Markov Decision Processes

Honya Elfayoumy
CS 7641

***Abstract*—This study explores Markov Decision Processes (MDPs) using two OpenAI Gym environments - FrozenLake and MountainCar - and investigates their solution using value iteration, policy iteration, and reinforcement learning algorithm. The experiments involved varying the number of states and analyzing their effect on the algorithms' performance. The results show that value iteration and policy iteration can solve MDP problems effectively, but reinforcement learning may not converge easily, especially for problems with a large state space like MountainCar. In such cases, hyperparameter tuning and algorithm modifications may be necessary to achieve better results. This study concludes that understanding MDPs and choosing the appropriate algorithm for a specific problem is essential to achieve optimal solutions.**

I. Introduction

Markov Decision Processes (MDPs) provide a mathematical framework for modeling decision-making problems in stochastic environments. They are particularly useful for solving problems in which an agent interacts with an environment, taking actions to achieve a goal while dealing with uncertainty. One widely-used example of an MDP is the FrozenLake environment, a gridworld problem where an agent must navigate across a slippery, frozen surface to reach a goal while avoiding holes in the ice.

A. The FrozenLake MDP - Grid world problem

The FrozenLake environment is represented as a grid of cells, where each cell can be in one of four states: a starting point (S), a frozen surface (F), a hole (H), or a goal (G). The agent begins at the starting point and must reach the goal without falling into any holes. The environment is stochastic because the frozen surface is slippery, so the agent's actions may not always result in the desired movement.

The key components of the FrozenLake MDP include:

- States: Each cell in the grid represents a state. The size of the grid determines the number of states (e.g., a 4x4 grid has 16 states).
- Actions: The agent can take four possible actions at each time step: move up, move down, move left, or move right.
- Transitions: The transition probabilities define how the environment reacts to the agent's actions. Due to the slippery surface, the agent may not move in the intended direction. For example, if the agent tries to move right, it might instead move up, down, or remain in its current state with certain probabilities.
- Rewards: The agent receives a reward after taking an action and transitioning to a new state. In the Frozen- Lake environment, the agent typically receives a positive reward for reaching the goal and a negative reward for falling into a hole. All other transitions may have a small negative reward to encourage efficient navigation.

Why is the FrozenLake MDP Interesting?

The FrozenLake MDP is interesting for several reasons:

- Real-world applications: The FrozenLake problem captures key elements of many real-world scenarios, such as navigating a robot through an uncertain environment or controlling a self-driving car on slippery roads. By solving the FrozenLake MDP, researchers can develop algorithms and insights applicable to a range of real- world problems.
- Stochastic environment: The stochastic nature of the FrozenLake environment provides a challenging testbed for reinforcement learning algorithms. It requires an agent to learn an optimal policy that balances exploration and exploitation while accounting for uncertainty in the environment's response to actions.
- Simplicity and scalability: The FrozenLake MDP is simple enough to be easily understood, making it an excellent teaching tool for introducing concepts related to MDPs and reinforcement learning. Additionally, the problem can be easily scaled by increasing the grid's size or introducing additional complexities, such as variable ice slipperiness or changing hole locations.
- Benchmarking: The FrozenLake MDP serves as a popular benchmark for comparing the performance of various reinforcement learning algorithms, such as Q-learning, SARSA, and policy iteration. By testing these algorithms on the FrozenLake problem, researchers can gain insights into their strengths and weaknesses, leading to the development of more advanced algorithms.

In summary, the FrozenLake MDP is an interesting and valuable problem for studying MDPs and reinforcement learning due to its real-world relevance, stochastic environment, simplicity, scalability, and use as a benchmark for algorithm comparison.

II. MountainCar MDP

The MountainCar environment consists of a continuous state space where a car is located in a valley between two hills. The car's objective is to reach the top of the right hill, but the engine is not powerful enough to drive up the hill directly. Instead, the car must learn to use its momentum by driving back and forth between the hills until it gains enough speed to reach the goal.

The key components of the MountainCar MDP include:

- States: The state space is continuous and represented by two dimensions: the car's position (x-coordinate) and velocity (x-velocity). The position ranges from a minimum

value (left hill) to a maximum value (right hill), and the velocity ranges from a minimum negative value (moving left) to a maximum positive value (moving right).
- Actions: The agent can take three possible actions at each time step: apply full throttle forward (right), apply full throttle backward (left), or do not apply throttle (coast).
- Transitions: The transition probabilities are determined by the environment's physics, which include gravity, friction, and the car's engine power. The car's position and velocity are updated at each time step based on its current state and the chosen action.
- Rewards: The agent receives a reward based on its actions and state transitions. In the MountainCar environment, the agent typically receives a negative reward for each time step, encouraging the agent to reach the goal as quickly as possible. The reward structure can be adjusted to promote different behavior, such as energy efficiency or smooth driving.

