Ministry of Education and Science of Ukraine

V. N. Karazin Kharkiv National University

School of Mathematics and Computer Science

Department of Theoretical and Applied Informatics

Qualification work

Master

On the topic: Program Implementation of Markov Decision Processes

specialty - Computer Sciences and

Information Technologies,

educational program: "Informatics"

Guo Qihe

Supervisor: Kostiantyn Nosov

Reviewer:

Adviser:

Kharkiv, 2024

Abstract: Markov Decision Processes (MDP) is an important mathematical model for solving dynamic and stochastic decision-making problems and is widely used in many fields. This study systematically implements the core algorithms of MDP and comprehensively explores its application in solving complex decision-making problems, especially through empirical analysis in two fields: robot navigation and financial investment. The theoretical basis of MDP is introduced, including its definition, constituent elements, and its relationship with reinforcement learning. Subsequently, the program implementation methods of MDP algorithms are discussed in detail, including the implementation of value iteration, policy iteration, Q-learning, and deep Q-learning (DQN), and the performance of these algorithms is evaluated. By designing specific application cases, the paper demonstrates the effectiveness of MDP algorithms in different scenarios and analyzes their limitations in practical applications. The study finds that MDP algorithms have good decision-making optimization capabilities in dynamic and stochastic environments, but they also face challenges such as high computational complexity and slow convergence speed. The paper concludes by proposing future research directions, including optimizing algorithms to improve computational efficiency and convergence speed, enhancing the adaptability and robustness of algorithms, and exploring the in-depth integration of MDP with other AI technologies. This research not only provides theoretical support and empirical evidence for the application of MDP algorithms in multi-stage decision-making problems but also points out the direction for the future application of MDP models in a broader range of fields.

Keyword: Markov Decision Process; Reinforcement Learning; Algorithm Implementation; Application Case; Performance Evaluation

CATALOGUE

INTRODUCTION

Research background and significance

As an important mathematical model, Markov Decision Process (MDP) can describe the stochastic and uncertain decision process in dynamic environment. MDP model is widely used in artificial intelligence, machine learning, operations research and other fields to solve sequential decision problems. With the development of intelligent systems, MDP has shown great application potential in autonomous driving, robot control, resource allocation, financial market analysis and other aspects. In these scenarios, systems often need to consistently make the best decisions in an environment that is not completely certain in order to maximize long-term returns, so MDP provides a systematic way to optimize this decision-making process.

In the field of artificial intelligence and machine Learning, MDP provides the theoretical foundation for Reinforcement Learning (RL). [1] By constantly interacting with the environment, reinforcement learning models learn a set of optimal strategies to obtain the highest returns. In the application of RL, the state transition and reward mechanism of MDP are closely related to the practical application. Therefore, MDP has become the core of reinforcement learning theory and application. In recent years, with the improvement of computing power and the development of big data, the combination of Deep Reinforcement Learning (DRL) and MDP has shown unprecedented advantages in solving complex problems, such as deep Q-learning algorithm and near-end strategy optimization algorithm. It

greatly promotes the application prospect of MDP.

The field of operations research also relies on MDP to solve many practical problems, such as inventory management, supply chain optimization, and traffic scheduling. The MDP model can deal with the unpredictable dynamic environment in these fields, and provides a scientific decision-making framework for optimization through the analysis of system state, action and transition probability. For example, in inventory management, the use of MDP can effectively balance inventory levels with service quality, thereby reducing operating costs.

To sum up, the study of MDP is of great significance both in theory and practice. At the theoretical level, it provides a clear framework for sequential decision making under uncertain environment. In practice, it has been widely used in many fields. With the rapid growth of large-scale data and computing power, it is of great academic value and practical significance to study the algorithm and program implementation of MDP, especially its efficiency and applicability in different application scenarios.

Literature review

The theory of MDP dates back to the 1950s, when it was proposed by Richard Bellman, an American mathematician. The Dynamic Programming theory proposed by Bellman laid the foundation of MDP and provided an effective method to solve the decision problem with the goal of maximizing long-term returns. As two classical algorithms of dynamic programming, Value Iteration and Policy Iteration achieve optimal decisions by updating state value function and policy improvement function respectively. [2] These algorithms solve the problem

of finding the optimal strategy in a deterministic environment and are widely used in simple decision scenarios.

Over time, the theory and applications of MDP have evolved. In the 1990s, with the enhancement of computer computing power, Q-learning and other model-free reinforcement learning methods were proposed. [3]Q-learning learns the optimal state-action value function through the balance of exploration and utilization, and no longer relies on the complete environment model. This approach greatly expands the application scenarios of MDP, enabling it to be applied to more complex, not fully understood environments. In recent years, the deep Q-learning (DQN) algorithm combined with deep neural networks has further improved the applicability of MDP in high-dimensional space and continuous state space. In the field of game AI, robot navigation and other fields, DQN has demonstrated remarkable results.

In the international research field, the application range of MDP continues to expand, from traditional operations research to artificial intelligence, automatic control, and even involving interdisciplinary applications such as finance and medical care. Researchers not only focus on the improvement of classical MDP algorithm, but also carry out research in complex scenarios such as partially observable Markov decision process (POMDP) and continuous state space MDP. [4] For example, in the POMDP model, partial observability allows the system to make decisions under conditions of limited information, providing theoretical support for applications such as robot positioning.

To sum up, MDP has made great progress in theory and application. The

literature presents the evolution process from classical algorithms (value iteration, strategy iteration, Q-learning) to modern combined methods of deep learning (deep Q-learning, PPO), showing the wide application and great potential of MDP in various fields. Research on the program implementation of MDP will further promote its popularization in different application scenarios.

Research purpose and problem definition

This study aims to systematically implement the core algorithm of MDP, and on this basis, demonstrate its applicability and effect in different scenarios through practical cases. Specific research objectives include:

The classical algorithms to implement MDP include value iteration, strategy iteration and Q-learning. These algorithms represent typical applications of MDP in different types of scenarios, so an important goal of this study is to implement these algorithms programmatically and explore their applicability to a variety of decision problems. Explore the optimization way of MDP algorithm: In different application scenarios, the implementation efficiency and effect of MDP algorithm are different. Therefore, how to improve the operation efficiency of the algorithm, how to deal with large-scale state space, how to deal with continuous state space and other problems become one of the focuses of this research. To this end, the research will explore Q-learning improvement schemes based on deep learning (such as DQN) and experiential playback techniques to enhance the performance of algorithms in complex environments. Design application cases and carry out experimental verification: This study will carry out application design in the fields of robot navigation and financial investment optimization, and verify the

performance and effect of MDP algorithm in different application scenarios through experiments. The research will focus on the analysis of the advantages and disadvantages of different algorithms under different conditions, operation results and applicability, to provide reference for the practical application of MDP algorithm.

Based on the above purposes, this research mainly focuses on the following questions: How to implement the classical algorithm of MDP? How to improve the computational efficiency of the algorithm? In different application scenarios, how to choose suitable MDP algorithm and obtain good decision effect?

Paper structure overview

This paper is divided into five chapters, the structure is arranged as follows:

The first chapter introduces the definition and basic elements of MDP, including state space, action space, transition probability and reward function. At the same time, the principles of dynamic programming methods such as value iteration and policy iteration are introduced, and the extended models of MDP, such as partially observable MDP (POMDP) and continuous state space MDP, are discussed. In addition, this chapter also analyzes the relationship between reinforcement learning and MDP, which lays a theoretical foundation for the following chapters.

The second chapter introduces the implementation of Markov decision process, including the implementation of value iteration, strategy iteration and Q-learning algorithm. The selection of programming environment (such as Python language, NumPy library, etc.) and the key technologies in the implementation

process are explained in detail. For the problem of high dimensional state space, this chapter discusses how to optimize Q-learning algorithm by deep learning method.

Chapter 3 Application case design and result analysis: design and implement two specific application scenarios - robot navigation and financial investment optimization. Experimental data are used to show the effects of different MDP algorithms in these two scenarios, including strategy iteration process, convergence curve and strategy income. The performance of different algorithms in different application scenarios is compared, and their applicability and limitations are discussed.

The conclusion part summarizes the main research results of the paper, including the realization, optimization and application scenario test analysis of MDP algorithm. Discuss the limitations of the study and make suggestions for future research directions, such as improving the scalability of the algorithm, exploring more application scenarios, and combining with other technologies, such as deep learning.

CHAPTER ONE THE THEORETICAL BASIS OF MARKOV DECISION PROCESS

1.1 Definition and constituent elements of MDP

Markov decision process (MDP) provides a mathematical framework for solving decision problems with uncertainty and randomness. [5]MDP has an extremely wide range of applications, covering many fields such as autonomous driving, robot control, financial investment, medical decision making, etc. In these scenarios, decisions are not just immediate and often need to take into account future states and rewards, making MDP an ideal modeling tool.

State space \(S \): The state space \(S \) is the most fundamental component in MDP and represents the set of all possible states of a system. Each state \(s \in S \) represents the "situation" of the system at a certain moment in time, i.e., the complete description of the system. [6] For example, in an autonomous driving scenario, states can include the vehicle's current position, speed, acceleration, and the state of the surrounding traffic; In a stock investment, the state may include current stock prices, market trends, macroeconomic indicators, etc. State Spaces can often be discrete or continuous. For discrete state Spaces, each state can be explicitly listed; For continuous state Spaces, the states are usually the range of values of some physical quantity, which needs to be solved by function approximation or discretization techniques. For large-scale systems, the size of the state space will increase rapidly, resulting in the so-called "dimensional disaster", so the choice and simplification of the state space is very critical in the application

of MDP.

Action space \(A \): The action space \(A \) represents all the possible actions that the agent can choose in each state. The action space can be discrete or continuous. [7] In a discrete action space, the agent may have a limited choice of actions; Whereas in continuous action space, every decision of the agent has infinite possibilities, such as controlling the Angle of the robot or the speed of movement. For example, in financial investing, actions might include "buy", "sell" and "hold"; In robot control, the actions might be "forward," "left," "right," or even "speed up" or "slow down." The design of action space directly affects the complexity of MDP solution, especially in continuous action space, it is necessary to use reinforcement learning and other methods to approximate the optimal strategy.

State transition probability \(P(s' |s, a) \): The state transition probability \(P(s' |s, a) \) is the core of MDP and represents the probability of moving to the next state \(s' \) after performing an action \(a \) at the current state \(s \). This transition probability reflects the dynamic properties of the system and is usually provided by either a physical model of the system or a data-driven model. [8] In the MDP model, it is assumed that the transition probability is known, but in practice the transition probability in many applications is not fully known, especially in systems that are not fully observable. The state transition satisfies the Markov Property, which states that "the transition of the current state depends only on the current state and current actions, and is independent of past states and actions". This property greatly simplifies the calculation and decision making process,

making MDP problems solvable in many real-world scenarios. However, in practical applications, sometimes state transitions are approximate or fuzzy, especially in partially observable Markov decision processes (POMDP), where the real state of the system cannot be directly observed and the next state needs to be predicted by inference.

Reward function \(R(s, a) \): The reward function \(R(s,a) \) defines the immediate reward that the agent receives when it performs an action \(a \) under state \(s \). The reward is usually a real number that indicates how good or bad the agent's behavior is. For example, in autopilot, staying in the center of the lane may result in a higher reward, while hitting an obstacle results in a negative reward; In financial investing, a high-return investment strategy may result in a high reward, while a loss will result in a negative reward. The design of the reward function is critical because it directly affects the agent's behavioral goals. The reward function can be immediate or an expected reward in the future. To balance immediate and future rewards, it is common to introduce a discount factor \(\gamma \), where \(0 \leq \gamma \leq 1\). The discount factor determines how much value the agent places on future rewards. When \(\gamma \) is close to 1, the agent tends to value long-term rewards, while when \(\gamma \) is small (close to 0), the agent focuses more on short-term rewards.

