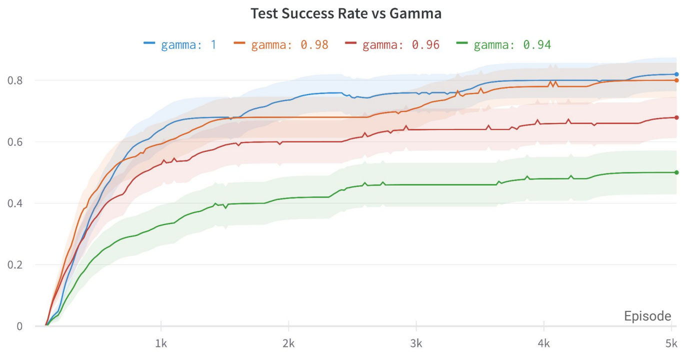
- Gammas tested: *[1.0, 0.98, 0.96, 0.94]*
- Epsilons tested: *[0.1, 0.2, 0.3, 0.4, 0.5]*

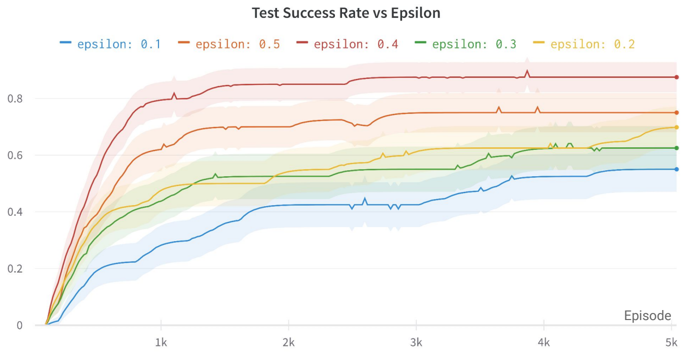
⟵ Average of 10 runs for each combination

Final Results ↴ with Gamma=1.0 and Epsilon=0.4



Implementation – Monte Carlo Results



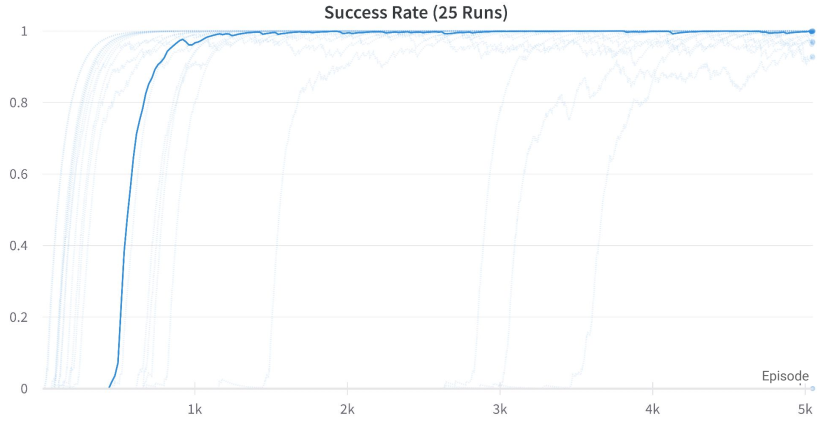


Environment: FrozenLake-v1

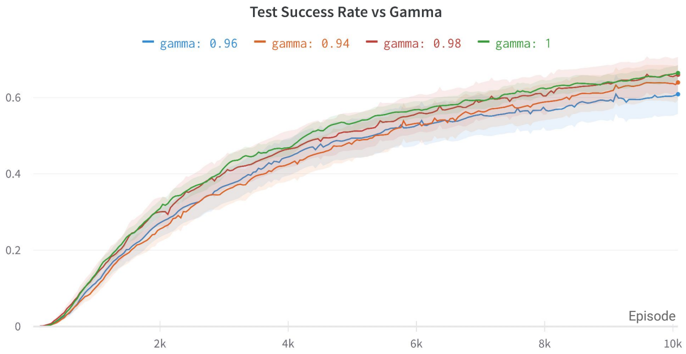
- Gammas tested: *[1.0, 0.98, 0.96, 0.94]*
- Epsilons tested: *[0.1, 0.2, 0.3, 0.4, 0.5]*

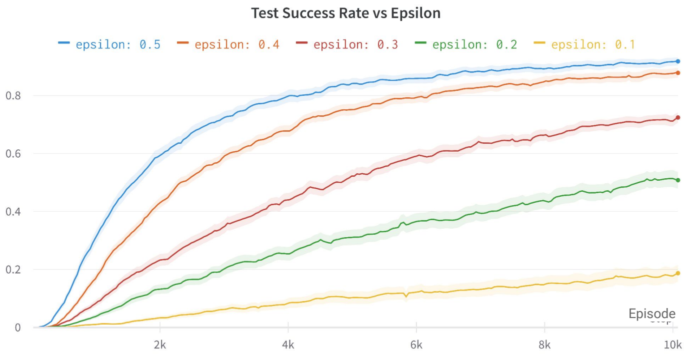
⟵ Average of 10 runs for each combination