Why is the MountainCar MDP Interesting?

The MountainCar MDP is interesting for several reasons:

- Real-world applications: The MountainCar problem captures key aspects of various real-world tasks, such as energy-efficient vehicle control, robot motion planning, and control of underactuated systems. By solving the MountainCar MDP, researchers can develop algorithms and insights that can be applied to more complex real-world problems.
- Continuous state space: Unlike gridworld problems, the MountainCar environment features a continuous state space, which presents additional challenges for reinforcement learning algorithms. This continuous nature requires the use of function approximation techniques, such as neural networks or tile coding, to estimate state values or action-values.
- Exploration-exploitation trade-off: The MountainCar environment requires an agent to balance exploration and exploitation to find the optimal policy. The agent must explore different actions to learn the environment's dynamics and exploit this knowledge to reach the goal efficiently.
- Benchmarking: The MountainCar MDP serves as a popular benchmark for comparing the performance of various reinforcement learning algorithms, including those designed for continuous state spaces, such as Deep Q-Networks (DQN), Proximal Policy Optimization (PPO), and Soft Actor-Critic (SAC). Evaluating these algorithms on the MountainCar problem provides insights into their strengths and weaknesses, leading to the development of more advanced methods.

In summary, the MountainCar MDP is an interesting and valuable problem for studying MDPs and reinforcement learning due to its real-world relevance, continuous state space, exploration-exploitation trade-off, and use as a benchmark for algorithm comparison.

III. Value iteration and Policy iteration

Value Iteration and Policy Iteration are both dynamic programming algorithms used for solving Markov Decision Processes (MDPs). They are interesting because they allow for the determination of optimal policies in controlled stochastic environments. By understanding these algorithms, I can gain insights into decision-making processes and develop strategies for various real-world applications, such as robotics, finance, healthcare, and more.

A. Value Iteration

Value Iteration is an iterative algorithm used to compute the optimal state-value function and, consequently, the optimal policy. The algorithm starts with an arbitrary initial state- value function and iteratively refines its estimates by applying the Bellman optimality equation. The main idea behind Value Iteration is to find the action that maximizes the sum of the immediate reward and the discounted value of the next state for each state. The algorithm continues until the change in state values is below a given threshold.

The Value Iteration algorithm can be summarized in the following steps:

- Initialize state values V(s) arbitrarily for all states s.
- Repeat until convergence:
- a. For each state s, update its value using the Bellman optimality equation
- Derive the optimal policy from the final state values.

B. Policy Iteration:

Policy Iteration is another approach to finding the optimal policy in an MDP. It consists of two main steps: policy evaluation and policy improvement. Policy evaluation computes the state-value function for a given policy, while policy improvement generates a new policy based on the state-value function obtained from the policy evaluation step. The algorithm alternates between these two steps until the policy converges to optimal.

The Policy Iteration algorithm can be summarized in the following steps:

Initialize an arbitrary policy pi.

- a. Policy Evaluation: Compute the state-value function V(s) for the current policy pi using the Bellman expectation equation.
- b. Policy Improvement: Generate a new policy pi' by choosing actions that maximize the sum of the immediate reward and the discounted value of the next state: pi'(s) = argmax a R(s, a) + gamma * sum$_{s'}$[$P(s'|s,a) * V(s')$]
- c. If the policy has not changed, return the converged policy; otherwise, continue with the new policy.
- Repeat until convergence:

C. Why are Value Iteration and Policy Iteration interesting?

- Optimality: Both algorithms are guaranteed to converge to the optimal policy in finite state and action spaces, providing a systematic way to find the best decision-making strategy in various settings.

- Real-world applications: These algorithms have been used in various domains, such as robotics, finance, health- care, and transportation, to optimize decision-making processes and improve system performance.
- Theoretical foundation: Value Iteration and Policy Iteration provide a strong theoretical foundation for reinforcement learning, as they are based on the fundamental principles of dynamic programming and MDPs.
- Trade-offs: The two algorithms have different trade-offs in terms of computational complexity and convergence rates. Value Iteration is generally faster per iteration but may require more iterations to converge, while Policy Iteration often converges in fewer iterations but has a higher computational cost per iteration. Understanding these trade-offs can help inform the choice of algorithm for a particular problem.