Strategy \ (\ \) (a | s) PI: strategy \ (\ PI (a) | s \) defines the agent under the condition \ (\ s) take action \ (\) a probability distribution. [9] A strategy can be either deterministic or random. If a definite action is chosen in each state, it is called a deterministic strategy; If the actiontaken in each state is random, it is

called a random strategy . The optimal strategy is the one that allows the system to get the maximum cumulative reward over the long term. In practical application, the choice of strategy not only affects the immediate decision result, but also affects the long-term performance of the whole system. Policy learning and optimization is one of the core problems of MDP, which usually needs to be iteratively optimized by algorithms.

1.2 Dynamic programming and optimal policy solving of MDP

Dynamic Programming (DP) is a classical method to solve MDP problems. The DP method approximates the optimal solution by solving subproblems in stages. By updating the value function, dynamic programming can find the optimal strategy under the premise of known environment model. [10] The optimal policy of MDP can be obtained by solving its value function or action value function. There are two classical methods for solving MDP problems: Value Iteration and Policy Iteration.

Bellman equation and value function: The Bellman equation is the basic formula describing the solution of the optimal MDP policy. It describes, through a recursive relationship, how the value of each state depends on the expected value of current and future rewards. For a given strategy \ (\ \) PI, behrman equation of the form of a: \ [V ^ \ PI (s) = R (s \ PI (s)) + \ gamma \ sum_ {' s} P (s' \ | s, PI (s)) V ^ \ PI (s') \]. The \ (V ^ / PI (s) /) is under the strategy \ (\ \) PI state \ (\ s), the value of \ \ gamma (\) is the discount factor, said the value of future rewards to (P (s \ ', \ | s PI (s)) \) is the state transition probability, Said to take action under state \

(\ s) \ [/ PI (s) /) transferred to the state after \ '(s \) probability. The Bellman equation can be used not only to calculate the value function, but also to evaluate how good or bad different strategies are. When we find the strategy that maximizes the value \(V(s) \) of the state, we get the optimal strategy.

Value iteration algorithm: Value iteration is an iterative algorithm for solving the optimal policy for MDP. [11] Value iteration updates the value function for each state by repeatedly applying the Bellman equation until the value function converges to the optimal value function. The core steps of value iteration are as follows:

Initialize value function: Given an initial state value \(V_0(s) \), it is usually possible to set the initial value of all states to zero or a constant.

Update: The value in each iteration, use behrman equation to update the value of each state: \ [V_ (s) = {k + 1} \ max_a \ sum_ {' s} P (s' | s, a)/R (s, a) + \ gamma V_k (' s)] \]

Convergence judgment: Check whether the change in value of all states is less than a preset threshold. If so, then stop the iteration, otherwise continue updating. [12] The advantage of the value iteration algorithm is that it is simple, intuitive, and can find the optimal strategy by progressively approximating the optimal value function. However, value iteration requires a large amount of computation, especially in the case of large state space and action space, which requires a lot of computing resources.

Strategy iteration algorithm: Strategy iteration is a phased dynamic programming algorithm, which mainly consists of two steps: strategy evaluation

and strategy improvement. The basic process of policy iteration is as follows:

Strategy evaluation: Given a strategy \(\pi \), calculate the value function \(V^\pi(s) \) for each state, which can be done by the Bellman equation.

Strategy Improvement : According to the current value function \(V^\pi(s) \), select the action that maximize the value and update the strategy: \[\pi'(s) = \arg\max_a \sum_ {s'} P(s' |s, a) [R(s, a) + \gamma V^\pi(s')]\]

Convergence judgment: If there is no change in the strategy, the algorithm terminates and the optimal strategy is obtained. If the policy has changed, go back to step 1.

Policy iteration is usually more efficient than iteration, especially in some more complex MDP problems, because each policy evaluation and improvement converges to the optimal policy within a finite number of iterations. However, a potential problem with policy iteration is its high computational overhead, especially in large scale state Spaces.

Relationship between Policy and Value Function: There is a close relationship between value function and policy. In the MDP model, the value function describes the expected return that can be obtained by following a particular strategy in a certain state. The optimal strategy, on the other hand, is the one that maximizes the value function. By solving the value function, the optimal strategy can be derived and the optimal decision scheme can be obtained. The strategy iteration and value iteration algorithms approach the optimal solution by constantly adjusting the value function and strategy.

1.3 Expansion of MDP

The standard Markov decision process (MDP) model is limited in real-world applications by its fully observability assumptions about states and transitions. [13] In order to adapt to more complex and dynamic environments, researchers have proposed a variety of extended models that can deal with the requirements of scenarios such as partial observation, continuous state space, and different time granularity. These extensions not only improve the adaptability of the MDP model, but also promote its application in more fields.

Partially observable Markov decision process (POMDP) : In many practical scenarios, the system cannot observe the complete state information, but can only get incomplete feedback of the environment through partial observation. For example, when a robot is walking in a complex environment, it cannot fully observe the specific position of the surrounding obstacles, which makes the standard MDP difficult to apply. For this reason, the partially observable Markov decision process (POMDP) is proposed to solve the decision problem with incomplete observation. In POMDP, the system introduces an Observation Space \(O \) and expresses the probability that a particular information \(o \) is observed in the current state and action by the observation probability \(P(o|s, a) \). In POMDP, the agent forms a Belief State about the state of the system by making inferences about the observed state. The belief state is a probability distribution representing the agent's estimate of the true state, which is constantly updated as new observations come in. Solving POMDP problems is usually complicated because the system needs to find an optimal strategy for each belief state. Common

approaches to solving POMDP include the following:

Approximation methods based on particle filtering: Particle filtering approximates the true state by discretizing the belief state into a collection of particles and updating those particles based on observations. Particle filtering is a computationally efficient algorithm that is widely used in many POMDP applications.

Belief state space value iteration: POMDP can find the optimal policy in the belief state space through the value iteration algorithm. Since belief Spaces are usually continuous, the value iteration process can become complicated, so discretization or approximation methods need to be introduced to simplify the calculation.

POMDP is widely used in robot navigation, autonomous driving, medical diagnosis and other fields. For example, in robot navigation, robots can obtain part of the environment information through lidar or cameras and infer the optimal path based on observation and belief states to achieve obstacle avoidance and target point navigation.

Continuous state space MDP: In scenarios such as autonomous driving and robot control, states and actions are often continuous. For these applications, standard discrete MDP methods are difficult to apply directly. [14] Therefore, continuous state space MDP becomes an important extension. The main challenge of its solution lies in how to represent and approximate the continuity of the state space and the action space.

Common approaches to solving continuous state space MDP include:

Function based approximation methods: Function approximation can be used to represent continuity in a state or action space. Methods such as Radial Basis Function (RBF), polynomial approximation, linear approximation, etc., are commonly used to deal with continuous state Spaces in low dimensions. The radial basis function reduces computational complexity by building a set of basis functions to approximate the state-valued function.

Approximation based on neural networks: In recent years, deep neural networks have been used to solve continuous state space MDP, and neural networks realize effective representation of high-dimensional states through automatic feature extraction. Algorithms such as deep Q-learning and deep Deterministic policy gradient (DDPG) approach state values or policy functions through neural networks to solve decision-making problems in high-dimensional continuous space. Such methods have achieved remarkable results in autonomous driving, drone navigation, financial investment and other fields.

Hierarchical MDP (Hierarchical MDP) : In complex task scenarios, decision problems can often be decomposed into multiple levels. Hierarchical MDP allows agents to formulate policies at each level by dividing the decision process into sub-tasks at different levels. [15] For example, in robot path planning, higher-level decisions can plan out the target points, while lower-level decisions are responsible for specific navigation and obstacle avoidance. Hierarchical MDP usually consists of two modules, high level and low level, each of which is an MDP model and realizes hierarchical control through its own policies. This layered design shows significant efficiency advantages in decision problems for complex tasks.

In summary, extended models such as POMDP, continuous state space MDP and hierarchical MDP are suitable for different decision problems, greatly expanding the application range of MDP, and providing effective solutions for decision problems in complex, dynamic and incomplete observable environments.

1.4 Reinforcement learning and MDP

Reinforcement Learning (RL) is a learning framework closely combined with MDP theory, which is committed to learning optimal decision strategies through environmental interaction and feedback mechanisms. The goal of reinforcement learning is to obtain strategies that maximize long-term returns through the Trial-and-Error process. RL does not just rely on preset models, but improves the agent's decision-making ability through learning and adaptation, which makes it widely used in situations where the environment is dynamically changing and the model is unknown.

The basic framework of reinforcement learning :RL models typically include agents, environments, states, actions, and rewards. The agent selects an action in a certain state, influences the state transfer of the environment, and receives feedback (i.e. reward) from the environment. This feedback helps the agent to determine whether the choice is favorable and thus adjust the strategy. The MDP model provides theoretical support for RL, which is based on the state, action, reward and transfer mechanism of MDP to obtain the optimal strategy by maximizing the cumulative return. Here are a few main algorithms in reinforcement learning:

Q-learning: Q-Learning is a model-free RL algorithm that is primarily used to estimate state-action value functions \(Q(s, a) \). Q-learning updates the \(Q \) value after each action selection, allowing the algorithm to obtain the optimal strategy without fully knowing the environment. Q-learning's update rules are: \ [Q \ leftarrow (s, a) Q (s, a) + \ alpha [R (s, a) + \ gamma \ max_ {a '} Q (s) ', a '- Q (s, a)] \] in this formula, the vector \ \ alpha \ control update rate, The discount factor \(\gamma \) balances the weight of immediate and long-term returns. The model-free nature of Q-learning makes it widely used in a variety of complex environments, especially in the field of game AI, intelligent transportation, etc.

Deep Q-Learning (DQN) : Deep Q-Learning (DQN) approximates \(Q(s, a) \) functions by using neural networks, so that Q-Learning can be applied to high-dimensional state Spaces. DQN improves the stability of the algorithm by introducing Experience Replay and Target Network. In experience Replay, the algorithm randomly draws samples from past experiences, reducing the correlation of the data and preventing oscillations during network training. The target network is used to store the updated \(Q \) values, helping to achieve stable convergence in the training process. The successful application of DQN in Atari game testing made it an important algorithm for deep reinforcement learning and pushed forward the field of game AI and robotics.

Strategy gradient method: Strategy gradient method is a kind of RL algorithm that directly optimizes strategy, mainly by optimizing the probability distribution of strategy, so as to maximize the expected return. [16] Policy gradient methods are usually suitable for continuous action space problems, the most common methods

include REINFORCE algorithm and Proximal Policy Optimization (PPO). The strategy gradient method avoids the difficulty of approximating the value function by directly learning the parameters of the policy function. PPO is a widely used reinforcement learning algorithm, which makes the policy update more stable by limiting the update stride. The strategy gradient method is especially suitable for complex continuous action decision problems such as robot control and manipulator operation.