Final Results ↴ with Gamma=1.0 and Epsilon=0.4



Implementation - Monte Carlo Results



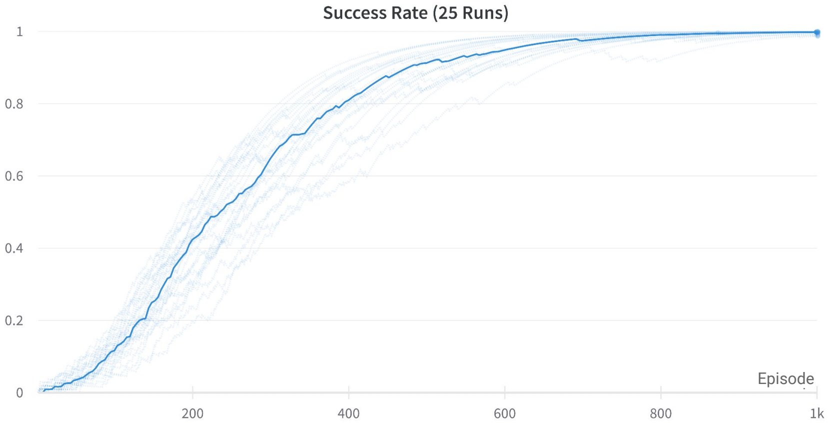


Environment: Taxi-v3

- Gammas tested: *[1.0, 0.98, 0.96, 0.94]*
- Epsilons tested: *[0.1, 0.2, 0.3, 0.4, 0.5]*

⟵ Average of 10 runs for each combination

Final Results ↴ with Gamma=1.0 and Epsilon=0.5



Implementation – Compare and Contrast

Number of Environment Samples*		
	Dynamic Programming	***Monte Carlo***
FrozenLake	3,328	24,193
CliffWalking	440,256	120,046
TaxiCab	3,894,000	1,055,934

*averaged over 10 runs

Implementation – Live Demo!

Hosted on HuggingFace Spaces: https://huggingface.co/spaces/acozma/CS581-Algos-Demo

GitHub Repository: https://github.com/andreicozma1/CS581-Algorithms-Project

Open Issues

- **Exploration vs Exploitation**
  - Exploit (Act Greedily) with respect to what it has already experienced to maximize reward.
  - Explore (Act Non-Greedily) or take actions which don't have the maximum expected reward in order to learn about new states
- **Stochastic Tasks**
  - Each action must be tried many times to gain a reliable estimate of its expected reward.
- **Delayed Reward**
  - Agents must consider more than the immediate reward because acting greedily may result in less future reward.
- **Sample Efficiency**
  - Some algorithms (like DP) require many environment samples which can be slow and inefficient

References

1. Sutton, R. S., & Barto, A. G. (2020). **Reinforcement learning: An introduction (2nd ed.).** MIT Press.
2. Dr. Sadovnik's ECE 517 Reinforcement Learning course (Fall 2022)

1. Llorente, F., Martino, L., Read, J., & Delgado, D. (2021). **A survey of Monte Carlo methods for noisy and costly densities with application to reinforcement learning**. *arXiv preprint arXiv:2108.00490*.
2. Ghojogh, B., Nekoei, H., Ghojogh, A., Karray, F., & Crowley, M. (2020). **Sampling algorithms, from survey sampling to Monte Carlo methods: *Tutorial and literature review.*** *arXiv preprint arXiv:2011.00901*.
3. Busoniu, L., Babuska, R., De Schutter, B., & Ernst, D. (2017). **Reinforcement learning and dynamic programming using function approximators.** *CRC press*.
4. Bellman, R. (1958). **Dynamic programming and stochastic control processes.** *Information and control, 1(3), 228-239*.
5. Voloshin, C., Le, H. M., Jiang, N., & Yue, Y. (2019). **Empirical study of off-policy policy evaluation for reinforcement learning.** *arXiv preprint arXiv:1911.06854*.
6. Świechowski, M., Godlewski, K., Sawicki, B., & Mańdziuk, J. (2023). **Monte Carlo tree search: A review of recent modifications and applications. Artificial Intelligence Review**, *56(3), 2497-2562*.
7. Afsar, M. M., Crump, T., & Far, B. (2022). **Reinforcement learning based recommender systems: A survey.** *ACM Computing Surveys, 55(7), 1-38*.
   Mo, S., Pei, X., & Wu, C. (2021). **Safe reinforcement learning for autonomous vehicle using monte carlo tree search.** *IEEE Transactions on Intelligent Transportation Systems, 23(7), 6766-6773*.
8. Barto, A. G., Thomas, P. S., & Sutton, R. S. (2017). **Some recent applications of reinforcement learning.** *In Proceedings of the Eighteenth Yale Workshop on Adaptive and Learning Systems*.
9. OpenAI Gym (2016) [Github] [paper]

Discussion

Test Questions (Revisited)

1. What does the “model” refer to in the terms “model-based” and “model-free”?
2. What are some of the limitations of Dynamic Programming methods?
3. How can you handle the exploration/exploitation trade-off in Monte Carlo methods?