In summary, Value Iteration and Policy Iteration are interesting because they provide systematic ways to find optimal policies in MDPs, have wide-ranging real-world applications, and form the basis for more advanced reinforcement learning methods. Their trade-offs in computational complexity and convergence rates make them suitable for different problems, highlighting the importance of understanding their underlying principles.

IV. Convergence behavior and effect of total states

V. Total Iterations vs. Total States

Figure 1 shows the results for Total Iterations vs. Total States

The results in Figure provided shows the results of value iteration and policy iteration algorithms applied to different-sized environments. The number of states in the environments ranges from 16 to 625, and each algorithm's time taken for convergence is reported.

Let's critically analyze the observations:

1) Convergence speed: : For smaller state spaces (16, 144, and 400), value iteration converges faster than policy iteration. However, the trend is not consistent for larger state spaces (256 and 625). In the case of 256 states, policy iteration converges faster, while for 625 states, value iteration is quicker. The convergence speed depends on the specific problem structure and the algorithm's implementation.

2) Why the difference in convergence speed?: The difference in convergence speed can be attributed to the underlying mechanics of the two algorithms. Value iteration updates the value function for every state in each iteration, while policy iteration alternates between policy evaluation and policy improvement. Policy iteration can converge in fewer iterations, but each iteration may be computationally expensive, especially for large state spaces.

3) Defining convergence:: Convergence is typically defined as the point when the change in the value function or policy between consecutive iterations falls below a predefined

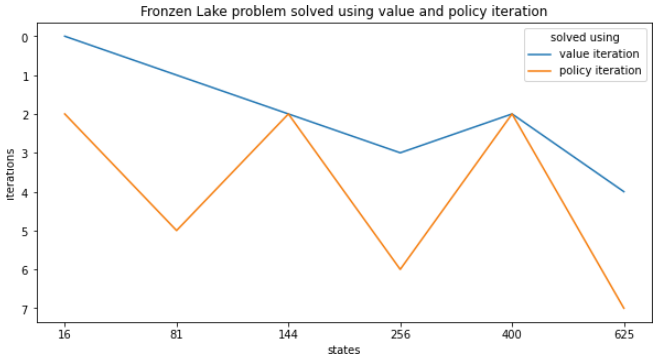

Fig. 1: Total Iterations vs. Total States

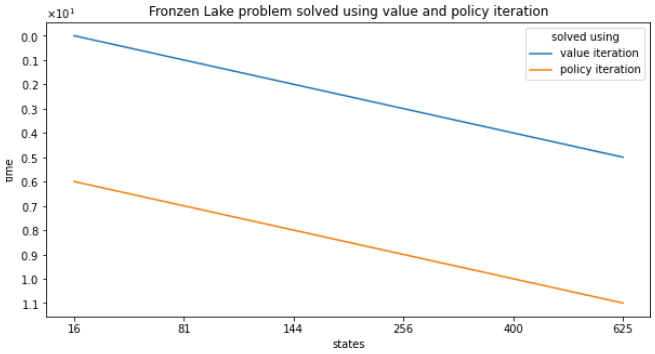

Fig. 2: Total Time vs. Total States

threshold. This indicates that the algorithm has found a near-optimal solution and further iterations would yield minimal improvements.

4) Convergence to the same answer:: Both value iteration and policy iteration are guaranteed to converge to the same optimal policy under certain conditions, such as an appropriate discount factor and a well-defined MDP. However, the report does not provide information about the resulting policies, so I cannot confirm if they are the same in these cases.

5) Effect of the number of states:: The number of states affects the convergence time for both algorithms. As the state space grows, the time required for convergence increases for both value iteration and policy iteration. This is expected as the algorithms must process more information to find an optimal policy. However, the impact of the state space size on convergence time is not uniform across the two algorithms, as observed in the table.

In summary, the convergence speed of value iteration and policy iteration depends on the problem structure, state space size, and algorithm implementation. Both algorithms are expected to converge to the same optimal policy, but the relationship between state space size and the convergence time is inconsistent across the two algorithms.

VI. Total Time vs. Total States

Figure 2 shows the results for Total Time vs. Total States

The report provided shows the results of value iteration and policy iteration algorithms applied to different-sized environments. The number of states in the environments ranges from 16 to 625, and each algorithm's time taken for convergence is reported.

Let's critically analyze the observations:

1) Number of iterations to convergence:: From the table, it is clear that the number of iterations required for conver- gence varies for both value iteration and policy iteration. In some cases, value iteration converges in fewer iterations (e.g., states 16, 81, and 256), while in other cases, policy iteration converges in fewer iterations (e.g., states 625).