The relationship between reinforcement learning and MDP :MDP provides a structured mathematical framework for RL, while RL searches for optimal strategies in uncertain environments through a trial-and-error process. In RL, many algorithms are based on the value function and policy structure of MDP, and dynamic programming and Bellman equations are at the core of these algorithms. RL realizes the adaptive decision of complex environment through MDP model, which enables RL to realize autonomous learning and behavior choice in unmodeled environment.

In addition, reinforcement learning also promotes the expansion and improvement of MDP. For complex environments where states and actions are difficult to define, RL provides model-free learning methods that allow decisions to be optimized in real environments. [17] The reinforcement learning methods represented by Q-learning and DQN have made breakthroughs in many application fields, not only promoting the performance improvement of AI in games, but also providing effective solutions for UAV flight control, automatic driving and medical decision-making. By combining MDP theory and reinforcement learning

technology, the system is able to gradually optimize behavior in dynamic and uncertain environments, and gradually reach global optimal. This combination shows a broad application prospect in the field of artificial intelligence, and provides new theoretical and technical support for solving the sequential decision problem in complex systems.

CHAPTER TWO THE PROGRAM REALIZATION METHOD OF MARKOV DECISION PROCESS

In this chapter, we will deeply discuss how to implement the algorithm of Markov decision process (MDP), and introduce the environment setting, technology selection, the specific implementation steps of the algorithm and the performance evaluation and optimization method in detail. The focus of this chapter is to translate the theory of MDP into practical programs through programming languages and tools, combined with specific algorithm implementation, and then verify the effectiveness of MDP in different problems.

2.1 Programming environment and technology selection

In the process of implementing Markov Decision Process (MDP), it is crucial to choose the right programming environment and tools. A good programming environment and technology selection can not only improve the development efficiency, but also ensure the scalability, computational performance and ease of use of the algorithm. This section will introduce in detail the programming environment, library and related technology selection reasons used in the implementation of MDP algorithm.

Programming language selection: Python

Python is our primary programming language of choice. With its simple syntax, powerful data processing capabilities, and rich library of scientific computations, Python is the language of choice for data science, artificial

intelligence, and machine learning. Here are a few main reasons to choose Python:

Simple and easy to learn: Python's syntax is concise and clear, suitable for quickly implementing complex algorithms, and easy to read and maintain. Even beginners can quickly learn and understand the logic of the code. Rich scientific computing libraries: Python has powerful scientific computing libraries such as NumPy, SciPy, Pandas, Matplotlib, etc., which can help us efficiently handle matrix arithmetic, data analysis, and visualization work. Operations such as state transition matrix, reward matrix and value function in MDP usually involve a lot of numerical computation, and Python's numerical computation library makes these operations more convenient. Machine Learning and reinforcement Learning support: Python's machine learning and deep learning frameworks (e.g. Scikit-learn, TensorFlow, PyTorch) provide great convenience for developing reinforcement learning (RL) algorithms. [18]MDP solutions are not limited to classical dynamic programming methods, but can also be combined with reinforcement learning methods to deal with more complex problems, and Python provides excellent support in this regard. Cross-platform compatibility: Python is cross-platform and can run on Windows, Mac, Linux and other operating systems, ensuring code portability and compatibility across different platforms.

Key libraries and frameworks: In specific implementations, we rely on several efficient and powerful Python libraries to accelerate the development and implementation of algorithms. Here are the main technical libraries we selected and their uses:

NumPy: NumPy is the basic library in Python for efficient numerical

computation, and is particularly good at dealing with matrix operations and multidimensional arrays. Many operations of MDP, such as multiplication of state transition probability matrices, update of value functions, etc., involve a lot of matrix operations, and NumPy provides efficient vectorization operations that make these computations very fast. [19] For example, the Bellman update formula in the value iteration algorithm requires one matrix operation for all states, and with NumPy's efficient array operations, the computation speed can be greatly improved.

SciPy: SciPy is an advanced library for scientific computation that contains a number of mathematical functions and optimization algorithms. When implementing some algorithms in MDP, SciPy provides advanced numerical solutions such as linear algebra solving, optimization, interpolation and other functions. For large-scale MDP problems, SciPy's optimization module can help us improve the efficiency of solving and reduce the computational complexity.

Pandas: Pandas is a powerful data analysis and manipulation library, especially good at dealing with tabular data and time series data. [20] When modeling MDP problems, we often need to manipulate multiple variables or matrices. Pandas provides a powerful data structure, DataFrame, to make data storage, manipulation and analysis more efficient and intuitive.

Matplotlib and Seaborn: These two libraries are used for data visualization. During the experiment, we need to visualize the convergence process of the algorithm, the evolution of the strategy, etc. Matplotlib is the most commonly used drawing library that can help us draw various types of graphs, while Seaborn is an

advanced interface based on Matplotlib that provides more aesthetically pleasing tools for statistical graphs and data visualization. Through these two libraries, we can analyze the performance of algorithms more intuitively, especially in the presentation of experimental results, and they provide us with rich visualization capabilities.

TensorFlow and PyTorch: Deep learning frameworks become the key to solving problems when it comes to dealing with large, high-dimensional state Spaces. TensorFlow and PyTorch are currently the most popular deep learning frameworks that support the training and reasoning of neural network models. [21] The use of neural networks is inevitable in algorithms such as DQN (Deep Q-learning), and these two frameworks provide efficient support for reinforcement learning algorithms such as deep Q-learning. Not only do they support GPU acceleration, but they can also easily handle large data sets and complex models.

Deep learning and reinforcement learning frameworks

For MDP extensions that require higher precision or deal with complex problems (reinforcement learning problems such as DQN), we need to use a deep learning framework. TensorFlow and PyTorch are both very popular deep learning frameworks with powerful computational graphs and model training capabilities. We will choose one of these frameworks for the implementation of deep Q-learning.

TensorFlow: TensorFlow is an open source machine learning framework developed by Google that is widely used for deep learning tasks. It supports efficient numerical computation and is capable of accelerated training on cpus and

Gpus. TensorFlow provides high-level apis (such as Keras) for building and training neural networks, as well as low-level apis for more flexible operations.

PyTorch: PyTorch is an open source deep learning framework provided by Facebook. It features dynamic computation graphs and is more flexible than TensorFlow, making it suitable for research and experimentation. PyTorch's ease of use and the advantages of dynamic graphs make it quickly widely used in the scientific research field. For reinforcement learning applications that require algorithms such as deep Q-learning, PyTorch provides greater flexibility and efficiency.

Efficient algorithms and parallelization

As the size of the problem increases, the computational complexity of the MDP usually increases. In order to improve the efficiency of the algorithm, especially in the face of large-scale state space and action space, it is very important to adopt efficient algorithms and parallel computation. Here are some optimization strategies:

Parallel computing vs. Multithreading: For parallel updates of states and actions, multithreading or multiprocess techniques can be used. With Python's multiprocessing and concurrent.futures libraries, we can compute in parallel on multiple CPU cores, improving the computational efficiency of large-scale MDP solutions. For example, in Q-learning or deep Q-learning, Q-value updates for each state-action pair can be performed in parallel in different threads, thereby reducing training time.

Distributed computing: For larger scale MDP problems, the use of a

distributed computing framework such as Apache Spark or Dask can further improve computing power. [22] With a distributed system, multiple computing nodes can calculate and store data together, which can greatly reduce the solving time of large-scale state space and action space problems, especially when performing deep reinforcement learning, distributed training can accelerate the training process of neural networks.

Hardware acceleration (GPU/TPU) : When training deep learning models, hardware acceleration is the key to improving efficiency. Both TensorFlow and PyTorch support GPU acceleration and are able to greatly accelerate matrix operations, especially when working with high-dimensional, large-scale data. By using NVIDIA Gpus (such as the GTX/RTX series) or Tensor-processing units (Tpus) provided by Google Cloud, we can get significant performance gains on deep Q-learning algorithms in reinforcement learning.

Version control vs. code management

Version control is an important tool for managing code during multi-team collaboration or long-term project development. We use Git and GitHub/GitLab as code management and version control tools. [23] With Git, developers can easily commit code, branch management and version rollback, ensuring the consistency and traceability of the project's code across different developers. GitHub and GitLab provide code hosting, collaborative development, and CI/CD (continuous integration/continuous delivery) capabilities, which can improve the efficiency of team development and the quality of code.

In this section, we discuss the programming environment and technology

selection of Markov decision process (MDP) algorithm in detail. By choosing Python as the main programming language and utilizing powerful tools and frameworks such as NumPy, SciPy, TensorFlow, PyTorch, etc., we are able to efficiently implement MDP algorithms and deal with complex and large-scale state space problems. At the same time, through optimization methods such as parallel computing and hardware acceleration, we can significantly improve the efficiency of the algorithm and ensure that MDP problems can be solved quickly and stably in different application scenarios. These technology selections have laid a solid foundation for the subsequent algorithm implementation and performance optimization.

2.2 Implementation details of MDP algorithm

In this section, we will discuss in detail how to programmatically implement several classical algorithms for Markov decision processes (MDP), including Value Iteration, Policy Iteration, and Q-learning. These algorithms are the foundation of solving MDP problems, and we will introduce their implementation process and key code one by one.

Implementation of the value iterative algorithm: The value iterative algorithm is a dynamic programming method that gradually approximates the optimal strategy by repeatedly updating the value of each state. [24] At the heart of value iteration is the Bellman optimal equation, which computes the maximum expected return for each state by recursively. The key step of the algorithm is to update the state value function and derive the optimal strategy based on the value function.

The steps of the value iteration algorithm

Initialize state values: First, we initialize a value V(s)V(s)V(s) for each state sss, usually the initial value can be set to zero or a random value.

Bellman update: For each state sss, we use the Bellman equation to update its value. To wit:

V (s) = Max a Σ s' P (s' | s, a)/R (s, a) + (' s)] gamma V V (s) = \ max_a \ sum_ {' s} \ | s, a) P (s' left [R (s, A) + (' s) \ \ gamma V V (s) = amax right] 's Σ P (s' | s, a)/R(s, a) + gamma V (' s)]

Among them, P (s' | s, a) P (s' | s, a) P (s' | s, a) is the state transition probability, R (s, a) R (s, a) R (s, a) is a reward function, gamma \ gamma gamma is the discount factor, V (' s) V (s') V (' s) is the value of the next state.

Convergence decision: Update state values repeatedly until the change in value of all states is less than a preset threshold (e.g. ϵ\epsilonϵ).

```
import numpy as np

def policy_iteration(states, actions, P, R, gamma=0.9, epsilon=1e-6):
    # Randomly initialize policy and status values
    policy = np.random.choice(actions, len(states))  # Initialize to random policy
    V = np.zeros(len(states))  # Initialize the status value to zero

    while True:
        # Policy evaluation: Calculates the value of each state under the current policy
        while True:
            delta = 0
            for s in states:
                v = V[s]
                a = policy[s]
```

```
                # Calculate the value of V(s) based on action a of the current policy
                V[s] = sum(P[s][a][s_next] * (R[s][a] + gamma * V[s_next]) for s_next in states)
                delta = max(delta, abs(v - V[s]))
            # Stop policy evaluation when the value change is less than the threshold
            if delta < epsilon:
                break

        # Policy Improvement: Update the policy so that each state takes the action that gets the most benefit
        policy_stable = True
        for s in states:
            old_action = policy[s]
            # Find the action that maximizes the expected payoff
            policy[s] = np.argmax([sum(P[s][a][s_next] * (R[s][a] + gamma * V[s_next])
                                       for s_next in states) for a in actions])
            # If the policy changes, the flag policy is unstable
            if old_action != policy[s]:
                policy_stable = False

        # If the policy does not change, it indicates convergence and ends the loop
        if policy_stable:
            break

    return policy, V
```

The implementation first initializes the value of all states to zero, and then iteratively updates through Bellman's equation. After convergence, the optimal state value and strategy are output.

Implementation of policy iteration algorithm : Policy iteration is a method to solve the optimal policy by alternating policy evaluation and policy improvement.

[25] In each policy evaluation phase, we calculate the value of each state under a given policy; In the strategy improvement phase, we update the strategy based on the current state value until the strategy is stable.

Steps of the policy iteration algorithm

1. Initializing policies and state values : Initializes a random policy for each state with an initial value of zero.

2. Policy evaluation : Given the current policy, calculate the value of each state through the Bellman equation:

V PI (s) = Σ 's P (s' | s, PI (s)/R (s, PI (s)) + PI (' s)] gamma V V ^ \ PI (s) = P (s \ sum_ {' s}' | s \ \ left PI (s))/R (s, / PI (s)) + / gamma V ^ \] V PI PI (' s) \ right (s) = 's Σ P (s' | s, PI (s)/R (s, PI (s)) + gamma V PI (' s)]

3. Strategy improvement : Select the best action according to the current state value and update the strategy:

PI '(s) = arg Max a Σ s' P (s' | s, a)/R (s, a) + gamma V PI (' s)] \' PI (s) = \ arg \ max_a \ sum_ {' s} \ | s, a) P (s' left [R (s, A) + \ gamma V ^ \ \ PI (s') right] PI '(s) = argamax' s Σ P (s' | s, a)/R (s, a) + gamma V PI ('s)]

4. Convergence judgment : If there is no change in strategy, then the iteration is stopped.

```
import numpy as np

def policy_iteration(states, actions, P, R, gamma=0.9, epsilon=1e-6):
    # Randomly initialize policy and status values
    policy = np.random.choice(actions, len(states))  # Initialize to random policy
    V = np.zeros(len(states))  # Initialize the status value to zero
```

```
    while True:
        # Policy evaluation: Calculates the value of each state under the current policy
        while True:
            delta = 0
            for s in states:
                v = V[s]
                a = policy[s]
                # Calculate the value of V(s) based on action a of the current policy
                V[s] = sum(P[s][a][s_next] * (R[s][a] + gamma * V[s_next]) for s_next in states)
                delta = max(delta, abs(v - V[s]))
            # Stop policy evaluation when the value change is less than the threshold
            if delta < epsilon:
                break

        # Policy Improvement: Update the policy so that each state takes the action that gets the most
benefit
        policy_stable = True
        for s in states:
            old_action = policy[s]
            # Find the action that maximizes the expected payoff
            policy[s] = np.argmax([sum(P[s][a][s_next] * (R[s][a] + gamma * V[s_next])
                                       for s_next in states) for a in actions])
            # If the policy changes, the flag policy is unstable
            if old_action != policy[s]:
                policy_stable = False

        # If the policy does not change, it indicates convergence and ends the loop
        if policy_stable:
            break

    return policy, V
```

This implementation updates the policy by alternating policy evaluation and policy improvement until the policy no longer changes.

Q-learning algorithm Implementation: Q-Learning is a value-based reinforcement learning algorithm that aims to optimize policies by learning the state-action value function Q(s,a)Q(s, a)Q(s,a) Q(s,a). Q-learning is model-free, it does not need to know the transition probability and reward function of the environment, but learns through interaction with the environment.

Steps of Q-learning algorithm

Initialize Q values: Initialize all state - action pairs with Q values of zero or small random values.

Interacting with the environment: At each step, the agent selects an action aaa based on the current state sss and the ε -greedy policy.

Update the Q value: Update the Q value based on the reward R(s,a)R(s, a)R(s,a) and the maximum Q value of the next state:

Q (s, a) please Q (s, a) + alpha [R (s, a) + gamma Max a 'Q (s' a') (s, a)] - Q Q (s, a) (s, a) + \ \ leftarrow Q alpha \ left [R (s, A) + \ gamma \ max_ {a} 'Q (s)', a '- Q (s, a)] \ right Q (s, a) please Q (s, a) + alpha [R + gamma (s, a) a' Max Q (s' a ') - Q (s, a)]

Where, α \alpha α is the learning rate and γ \gamma gamma is the discount factor. 4 Convergence judgment: Repeatedly update the Q value until it converges.

Python implementation of Q-learning

```
import numpy as np
```

```
def policy_iteration(states, actions, P, R, gamma=0.9, epsilon=1e-6):
    # Randomly initialize policy and status values
    policy = np.random.choice(actions, len(states))    # Initialize to random policy
    V = np.zeros(len(states))    # Initialize the status value to zero

    while True:
        # Policy evaluation: Calculates the value of each state under the current policy
        while True:
            delta = 0
            for s in states:
                v = V[s]
                a = policy[s]
                # Calculate the value of V(s) based on action a of the current policy
                V[s] = sum(P[s][a][s_next] * (R[s][a] + gamma * V[s_next]) for s_next in states)
                delta = max(delta, abs(v - V[s]))
            # Stop policy evaluation when the value change is less than the threshold
            if delta < epsilon:
                break

        # Policy Improvement: Update the policy so that each state takes the action that gets the most
benefit
        policy_stable = True
        for s in states:
            old_action = policy[s]
            # Find the action that maximizes the expected payoff
            policy[s] = np.argmax([sum(P[s][a][s_next] * (R[s][a] + gamma * V[s_next])
                                       for s_next in states) for a in actions])
            # If the policy changes, the flag policy is unstable
            if old_action != policy[s]:
                policy_stable = False
```

```
        # If the policy does not change, it indicates convergence and ends the loop
        if policy_stable:
            break

    return policy, V
```

Q-learning learns optimal strategies by repeatedly updating Q values, using ε-greedy strategies in selecting the optimal action at each step, balancing exploration and exploitation. Q-learning's strength is its ability to deal with complex, not fully known environments.

Deep Q-learning (DQN) implementation: When the state and action space is very large or continuous, the traditional Q-learning will be limited by storage space and computing resources. [26] In order to solve this problem, deep Q-learning (DQN) introduces deep neural network to approximate Q-value function, so that Q-learning can deal with high-dimensional state space problems.

Implementation steps of DQN

Neural network construction: Use neural networks to approximate Q-valued functions. The input to the network is the state and the output is the Q value for each possible action.

Experiential playback: In order to improve learning stability, DQN uses experiential playback technology, in which the agent stores the experience of each interaction with the environment (state, action, reward, next state) into an experience pool. [27] At the time of update, a small batch of experiences is randomly selected for training.

Target network: Introduce a target network to stabilize the learning process.

The target network is used to calculate the target Q value and is updated regularly.

Python implementation framework for DQN

```
import numpy as np


def policy_iteration(states, actions, P, R, gamma=0.9, epsilon=1e-6):
    # Randomly initialize policy and status values
    policy = np.random.choice(actions, len(states))  # Initialize to random policy
    V = np.zeros(len(states))  # Initialize the status value to zero

    while True:
        # Policy evaluation: Calculates the value of each state under the current policy
        while True:
            delta = 0
            for s in states:
                v = V[s]
                a = policy[s]
                # Calculate the value of V(s) based on action a of the current policy
                V[s] = sum(P[s][a][s_next] * (R[s][a] + gamma * V[s_next]) for s_next in states)
                delta = max(delta, abs(v - V[s]))
            # Stop policy evaluation when the value change is less than the threshold
            if delta < epsilon:
                break

        # Policy Improvement: Update the policy so that each state takes the action that gets the most benefit
        policy_stable = True
        for s in states:
            old_action = policy[s]
            # Find the action that maximizes the expected payoff
            policy[s] = np.argmax([sum(P[s][a][s_next] * (R[s][a] + gamma * V[s_next])
```

```
                                        for s_next in states) for a in actions])
            # If the policy changes, the flag policy is unstable
            if old_action != policy[s]:
                policy_stable = False

        # If the policy does not change, it indicates convergence and ends the loop
        if policy_stable:
            break

    return policy, V
```

In DQN, the neural network learns Q values by backpropagation, and the input to the network is the state and the output is the Q value for each action. The update strategy of the target network is somewhat different from that of the behavioral network to ensure the stability of the learning. In this section, we introduce the specific implementation of MDP algorithm in detail, covering the classic value iteration, policy iteration, Q-learning and the best deep Q-learning (DQN) algorithm. Each algorithm is provided with detailed Python implementation code, and the key steps and implementation details of each algorithm are explained. [28] Through the implementation of these algorithms, we are able to effectively solve Markov decision process problems, providing a solid foundation for subsequent experiments and applications.

2.3 Performance evaluation and optimization

After implementing an algorithm for the Markov decision process (MDP), it is critical to evaluate its performance. [29] Performance evaluation not only helps us understand the performance of the algorithm under different conditions, but also

provides data support for optimization. The optimization process mainly includes improving the convergence speed of the algorithm, improving the quality of the strategy, and reducing the consumption of computing resources. In this part, we will introduce in detail how to evaluate the performance of MDP algorithm and explore the optimization methods.

Main performance evaluation indicators: The purpose of performance evaluation is to measure the performance of MDP algorithm in different application scenarios. The main evaluation indicators include:

Convergence speed: The time and number of iterations required to evaluate the algorithm to reach the optimal strategy during the iteration process. For dynamic programming methods such as value iteration and policy iteration, the convergence speed directly affects the practical application of the algorithm on large-scale problems. Policy quality: Evaluate whether the strategy output by the algorithm is effective, that is, whether the strategy can maximize long-term returns. Strategy quality is often measured by comparing the payoff of the actual strategy to the payoff of the optimal strategy. In reinforcement learning algorithms such as Q-learning and deep Q-learning, policy quality is often assessed by rewards accumulated during training. Computing resource consumption: The time, memory, and computing resources required to evaluate the algorithm during computation. As the state space and action space increase, the computational requirements of the MDP algorithm can rise rapidly. [30] Therefore, reducing the consumption of computing resources is an important direction to improve the efficiency of the algorithm. Stability and robustness: Evaluate the performance of the algorithm in

the face of different initial conditions, randomness, and external perturbations. For reinforcement learning algorithms, such as Q-learning and deep Q-learning, the stability of the algorithm is particularly important because these algorithms rely on interactive learning with the environment and can be affected by external noise.

Evaluation methods: For the evaluation of MDP algorithms, the following methods are commonly used:

Experimental verification: Test the performance of the algorithm in practical problems by designing experimental scenarios. These experiments can be performed with different state Spaces, reward structures, and strategy evaluation criteria. For example, in robot navigation, experiments can look at how algorithms perform under different obstacle layouts; In investment problems, historical data can be used to simulate the effectiveness of the algorithm's decision making under different market conditions.

Convergence analysis: For dynamic programming methods such as value iteration and strategy iteration, performance can be evaluated by analyzing how quickly the algorithm converges. Often, the rate of convergence can be measured by looking at changes in state values or policies. The faster the convergence, the less computational resources and time the algorithm needs to find the optimal policy.

Experimental comparison: Compare the performance of different algorithms to analyze their strengths and weaknesses on the same problem. For example, you can compare the performance of value iteration and strategy iteration with Q-learning and deep Q-learning in the same scenario to observe their differences in

convergence speed, strategy quality and computing resource consumption.