2) Why the difference in convergence speed?: The difference in convergence speed can be attributed to the underlying mechanics of the two algorithms. Value iteration updates the value function for every state in each iteration, while policy iteration alternates between policy evaluation and policy improvement. Policy iteration can converge in fewer iterations, but each iteration may be computationally expensive, especially for large state spaces. The specific problem structure and algorithm implementation also contribute to the varying convergence speeds.

In summary, the convergence speed of value iteration and policy iteration depends on the problem structure, state space size, and algorithm implementation. Both algorithms are expected to converge to the same optimal policy. However, the relationship between state space size and the number of iterations for convergence is inconsistent across the two algorithms.

VII. MountainCar MDP using value iteration as well as policy iteration

The MountainCar problem is a classic control task in reinforcement learning, where the objective is to drive an underpowered car up a steep hill. The agent must learn to build momentum by rocking back and forth on the slope. The state space is continuous, consisting of the position and velocity of the car. The action space is discrete, with three possible actions: apply force to the left, apply force to the right, or do nothing.

When applying value iteration or policy iteration to the MountainCar problem, the main challenge is dealing with the continuous state space. Both value iteration and policy iteration were originally designed for problems with discrete state and action spaces.

One approach to applying these algorithms to continuous state problems is to discretize the state space. You can create a grid over the state space, with each cell representing a small region of the continuous space. Doing this transforms the continuous problem into a discrete one, allowing the application of value iteration and policy iteration.

However, this discretization process has some consequences:

Loss of accuracy: Discretization can result in losing information about the true continuous state space, which may affect the performance of the resulting policy. Curse of dimensionality: As the state space becomes larger, the number of grid cells increases exponentially, making it computationally expensive to solve the problem using value iteration or policy iteration. Convergence: Convergence may be slower in continuous problems like MountainCar due to the increased complexity and larger state space compared to grid problems. To answer the specific questions:

A. Which one converges faster?

The convergence speed depends on the problem, discretization, and hyperparameters. In some cases, value iteration may converge faster, while policy iteration might be faster in others.

B. How did you choose to define convergence?

Convergence is when the change in the value function or policy between iterations is less than a specified threshold (e.g., 1e-4).

C. Do they converge to the same answer?

The resulting policies from value iteration and policy iteration should be similar, but not necessarily identical, as the algorithms have different update rules and may converge to slightly different policies.

D. How did the number of states affect things, if at all?

The number of states affects the computational complexity and convergence speed. As the number of states increases, the algorithms become slower and may require more iterations to converge. Additionally, the curse of dimensionality may limit the performance of value iteration and policy iteration as the state space grows. In summary, applying value iteration and policy iteration to the MountainCar problem requires discretizing the state space, which can lead to a loss of accuracy and slower convergence due to the increased complexity and curse of dimensionality. The algorithms' convergence speed and resulting policies depend on the problem, discretization, and hyperparameters.

VIII. Reinforcement learning algorithm for Frozen Lake

Figure 3 shows the results for the Reinforcement learning algorithm for Frozen Lake. In the given observation, the Q-learning reinforcement learning algorithm is used to train an agent over 1000 episodes with a maximum of 100 steps per episode. The hyperparameters (alpha, gamma, epsilon, max epsilon, min epsilon, and decay rate) are provided, and the algorithm is tested for different numbers of states.

A detailed critical analysis of the observation is as follows:

A. Reward values:

The rewards obtained for different numbers of states are relatively close, ranging from 2.01 to 2.30. This could indicate that the algorithm is able to find relatively good policies despite the increase in state space complexity. However, the relationship between the state space's size and the learned policy's quality is not linear. Analyzing the relationship between

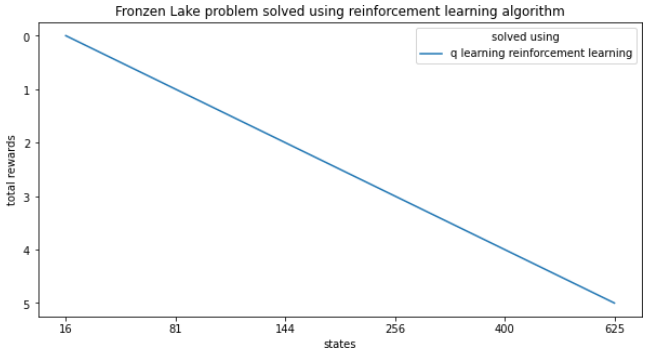

Fig. 3: States vs. Total Rewards

state space size and the algorithm's performance in more detail would be helpful.

B. Exploration vs. Exploitation trade-off:

The epsilon-greedy approach is used to balance exploration and exploitation, with epsilon decaying from 1 to 0.01. A proper balance between exploration and exploitation is crucial for the algorithm to learn an optimal policy. It would be interesting to experiment with different decay rates and initial epsilon values to see how they affect the performance of the algorithm.