Test different parameter configurations: for reinforcement learning algorithms, it is possible to observe their effects on the performance of the algorithm by changing hyperparameters such as learning rate, discount factor, exploration rate (ε in ε-greedy strategy). [31] This helps to understand the sensitivity of the algorithm to the parameters and provides a basis for selecting the best parameter configuration.

Example: Performance evaluation of Q-learning

Suppose we use the Q-learning algorithm to solve a simple grid world problem. We can evaluate the performance of Q-learning by the following metrics: Cumulative rewards during training: During training, the rewards of each iteration can be used to measure whether the agent gradually tends to the optimal strategy during the learning process. A higher cumulative reward usually indicates that the strategy has been effectively learned and improved. Learning curve: The convergence of the algorithm can be intuitively understood by plotting the relationship between cumulative rewards and the number of iterations during training. A smooth rise in the learning curve usually indicates that the algorithm is gradually finding an optimal strategy. Compare with random strategies: We can compare the results of Q-learning with strategies that randomly select actions. A random strategy should have a lower cumulative reward, while a Q-learning strategy should be able to get a higher cumulative reward by choosing more appropriate actions.

Example: A performance review for Deep Q-learning

For Deep Q-learning (DQN), a performance review may involve the following:

Q value convergence: During the training process, the output of the neural network, the Q value, will gradually converge to the optimal value function. The training progress of deep Q-learning can be evaluated by drawing the curve of Q value change. If the Q value gradually becomes stable, it indicates that the algorithm is converging. Reward curve: The performance of deep Q-learning can be evaluated by a reward curve. We can plot the cumulative reward for each episode during the training to see if the agent is able to progressively earn higher rewards as the training progresses. [32] Training duration and stability: DQNS typically require longer training times and larger computational resources, especially in high-dimensional state Spaces and complex environments. Evaluating the training duration and stability of DQN is helpful to judge the applicability of the algorithm in practical applications.

Optimization method: In the implementation of MDP algorithm, we may face some problems such as high computational complexity, slow convergence speed and unstable algorithm. Therefore, optimization is a key step to improve the performance of MDP algorithm. Here are several commonly used optimization methods:

Reduce computational complexity; In large scale state space and action space, the computation amount of MDP algorithm will increase dramatically. In order to improve the computational efficiency, we can adopt the following strategies: Function approximation: For high-dimensional state Spaces, the function

approximation method can be used to estimate state-action value functions. Common functional approximation methods include linear regression, radial basis function (RBF), and neural network. By using neural networks as an approximation of the Q function, we can effectively deal with continuous and high-dimensional state Spaces. State space simplification: In real problems, the state space can be very large, and the amount of computation can be effectively reduced by abstracting, clustering, or dimensionality reduction of the state space (such as PCA). State clustering merges states with similar properties into one group, thereby reducing the size of the state space.

Increase the speed of convergence; The convergence speed is the key factor affecting the efficiency of MDP algorithm. The following methods can help improve convergence speed: Learning rate adjustment: In Q-learning and other reinforcement learning algorithms, the learning rate (\(\alpha\)) determines the number of steps for each Q-value update. By dynamically adjusting the learning rate (for example, gradually decreasing as the number of iterations increases), convergence can be accelerated. Experiential replay: In deep Q-learning, experiential replay is an important optimization technique. By storing past experiences in an experience pool and randomly drawing training samples from it, correlation of data is avoided, thus improving the stability and speed of convergence of training. Target Network: Deep Q-learning uses a target network to stabilize the learning process. By updating the target network on a regular basis, rather than every iteration, shocks during training can be reduced and convergence accelerated.

Enhance the stability of the algorithm; In reinforcement learning algorithms, especially in deep Q-learning, algorithmic instability is often a challenge. The following can help improve the stability of the algorithm: Gradient clipping: In deep Q-learning, too many gradients can lead to unstable training. By gradient clipping technique, the maximum value of the gradient can be limited and the problem of gradient explosion can be avoided, thus improving the stability of the algorithm. [33] Delayed update: In DQN, the update of experience playback and target network is usually delayed, that is, the target network is updated after a certain number of iterations. This can reduce the noise in the training process and improve the stability of the algorithm.

Parallel computing and distributed computing; In order to improve computing efficiency and shorten training time, MDP algorithm can be optimized by parallel computing and distributed computing technology: Parallel Q-value update: for algorithms such as Q-learning, the calculation can be accelerated by parallel updating the Q values of multiple state-action pairs. [34] Parallel computing can take full advantage of multi-core processors and significantly improve the training speed. Distributed training: In reinforcement learning, distributed training can distribute the training process across multiple computing nodes, especially in environments requiring large-scale computing, which can significantly improve performance.

CHAPTER THREE APPLICATION CASE DESIGN AND RESULT ANALYSIS

In this chapter, we will design specific application cases to verify the effectiveness of the aforementioned MDP algorithm, show the performance of different algorithms in actual scenarios, and analyze the results. Through these application cases, we can not only verify the theoretical correctness of MDP algorithm, but also reveal its application potential and limitations in complex environments.

3.1 Select specific application scenarios

In this section, we will take two typical application scenarios - robot navigation and financial investment - and explore in depth how they can be modeled and solved using the Markov Decision Process (MDP) model. Through the analysis of these two scenarios, we can show the wide applicability of MDP algorithm in different industries, especially when facing decision-making problems with dynamic changes and uncertainties.

Robot navigation

Background: The robot navigation problem is a classic decision problem, usually involving an agent (i.e., a robot) moving autonomously in an environment with the goal of getting from the starting position to the target position and avoiding obstacles as much as possible. Navigation problems have a wide range of applications in many fields, including autonomous driving, service robots, drones,

robot competitions, etc.

Problem Description: In robot navigation, the robot usually needs to make decisions based on sensor information in a known or unknown environment and choose the best moving path. In this scenario, MDP can effectively model every state, action, reward, state transition and other elements faced by the robot.

State space SSS: The state space represents all possible positions the robot can be in. In a two-dimensional grid world, states can be represented by coordinates. For example, suppose the environment is a 5×55 \times 55×5 grid, and the position on each grid of the robot is a state, for a total of 25 states.

Action space AAA: The actions the robot can take in each state. For example, the robot can select four basic actions in the grid: "Move up," "Move down," "Move left," and "move right." Each action corresponds to a shift in one state.

Transition probability P(s' ∣ s,a)P(s' |s, a) : The transition probability represents the probability that the robot will move to the next state s's 'after given the current state sss and the action aaa taken. In an ideal case, this process is deterministic, i.e. the robot will move to the new state with certainty for every state and action. In practical applications, however, there may be noise that causes the robot to not always reach the target position precisely when performing the action.

Reward function R(s,a)R(s, a)R(s,a) : The reward function is used to evaluate the effect of each state-action combination of the robot. For example, success in reaching the target position can give a reward of +100+100+100, while hitting an obstacle can give a penalty of -100-100-100. In other states, the robot may receive a smaller penalty or negative reward (such as -1-1-1 given for each move to

encourage reaching the target quickly).

MDP Modeling: When translating the robot navigation problem into an MDP model, the robot selects an action in each state, moves to the new state based on transition probability, and evaluates the effect of the current action based on the reward function. [35] The goal is to maximize the cumulative reward from the starting point to the goal by selecting the optimal strategy, i.e., the best action in each state.

State space size: If the robot moves in a 5×55 \times 55 ×5 grid, there are 25 positions in total, that is, the state space size is 25.

Reward function design: A relatively simple reward structure can be designed, such as the reward of reaching the target position is +100, the penalty of bumping into an obstacle is -100, and the penalty of each move is -1.

Apply MDP algorithm to solve the robot navigation problem: In robot navigation problems, we can use several MDP algorithms to find the optimal strategy: Value iteration: With the value iteration algorithm, we can calculate the value of each location and, by updating the status value, find the shortest path from the starting point to the goal. Strategy iteration: Through alternating iterations of policy evaluation and policy improvement, the strategy iteration algorithm can also find the optimal path.

Q-learning: In practical applications, Q-learning can help robots learn the optimal path step by step in their interactions with the environment. Q-learning does not need an environment model, and constantly optimizes strategies through interaction with the environment.

Through the application of these algorithms, the robot can autonomously learn the optimal navigation path in a dynamic or complex environment, avoid obstacles, and finally reach the target.

Financial investment

Background: Financial investing is another classic decision problem where the investor's goal is to maximize investment returns by making sound decisions in an uncertain market. Decision-making in financial investment often involves the allocation and management of a number of assets, while the price fluctuations of the market and economic factors are highly uncertain. MDP provides a theoretical framework for decision making in financial investment.

Problem Description: In a financial investment problem, an investor needs to make a choice among several possible market states, including buying, selling, or holding an asset. The price of an asset is affected by market fluctuations and external factors, so the state space and action space are highly dynamic and random. [36] The MDP model can help investors make optimal decisions in each state, thus maximizing the long-term return of the investment portfolio.

State space SSS: The state space describes the investor's portfolio and market conditions at a given point in time. For example, in the stock market, the state can include the current amount of money, the stock price, the market index, etc.

Action space AAA: Investors can choose actions including "buy", "sell" and "hold" three options. Investors can choose one move at each moment based on current market conditions.

Transition probability P(s' ｜ s,a)P(s' |s, a)P(s' ｜ s,a) : The changes in the

market are random, so the transition of state depends on market volatility. The transition probability reflects how the investor's assets and the market state change from the current state and action.

Reward function R(s,a)R(s, a)R(s,a) : The reward function is usually defined by the investor's return on assets. For example, after buying a stock, an investor receives a positive reward if the stock price rises; If the stock price goes down, they get a negative reward. The reward can be expressed as the return of the portfolio and is usually calculated over time periods.

MDP Modeling: To translate a financial investment problem into an MDP, the investor needs to make a decision each time period to choose whether to buy, sell, or hold the asset. By accumulating the rewards for each decision, the investor can get the maximum return. The MDP model can help investors make optimal decisions in each state, thereby optimizing the portfolio.

State space: can include variables such as current funds, number of stocks held, market index, etc. If the market has multiple stocks, the dimension of the state space will increase rapidly.

Reward function: Rewards are usually based on the return of the portfolio and can be defined as the change in the value of the asset. For example, a positive reward is given for a high return and a negative reward for a loss.

Apply MDP algorithm to solve financial investment problem:

In financial investment problems, we can similarly use several MDP algorithms to solve the optimal investment strategy: Value iteration: The value iteration algorithm can help investors calculate the value in each market state, so as

to make investment decisions by selecting the action with the greatest value. Strategy iteration: Strategy iteration allows investors to adjust their investment decisions based on the current state value by alternately evaluating and improving the strategy.

Q-learning: By learning state-action value function (Q-value), Q-learning algorithm enables investors to gradually learn the optimal investment strategy through interaction with the environment in the market changes.

3.2 Show how to apply MDP algorithm to solve practical problems

In this section, we will analyze in depth how to apply MDP algorithm to solve real problems, focusing on how to use MDP algorithm (including Q-learning, value iteration, strategy iteration, and deep Q-learning) in two scenarios of robot navigation and financial investment. [37] Through specific experimental design and result analysis, the performance and advantages and disadvantages of these algorithms in practical applications will be demonstrated. Robot navigation Case

Experiment design:

We model the robot navigation problem as an MDP problem in which the robot needs to move from the starting point of the grid world to the target position and avoid obstacles. The key to this task is how to formulate the optimal path in an environment with uncertainty.

State space SSS: We set up a 5×55 \times 55×5 grid world, which contains a starting position, a target position and several obstacles. Each grid represents one state, so the total state space contains 25 states.

Action space AAA: The robot can choose four basic actions: move up, down, left, and right. Each action will cause the robot to move from one state to another.

Transition probability P(s' ｜ s,a)P(s' |s, a)P(s' ｜ s,a) : In the ideal case, the robot will reach the target position precisely when performing an action. However, in order to increase the uncertainty, we assume that each action will have a certain probability of deviation, i.e. the robot may not be able to move exactly in the expected direction. Reward function R(s,a)R(s, a)R(s,a) : gives a reward of +100 when the target position is reached. A penalty of -100 is given when hitting an obstacle. The reward for each move is -1, encouraging the robot to get to the target position as quickly as possible.

Apply the MDP algorithm: Value iteration:

The value iteration algorithm can find the optimal path by repeatedly updating the value of each state. In each iteration, the robot chooses the action that maximizes the expected reward at each position. [38] Experimental results show that value iteration can converge quickly and perform very well in small-scale grid worlds. However, with the increase of environmental complexity, the amount of computation will increase significantly, resulting in the decrease of algorithm efficiency.

Experimental results:

In the 5×55 \times 55×5 grid world, the calculation speed is fast, and the shortest path can be found after several iterations. With the increase of the scale of the grid world, the computing time and memory requirements of value iteration increase exponentially, which shows greater computational complexity.

Q-learning: In Q-learning, the robot learns the optimal path through interaction with the environment. Since Q-learning does not rely on an exact model of the environment, it can progressively update the state-action value function Q(s,a)Q(s, a)Q(s,a) as it is executed, and without prior knowledge of the transition probability and reward function.

Experimental results:

On the randomly initialized Q value table, the robot gradually converges to the optimal path through multiple interactions with the environment. Although Q-learning has a slower convergence rate, it has significant advantages when faced with dynamic changes or unknown environments. In larger grid worlds (such as 10 × 1010 \times 1010 × 10 or larger), Q-learning performs more stable because it does not rely on a complete environment model, but optimizes strategies through trial-and-error learning.

Deep Q-learning (DQN) : For more complex environments, especially when the state space is very large or continuous, traditional Q-learning may not be able to handle it. At this point, we take a deep Q-learning (DQN) approach to approximate Q-valued functions by using deep neural networks. DQN is able to efficiently handle high-dimensional state Spaces and improve stability through target networks and experiential playback techniques.

Experimental results:

In a large or complex grid world (such as an obstruction-dense environment or an environment with dynamic obstacles), DQN approximates the Q-value function by neural network and can stably learn the optimal strategy through

experiential playback over a long period of time. The training time of DQN is longer, but as the training progresses, the Q value gradually converges, and the robot is able to effectively avoid obstacles and find the optimal path.

Analysis of results:

Value iteration: Although the value iteration algorithm performs well in small-scale problems, it is very expensive to compute in large-scale grids or complex environments, requiring a lot of memory and computation time. Therefore, value iteration works well in environments where the state space is small and known.

Q-learning: Q-learning is a model-free reinforcement learning algorithm that learns optimal policies without a model of the environment. Despite its slow convergence, it is able to adapt to changing environments and is suitable for dynamic or incompletely known environments. In the medium scale grid world, Q-learning shows better stability.

Deep Q-Learning (DQN) : DQN approximates Q value through neural network, which can deal with high-dimensional and complex environment. [39] Although it takes a long time to train, it has strong generalization capabilities and is able to quickly learn optimal strategies in complex environments. DQN is particularly suitable for large-scale environments or multi-task learning.

Financial Investment Case Experimental design:

In the financial investment problem, we will simulate a simple stock market model where investors need to maximize their long-term returns by making buy, sell, or hold decisions. State space SSS: Each state represents the investor's

current amount of money and the state of the market. In a simple model, the state can be composed of the amount of cash currently held by the investor and the change in the market index. For example, the state might be the investor's cash balance along with the change in the stock market over the last three days.

Action space AAA: The investor's actions include "buy," "sell," and "hold." Each action determines how the investor's assets change.

Transfer probability P(s' ｜ s,a)P(s' |s, a) : Stock market fluctuations are random, so changes in market prices are uncertain, and the transfer probability reflects how the current asset and market state change over time.

Reward function R(s,a)R(s, a)R(s,a) : The reward is usually based on the return of the portfolio. For example, if an investor chooses to buy a stock and the stock price rises, then the action will result in a positive reward; If the stock price falls, it will bring a negative reward. The reward function can be defined as the return on assets for each time period, usually expressed as a percentage.

Apply the MDP algorithm: Value iteration:

In financial investment problems, value iteration can help investors evaluate the value of each state and guide them to choose the optimal investment strategy. By solving Bellman's equation, value iteration can calculate the optimal investment behavior in each state.

Experimental results:

In a simple market model, value iteration is able to calculate the value of each state and provide optimal decision recommendations for investors. Although convergence is faster, computational complexity is higher in high-dimensional

market models.

Strategy iteration:

Strategy iteration helps investors gradually optimize investment decisions through continuous strategy evaluation and strategy improvement. By evaluating the benefits of the current strategy and improving the strategy according to the evaluation results, the optimal investment strategy is finally found.

Results of the experiment:

Strategy iteration can effectively find the optimal strategy in small scale problems, but when facing more complex market models, the computational efficiency of strategy iteration will also be affected, especially when the state space is large and the computational resource consumption is large.

Q-learning: Q-learning is a model-free reinforcement learning algorithm, which can gradually learn the value of each state-action pair through interaction with the environment, and finally optimize the investment strategy. In financial investment scenarios, Q-learning can help investors make decisions in complex markets.

Experimental results:

Q-learning can gradually improve investment returns by simulating market fluctuations for many times and constantly adjusting investment strategies. Although the convergence is slow, in the real environment, Q-learning gradually optimizes investment decisions through experiential replay and exploration-utilization strategies.

Deep Q-learning (DQN) : Deep Q-learning (DQN) approximates Q-value

functions through neural networks, which can effectively handle high-dimensional and complex market environments. In complex investment scenarios, DQN is able to better approximate strategies through deep neural networks and learn optimal decisions in large-scale data.

Experimental results:

In the multi-asset investment model, DQN can learn more efficient strategies than traditional Q-learning, especially in the face of complex markets, DQN's performance is particularly outstanding. Although the training time is longer, the final return is higher than that of the traditional method.

Result analysis: Value iteration and strategy iteration: suitable for small scale, known environment of financial markets. When the state space is small and the market behavior is certain, value iteration and strategy iteration can converge quickly and find the optimal strategy.

Q-learning: In a dynamic market environment, the model-free nature of Q-learning enables it to cope with market uncertainty and gradually learn the optimal strategy. Despite its slow convergence rate, Q-Learning is adaptable and suitable for markets that are not fully known.

Deep Q-Learning (DQN) : DQN approximates Q value through neural network, and can find more accurate strategies in high-dimensional and complex environments, especially in multi-asset investment, complex market fluctuations and other scenarios, showing a strong ability.

In this section, we show in detail how to apply MDP algorithm to two typical practical problems of robot navigation and financial investment. Through the

application and experimental analysis of different algorithms (value iteration, strategy iteration, Q-learning and deep Q-learning), we find that each algorithm has its advantages and limitations in different application scenarios. [40] In robot navigation, Q-learning and deep Q-learning are especially suitable for dynamic environments, while in financial investment, deep Q-learning shows strong adaptability and can handle high-dimensional complex market environments.

3.3 Analysis of Results

In this section, we will conduct an in-depth analysis of the experimental results of the above two application cases. By comparing the performance of different MDP algorithms in robot navigation and financial investment scenarios, we can reveal their advantages and disadvantages, and explore the application potential of these algorithms in practical problems. The focus of the analysis includes convergence, strategy quality, computational efficiency and adaptability to help us choose the algorithms that are most suitable for different application scenarios.

Analysis of robot navigation results

In the experiment of robot navigation, we use different scale grid world environments to compare the performance of each MDP algorithm. The complexity of the environment and the size of the state space have important effects on the convergence speed, computational efficiency and strategy quality of the algorithm.

The performance of value iteration algorithm:

Convergence: In small-scale grids (such as 5×55 \times 55 × 5 grids), the

value iterative algorithm performs well. Because value iteration quickly converges to an optimal solution by repeatedly updating the state value, it is efficient at finding the shortest path in these environments.

Computational efficiency: For small-scale grids, value iteration is able to calculate the optimal strategy in a relatively short time. However, as the size of the grid increases, the state space and the amount of computation will increase rapidly, resulting in a significant increase in computation time and memory requirements. In large-scale grids (such as 10×1010 \times 1010×10 or larger), the computational complexity of value iterations increases exponentially.

Quality of policy: The value iteration algorithm is able to calculate the optimal value for each state and determine the optimal path by selecting the action with the most value. In a simple grid world, value iteration is able to find the shortest path from the starting point to the goal and avoid obstacles.

Q-learning algorithm performance:

Convergence: The learning speed of Q-learning is slow in the initial stage, because it constantly updates the Q value through interaction with the environment to find the optimal strategy. Especially in the small-scale grid world, Q-learning gradually converges to the optimal strategy through multiple interactions with the environment.

Computational efficiency: Q-learning converges more slowly than value iteration. At each iteration, Q-Learning needs to interact with the environment, updating the state-action value function Q(s,a)Q(s, a)Q(s,a). Q-learning performs more consistently in a dynamically changing environment because it does not rely

on an exact environment model.

Quality of strategy: Although Q-learning gradually converges to the optimal strategy over a longer training period, it can adapt well to the changing environment. Even under dynamic obstacles or uncertain environments, Q-learning can gradually find the optimal path through trial and error.

Deep Q-learning (DQN) performance:

Convergence: In large-scale or complex environments, deep Q-learning (DQN) approximated Q-valued functions by neural networks, which can better handle high-dimensional state Spaces. [41] Although the training time of DQN is longer, the Q value gradually converges with the progress of training, and the robot can quickly learn the optimal path.

Computational efficiency: The computational overhead of DQN is large, especially in complex environments, and the training process may require hundreds to thousands of training cycles to achieve good results. However, DQN can deal with more complex environment and continuous state space, and has strong adaptability.

Policy quality: DQN is able to approach policies through neural networks, thus solving decision-making problems in high-dimensional, complex environments. Compared with Q-learning, DQN performs better, especially when the environment scale expands, obstacles increase or state space is continuous, DQN can find a better path.

Summary of the results:

In small-scale grid worlds (such as 5×55 \times 55×5 or 6 × 66 \times 66

× 6 grid), value iteration performs better because of its fast computation speed and the ability to quickly find the optimal path. However, in large scale or complex environments, the computational overhead of value iteration can become significant. Q-learning does not perform as well as value iteration in smaller grid worlds, but it is more adaptable and can gradually learn optimal policies through interaction with the environment. In dynamic or incomplete known environments, Q-learning is able to cope effectively with uncertainty. Deep Q-learning (DQN) performs well in large-scale, complex environments and is able to efficiently handle high-dimensional state Spaces. Despite its long training time, its adaptability and ability to generalize make it ideal for dealing with complex problems.

Analysis of financial investment outcomes

In our experiment with financial investing, we construct a simplified market model where investors need to make decisions between buying, selling, and holding in order to maximize the long-term return of their portfolio. We used Q-learning, deep Q-learning, and value iteration algorithms to learn investment strategies and compare them.