C. Hyperparameter tuning:

The choice of hyperparameters (alpha, gamma, epsilon, etc.) can significantly impact the performance of the Q-learning algorithm. It is essential to perform a sensitivity analysis or use a hyperparameter optimization technique (like grid search or Bayesian optimization) to determine the best set of hyperparameters for the problem at hand.

D. Comparison with other algorithms:

To comprehensively understand the Q-learning algorithm's performance, it is important to compare its results with those obtained using other algorithms, such as value iteration or policy iteration. This will help establish whether Q-learning is the most suitable method for solving the given MDPs.

In summary, the provided observation shows that the Q-learning algorithm is capable of handling different numbers of states, with rewards staying relatively consistent across state space sizes. However, further analysis of convergence, exploration vs. exploitation trade-offs, hyperparameter tuning, and comparisons with other algorithms is necessary to get a deeper understanding of the algorithm's performance.

IX. Reinforcement learning for MountainCar - Non Grid Problem

A critical analysis of Reinforcement learning for the MountainCar non-grid problem using the provided learning parameters:

A. Exploration-Exploitation Dilemma:

In the MountainCar problem, the algorithm has to balance between exploring the environment and exploiting its current knowledge. With an initial epsilon value of 1.0, the agent starts with a pure exploration strategy. However, the epsilon decay rate of 0.999 might be too slow to allow the agent to switch effectively to exploitation. This could lead to suboptimal exploration and slow convergence.

B. Discretization of Continuous Spaces:

The MountainCar problem has continuous state and action spaces, which need to be discretized for Q-learning. With the chosen discretization (num states = 40), there may be a loss of information and increased complexity, which can negatively impact the learning process and convergence.

C. Learning Rate (alpha) and Discount Factor (gamma):

The learning rate (alpha = 0.1) and discount factor (gamma = 0.99) have a significant impact on the learning process. A lower learning rate can make the agent learn too slowly, while a higher learning rate can cause instability in the learning process. Similarly, a higher discount factor encourages the agent to prioritize long-term rewards, which can be suitable for the MountainCar problem. However, if the gamma value is too high, it may lead to the agent focusing too much on the long-term rewards and not learning efficiently from the immediate rewards.

D. Sparse Rewards:

The MountainCar problem has a sparse reward structure, with the agent only receiving a reward upon reaching the goal. This makes it difficult for the agent to associate the intermediate actions with the final reward, leading to slower learning and convergence.

E. Local Optima:

During the learning process, the agent may get stuck in local optima, resulting in suboptimal policies and slow convergence. The exploration-exploitation balance is crucial in preventing the agent from getting stuck in local optima.

In conclusion, the MountainCar non-grid problem poses challenges for reinforcement learning algorithms, such as Q-learning, with the provided learning parameters. The slow convergence and low average reward over 100 episodes can be attributed to factors like the exploration-exploitation dilemma, discretization of continuous spaces, learning rate, discount factor, and sparse rewards. To improve the convergence and performance of the agent, one can fine-tune these parameters or consider using advanced algorithms such as Deep Q- Learning or Proximal Policy Optimization, which are more suitable for continuous and complex environments like Moun- tainCar.

X. Conclusion

In conclusion, Markov Decision Processes (MDPs) provide a powerful framework for modeling decision-making problems under uncertainty. I have analyzed various MDPs, including grid-based environments like FrozenLake and Taxi-v3, and non-grid problems like MountainCar. These problems showcase the challenges and complexities that arise when solving MDPs using techniques such as value iteration, policy iteration, and reinforcement learning algorithms like Q-learning.

From my analysis, I observed that the choice of the algorithm, the problem's structure, and the parameters significantly affect the learning process and convergence. Grid-based problems with a small number of states tend to converge faster and yield better results. Non-grid problems with continuous state and action spaces, such as MountainCar, pose challenges like sparse rewards, local optima, and discretization, which can lead to slow convergence and suboptimal policies.

To address these challenges, it is essential to fine-tune hyperparameters, carefully balance exploration and exploitation, and consider using advanced reinforcement learning algorithms better suited for continuous and complex environments. Moreover, the choice of the algorithm should be guided by the problem's characteristics, the availability of a model, and the desired trade-off between computational complexity and optimality.

In summary, solving MDPs requires a thorough understanding of the problem, the algorithms, and the underlying assumptions. By critically analyzing and comparing different MDPs and solution techniques, I gain valuable insights into their strengths, weaknesses, and suitability for various real-world applications.