The performance of the value iterative algorithm:

Convergence: In a simple market model, value iteration is able to efficiently calculate the value of each state and select the appropriate investment strategy based on the optimal value. Value iteration can quickly converge to the optimal strategy, but its adaptability to large-scale, complex markets is limited.

Computational efficiency: Value iteration has high computational efficiency,

especially in small-scale markets, and can solve the optimal strategy in a short time. However, when the state space is greatly increased, the amount of computation and memory demand will also increase rapidly, making value iteration not suitable for large-scale or dynamically changing market environment.

Quality of strategy: In a small, known market environment, value iteration can effectively calculate the optimal strategy and maximize the return on investment. However, in markets with high volatility and uncertainty, value iteration may not perform as well as other reinforcement learning methods.

Q-learning algorithms perform:

Convergence: Q-learning has shown strong learning ability in the financial market, especially in the case of large market fluctuations. Although Q-learning's convergence speed is slow, it can gradually learn the optimal investment strategy through interaction with the environment.

Computing efficiency: The computing efficiency of Q-learning is low, especially when the state space is large. Every time the state-action value function is updated, Q-learning needs to interact with the environment several times, resulting in longer training time. However, Q-learning's model-free nature allows it to handle complex and dynamically changing market environments.

Quality of strategy: Q-learning is able to make decisions based on the real-time state of the market and gradually improve its strategy to maximize long-term returns. In a more complex market, Q-learning can gradually obtain higher returns through interaction with the environment.

Deep Q-learning (DQN) performance:

Convergence: DQN approximates Q-valued functions through neural networks and is able to find optimal strategies in complex market environments. Despite the long training time, DQN is able to effectively deal with high-dimensional and complex market state Spaces.

Computational efficiency: Since DQN involves deep learning training, the computational resource consumption is large, especially in the case of high-dimensional data and multi-assets, the training time is longer. However, DQN is able to enhance the stability of training and improve the quality of the final strategy through the target network and experience playback mechanism.

Strategy quality: DQN performs well in the face of dynamic market environments, is able to approximate optimal strategies through neural networks, and ADAPTS to complex market fluctuations. Compared with Q-learning, DQN is able to find more precise strategies in a higher-dimensional state space and ultimately obtain higher returns.

Summary of the results:

Value iteration: In simpler market models, value iteration can quickly find the optimal strategy. However, in the face of dynamically changing markets or complex multi-asset problems, the computational complexity and stability of value iteration can become limiting factors. Q-learning: Q-learning is capable of learning optimal investment strategies through experiential replay and interaction with the environment. Although the convergence rate is slow, it is suitable for dynamic and complex market environments and can effectively deal with market uncertainties. Deep Q-learning (DQN) : DQN has significant advantages when dealing with

high-dimensional, large-scale market data. Using deep neural networks, DQN is able to effectively approximate optimal strategies and achieve high long-term returns in complex markets. Despite the long training time, its adaptability and ability to generalize make it the strongest algorithm for financial investment decisions.

CONCLUSION

This study takes Markov decision process (MDP) model as the core, and expounds the potential of MDP in solving complex decision problems through theoretical discussion and application case analysis. Taking two representative scenarios of robot navigation and financial investment as examples, the performance of classical MDP algorithms such as value iteration, strategy iteration, Q-learning and deep Q-learning (DQN) in different environments is systematically verified. [42] The results show that MDP model, as a multi-stage decision-making tool, not only has a wide range of applicability and flexibility, but also shows the ability of decision optimization in dynamic and random environments. However, the MDP algorithm has some limitations in terms of computing resource consumption and policy convergence speed in practical applications. The following is a detailed summary of the main achievements, application significance, limitation analysis and future development direction of this research.

Main Research results

Application effectiveness of MDP algorithm in complex decision problems: This study verifies the effectiveness of MDP algorithm in solving dynamic and uncertain decision problems through two application cases of robot navigation and financial investment. No matter for static and deterministic environment, or dynamic and stochastic environment, MDP model can provide optimal guidance for the agent's behavior through state transition and reward feedback. [43] Value iteration and strategy iteration are suitable for situations where the environment

structure is relatively simple and the state transition probability is known, and can quickly converge to the optimal strategy. Q-learning and deep Q-learning, on the other hand, show stronger adaptability and robustness in complex environments, do not depend on the exact model of the environment, and are suitable for situations with dynamic changes in state and reward.

Applicability and performance analysis of different algorithms: Through experimental analysis, this study summarized the applicability and performance of different MDP algorithms in specific application scenarios, and revealed their advantages and disadvantages in different environments. The experimental results show that:

The value iteration and strategy iteration algorithms perform best in the environment with small state space, clear structure and fully known information, and are suitable for static scenes requiring rapid solution, with efficient computing performance. Because of its model-free characteristics, Q-learning algorithm is suitable for the environment that is not completely known or dynamically changing. Although the convergence speed is slow, it can still gradually optimize the strategy through continuous interaction with the environment in the case of limited resources, and has strong adaptability. Deep Q-learning (DQN) performs well in environments with high dimensional state space and drastic changes, and can handle complex environmental features with the help of nonlinear approximation ability of neural networks. Although it takes a long time to train, its strong generalization ability makes it show significant advantages in dealing with complex decision-making tasks.

Limitations and Challenges of MDP algorithm

Although MDP model shows wide application prospects in solving multi-stage decision problems, some limitations are also found in the research. High computational complexity: When the state space of value iteration and strategy iteration algorithms is large, the computational complexity increases exponentially, resulting in excessive consumption of computing resources and limiting their applicability in high-dimensional problems. Slow convergence speed: Q-learning and deep Q-learning need to explore repeatedly in the environment, and rely on a large number of training iterations to gradually converge to the optimal strategy, and the training process takes a long time. Policy instability: In a dynamic changing environment, the strategy may fluctuate, resulting in the instability of the strategy in the later training period. Especially Q-learning and DQN may deviate from the optimal solution when faced with sudden changes in the environment or noisy scenes.

Research significance and contribution

The integration of theory and application; On the basis of theoretical analysis, this research verifies the application effect of MDP algorithm in the field of robot navigation and financial investment through experiments. The MDP model provides a systematic theoretical framework for multi-stage decision making, and the research results prove that it has a good performance in the environment of uncertainty, randomness and dynamics. [44] At the same time, the experimental results provide data support and theoretical basis for the future application of MDP algorithm in other fields, and provide a reference for the solution of multi-stage

decision making problems.

It also provides a basis for the comparison and selection of multiple algorithms. This study compares the performance of value iteration, strategy iteration, Q-learning and deep Q-learning in different scenarios through experiments, and deeply discusses the advantages and disadvantages of each algorithm and its applicable scope. [45] It is found that value iteration and strategy iteration are suitable for fast solutions in small-scale, known environments, while Q-learning and deep Q-learning are more suitable for high-dimensional, dynamic and complex environments. It provides a basis for selecting the appropriate MDP algorithm in practical problems, and has practical guiding significance.

It enriches the application perspective of reinforcement learning in many fields; This study applies MDP algorithm to two different fields, namely robot navigation and financial investment, demonstrating the potential of reinforcement learning in multiple fields. The research shows that by adjusting the state space, reward structure and algorithm selection, the MDP model can be effectively applied to different industry scenarios, which provides enlightenment for further expanding the application of MDP. [46]

Reflection on the limitations of MDP algorithm in practical application

High computing cost and time consumption: value iteration and policy iteration increase rapidly when the state space is expanded, especially in large-scale environments, computing resource consumption increases significantly, which will cause hardware pressure in practical applications. Q-learning and DQN rely on a large number of environment interactions in the training process,

resulting in large time consumption. In the future practical application, it is necessary to make a balance between computing efficiency and resource consumption to ensure that good enough strategies can be obtained under the condition of limited computing resources.

The need to improve policy stability and generalization ability; In the actual environment, especially in the dynamic and complex environment, the strategy stability and generalization ability of the algorithm are particularly important. [47] Experiments show that Q-learning and DQN strategies may be unstable or even fluctuate in noisy or dynamically changing environments. Especially, when the environment changes beyond the scope of what the algorithm has learned, the strategy is likely to deviate from the optimal. Therefore, it is necessary to pay attention to the adaptability and anti-interference ability of the algorithm in practical applications in the future, and it may be necessary to introduce strategies such as multi-objective optimization and self-adaptive regulation.

The adaptability of algorithm design to application scenarios; The applicability of different MDP algorithms in different application scenarios is quite different, so the appropriate algorithm should be selected according to the specific environmental characteristics. For static environment with small state space and clear information structure, value iteration and policy iteration are more advantageous. In an uncertain and dynamic environment, Q-learning and DQN's model-free characteristics enable them to adapt to changes in the environment. [48] Therefore, when applying MDP algorithm, factors such as the state dimension, change frequency and resource limitation of the scene should be considered

comprehensively to design more targeted strategies.

Future research Directions

Optimize algorithms to improve computational efficiency and convergence speed; In the future, the computational complexity of the existing MDP algorithm can be optimized to improve its application effect in large-scale environment. [49] Specific optimization approaches include: Distributed and parallel computing: With the help of distributed computing and parallel processing technologies, computing tasks are allocated to multiple computing nodes to achieve efficient solution of large-scale state space. Model-based reinforcement learning combined with model-free methods: Reduce the number of actual interactions by combining model predictions, thereby increasing the speed of convergence, such as using model-based simulations to reduce the number of training times in real environments.

Improve the adaptability and robustness of the algorithm; In practical applications, dynamic changes and uncertainties are universal, so future research can focus on enhancing the adaptability and robustness of MDP algorithm: adaptive exploration mechanism: dynamically adjust the exploration rate (such as ε value in ε -greedy strategy), automatically adjust the balance of exploration and utilization according to environmental changes, so as to improve the adaptability of strategy in different environments. [50] Multi-objective optimization and multi-strategy fusion: For multi-objective decision-making problems, the multi-objective optimization framework can be considered, and a variety of strategies can be integrated to cope with complex decision-making needs,

and improve the robustness of the strategy. Improve the stability and generalization ability of the strategy; In terms of policy stability, we can improve the algorithm architecture (such as double DQN, A3C, Dueling DQN, etc.) and combine regularization methods to enhance the stability and generalization of the policy. In addition, explore the structured design of policy network and use a more robust neural network architecture, as shown in the figure of Neural network (GNN), to further improve the performance of the algorithm in complex scenarios.

Extend multi-agent and multi-objective optimization scenarios; The current research focuses on the optimal decision of a single agent, and the future can be extended to multi-agent cooperative decision making and multi-objective optimization. This extension is particularly important for solving scenarios that require multi-agent cooperation, such as transportation and logistics. For example, in intelligent transportation, each autonomous vehicle needs to make an independent decision, but the optimization of the overall transportation system requires the collaboration of multiple workshops. In the future, the multi-agent learning method can be introduced into the MDP framework to enhance the collaborative control ability of the algorithm.

Explore the deep integration of MDP with other AI technologies; The combination of MDP with other AI technologies will further enhance its performance in complex environments. [51] For example, combining reinforcement learning with deep learning, natural language processing (NLP), and computer vision (CV) enables MDP algorithms to optimize decisions in more perceptive-capable environments. In robotics applications, combining NLP and CV

technologies can enable MDP algorithms to make decisions based on visual information and verbal instructions, thus broadening the possibilities of MDP in interactive AI applications.

Research contributions and limitations

Research contribution: This study systematically discusses the application effects and limitations of MDP model in solving multi-stage decision problems, and reveals the applicability of different MDP algorithms in different industry scenarios through multi-algorithm comparison experiments. These research results provide theoretical support and empirical basis for the application of MDP model in a wider range of fields in the future, and help to promote the further development of multi-stage decision theory.

Limitations: This study is mainly based on the simulation environment, which may be different from the actual application scenario. In addition, the experiment mainly focuses on single-agent and single-objective optimization problems, and the multi-agent and multi-objective decision-making problems are not deeply discussed. Future research may consider verifying the algorithm performance in the real data environment, and extend the situation of multi-agent cooperative optimization to further enhance the application breadth and practical significance of the research.

Through system analysis and experimental verification, this study reveals the advantages and limitations of MDP model in complex decision-making problems, and shows the applicability and performance differences of different MDP algorithms. Future research can further optimize the computational efficiency of

the algorithm, improve the stability and adaptability of the strategy, and explore the application of MDP in more complex scenarios such as multi-agent and multi-objective optimization, so as to promote the application of MDP theory in a wider range of practical scenarios.

LIST OF REFERENCES

1. Yuan J, Huang Y, Tao T, et al. A cooperative approach for multi-robot area exploration [C]. Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, Taipei: IEEE, 2010: 1390-1395.

2. Ge Yuanyuan, Zhang Hongji, Tang Hong. Path planning Method based on Particle Swarm Optimization Algorithm Integrating Behavioral Dynamics [J]. Mechanical Science and Technology for Aerospace Engineering, 2018, 37(2): 244-249.

3. Asama H, Matsumoto A, Ishida Y. Design of an autonomous and distributed robot system: actress [C]. Proceedings of the IEEE/RSJ International Workshop on Intelligent Robots and Systems, Tsukuba: IEEE, 1989: 283-290.

4. Fukuda T, Nakagawa S. Dynamically reconfigurable robotic system [C]. Proceedings of the 1988 IEEE International Conference on Robotics and Automation, Philadelphia: IEEE, 1988: 1581-1586.

5. Fukuda T, Nakagawa S. Approach to the dynamically reconfigurable robotic system [J]. Journal of Intelligent and Robotic Systems, 1988, 1(1): 55-72.

6. Fukuda T, Nakagawa S, Kawauchi Y, et al. Structure decision method for self organising robots based on cell structures-CEBOT [C]. Proceedings of the 1989 International Conference on Robotics and Automation, Scottsdale: IEEE, 1989: 695-700

7. Alami R, Fleury S, Herrb M, et al. Multi-robot cooperation in the MARTHA project[J]. IEEE Robotics & Automation Magazine, 1998, 5(1):36-47.

8. Caloud P, Wonyun C, Latombe J, et al. Indoor automation with many mobile robots[C]. Proceedings of the EEE International Workshop on Intelligent Robots and Systems, Towards a New Frontier of Applications, Ibaraki: IEEE, 1990: 67-72.

9. KemppainenA, Haverinen J, Roning J. A distributed multi-robot sensing system using an infrared location system [M]. Angers: DBLP, 2007:280-283.

10.Jia Jianqiang, Chen Weidong, Xi Yugeng. System Design and Control Implementation of Open Autonomous Mobile Robot [J]. Journal of Shanghai Jiao Tong University, 2005(6): 905-909.

11.LI Xun, Yang Shaowu, Tang Shuai et al. Research on NuBot Medium-sized Soccer Robot System [J]. Robot Technology and Application, 2010(4): 14 and 16.

12.HUANG Shan, CAI Hegao, TAN Dalong. Multi-robot Cooperative Coordination System for Assembly Operation [J]. Robot, 1999(1): 51-57.

13.LIU Dong, Tong Minming, LU Hongrui. Uav multi-aircraft collaborative exploration of coal mine disaster environment algorithm [J]. Journal of Computer Applications, 2017, 37(8): 2401-2404.

14.Khawaldah M A. Multi-robot exploration of unknown environment using 2D laser scanner [J]. Research Journal of Applied Sciences, Engineering and Technology, 2014, 7(23): 5057-5062.

15.Ghasemlou S, Mohades A, Shangari T A, et al. Homecoming: a multi-robot exploration method for conjunct environments with a systematic return procedure [C]. European Conference on Multi-agent Systems, Cham: Springer, 2014:

111-127.

16.Julian B J, Angermann M, Schwager M, et al. Distributed robotic sensor networks: An information-theoretic approach [J]. International Journal of Robotics Research, 2012, 31(10): 1134-1154.

17.Mendonca R, Monteiro M, Marques F, et al. A cooperative multi-robot team for the surveillance of shipwreck survivors at sea [C]. Oceans, California:IEEE, 2016: 1-6.

18.Lein A, Vaughan R T. Adapting to non-uniform resource distributions in robotic swarm foraging through work-site relocation [C]. Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, Louis, MO:IEEE, 2009: 601-606.

19.Olfati-Saber R. Flocking for multi-agent dynamic systems: algorithms and theory[J]. IEEE Transactions on Automatic Control, 2006, 51(3): 401-420.

20.Mendiburu F, Morais M, Lima A. Behavior coordination in multi-robot systems [C]Proceedings of the 2016 IEEE International Conference on Automatica, Curico: IEEE, 2016:1-7.

21.Wang X, Ni W, Wang X. Leader-Following formation of switching multirobot systems via internal model [J]. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 2012, 42(3): 817-826.

22.Ghosh A, Ghosh A, Konar A, et al. Multi-robot cooperative box-pushing problem using multi-objective Particle Swarm Optimization technique [C]. Proceedings of the 2012 World Congress on Information and Communication Technologies, Trivandrum: IEEE, 2012: 272-277.

23.Isern-Gonzalez J, Hernandez-Sosa D, Fernandez-Perdomo E, et al. Path planning for underwater gliders using iterative optimization [C]. Proceedings of the 2011 IEEE International Conference on Robotics and Automation, Shanghai: IEEE,2011: 1538-1543.

24.Wang Hongbin, Yin Pengheng, Zheng Wei et al. Path Planning of Mobile Robot Based on Improved A~* Algorithm and Dynamic Window Method [J]. Robot, 2020, 42(3): 346-353.

25.Zhao Xiao, Wang Zheng, Huang Cheng-Kan et al. Path planning of Mobile Robot based on Improved A* Algorithm [J]. Robot, 2018, 40(6): 903-910.

26.Duan Yong, Wang Yu, Yu Xiangyou. Multi-robot environment exploration based on immune genetic algorithm [J]. Journal of Shenyang University of Technology, 2018, 40(3): 299-303. 27.Nazarahari M, Khanmirza E, Doostie S. Multi-objective

multi-robot path planning in continuous environment using an enhanced genetic algorithm [J]. Expert Systems With Applications, 2019, 115: 106-120.

28.Lolla T, Lermusiaux P F J, Ueckermann M P, et al. Time-optimal path planning in dynamic flows using level set equations: theory and schemes [J]. Ocean Dynamics, 2014, 64(10): 1373-1397.

29.Liwei Z, Zhibin L, Jie W, et al. Rapidly-exploring Random Trees multi-robot map exploration under optimization framework [J]. Robotics and Autonomous Systems, 2020, 131.

30.Solovey K, Salzman O, Halperin D: Finding a needle in an exponential haystack: siscrete RRT for exploration of implicit roadmaps in multi-robot motion

Planning[M]. Cham: Springer, 2015: 591-607.

31.Lv Weixin, Zhao Lijun, Wang Ke et al. Unknown environment exploration method based on boundary constrained RRT [J]. Journal of Huazhong University of Science and Technology (Natural Science Edition), 2011, 39(S2): 366-369.

32.Das P K, Jena P K. Multi-robot path planning using improved particle swarm optimization algorithm through novel evolutionary operators [J]. Applied Soft Computing Journal, 2020, 92: 106312.

33.Stentz A. Optimal and Efficient Path planning for partially known environments[J]. The Springer International Series in Engineering and Computer Science, 1997: 203-220.

34.Hasan A, Mosa A. Multi-Robot Path planning based on Max – Min ant colony optimization and D* algorithms in a dynamic environment [C]. InternationalConference on Advanced Science and Engineering, Duhok(IQ): Department of Software University of Babylon Babylon IRAQ, 2018: 110-115.

35.Hong Ye, Wang Hongjian, Bian Xinqian. Research on AUV global path Planning based on Hierarchical Markov Decision Process [J]. Journal of System Simulation, 2008(9): 2361-2363.

36.Xia Chun-Rui, Wang Rui, Li Xiao-Juan et al. Path planning method based on probabilistic Model Checking in Dynamic Environment [J]. Computer Engineering and Applications, 2016, 52(12): 5-11.

37.Dean T, Kaelbling L, Kirman J, et al. Planning with deadlines in stochastic domains[C]. Proceedings of the Eleventh National Conference on Artificial Intelligence,1996.

38.Achour N, Braikia K. An MDP-based approach oriented optimal policy for path planning [C]. Proceedings of the 2010 International Conference on Machine and Web Intelligence, Algiers: IEEE, 2010: 205-210.

39.Wang Jian, Zhang Rubo. Robot Navigation Control Method based on POMDP Model [J]. Journal of Huazhong University of Science and Technology (Natural Science Edition), 2008(S1): 12-15.

40.Yang Qiming, Xu Jiancheng, Tian Haibao et al. Online path planning decision modeling for UAV based on IMM [J]. Journal of Northwestern Polytechnical University, 2018, 36(2): 323-331.

41.Candido S, Hutchinson S. Minimum uncertainty robot path planning using a POMDP approach [C]. Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, Taipei: IEEE, 2010: 1408-1413.

42.Chanel C P C, Teichteil-Konigsbuch F, Lesire C. POMDP-based online target detection and recognition for autonomous UAVs [C]. Proceedings of the 20th European Conference on Artificial Intelligence, Montpellier: IOS Press, 2012: 955-960.

43.Zhang Cymbals, Zhu Jun, Su Hang. Towards the third generation of artificial intelligence. Science in China:Information Science, 2020, 50(9): 1281-1302.]

44.Zhang Bin, Lu Lujia, Wang Fazhong. Information Technology and Informatization, 2020(8): 209-211. Zhang B, Wang B, Wang B, et al.

45.Zong P, Zhu Y, Wang H, et al. WRF-Chem simulation of winter visibility in jiangsu, China, and the application of a neural network algorithm [J].

Atmosphere, 2020, 11(5): 520.

46.Kang H, Kim H, Kwon Y M. RECEN: Resilient manet based centralized multi robot system using mobile agent system [C]. Proceedings of the 2019 IEEE Symposium Series on Computational Intelligence (SSCI), Xiamen: IEEE, 2019: 1952-1958.

47.Yu Chong, Qiu Qiwen. Path planning algorithm for hierarchical robot based on raster map. Journal of University of Chinese Academy of Sciences, 2013, 30(4): 528-538.

48.Wu Jin, ZHANG Guoliang, Jing Bin et al. Control Engineering of China, 2014, 21(S1): 18-22.

49.Pugh J, Martinoli A. Inspiring and modeling multi-robot search with particle swarm optimization [C]. Proceedings of the 2007 IEEE Swarm Intelligence Symposium, Honolulu: IEEE, 2007: 332-339.

50.Liu F, Narayanan A, Bai Q. Effective methods for generating collision free paths for multiple robots based on collision type (demonstration) [C]. Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems, Valencia: International Foundation for Autonomous Agents and Multiagent Systems, 2012: 1459-1460.

51.Richard V M. Mathematical theory of probability and statistics [M]. Academic Press, 2014.