Autonomous Mobile Robot Guidance Map Designer

Submitted April18, 2022, in partial fulfilment of the conditions for the award of the degree of
BSc Hons Computer Science with Artificial Intelligence

Tengjian Zhang

20031756

Supervised by Dr. Tianxiang Cui

School of Computer Science
University of Nottingham, Ningbo China

Abstract

In recent years, autonomous mobile robots (AMRs) have been increasingly used in warehouses, logistics companies, medical institutions, and other locations due to their high efficiency, precision, safety, and speed. AMRs differ from their predecessors, the Automated Guided Vehicles (AGVs), in that they can understand and move independently in their environment. AMRs utilize sophisticated sensors, artificial intelligence, and machine learning technologies to perform path planning and perceive their environment. If AMRs encounter unexpected obstacles while navigating, the robots can use navigation techniques such as obstacle avoidance to adjust their speed, stop, or re-route around the obstacle before resuming their task. A simpler and more convenient approach to controlling and planning the robot's movement is through software that creates a map based on the actual environment and employs built-in algorithms to determine the optimal route for the robot. This project aims to design a functionally similar map designer that allows the user the flexibility to add map objects to the software and assign tasks to the robot during the decision-making process to automatically calculate and present the best path. Additionally, the user can switch between different algorithms to compare and analyze the calculation results for in-depth performance evaluation. This design provides the user with the flexibility to select different algorithms for experiments and easily compare and analyze them to support research.

Contents

List of Figures

List of Abbreviations

AMR Autonomous mobile robot

IMUs Inertial measurement units

AGV Automated guided vehicles

MDP Markov Decision Process

DQN Deep Q-Network

P-DQN Prioritized-DQN

N-DQN N-step DQN

PN-DQN Prioritized N-step DQN

OVF Optimal Value Function

DFS Depth First Search

Chapter 1

Introduction

In this chapter, we present a concise overview of the project. Sections 1.1 and 1.2 elucidate the background and motivation behind the project, providing context for the research. Section 1.3 clearly outlines the aims and objectives of the project. Section 1.4 delineates the comprehensive process undertaken in conducting the study, including the methodologies employed and the steps followed. Section 1.5 briefly summarizes the contribution of our research in comparing different algorithms for computing the optimal path, highlighting its significance. Lastly, in Section 1.6, we provide a succinct overview of the thesis structure, outlining the organization of subsequent chapters.

1.1 Background

An autonomous mobile robot (AMR) is a sophisticated machine capable of autonomously perceiving and navigating its environment without relying on physical or electromechanical guidance devices. In recent years, AMRs have gained widespread adoption in various sectors, including companies, industries, hospitals, and homes, due to their ability to enhance productivity and streamline daily activities. Operating in unpredictable and partially unknown environments, AMRs are designed to circumvent obstacles within their range of movement. They follow predefined paths in indoor or outdoor environments, leveraging floor plans, sonar sensing, and inertial measurement units (IMUs) to provide accurate positioning information, including position, velocity, and orientation. In practice, AMRs enhance pick-and-place applications, such as warehousing, distribution, and production, by minimizing repetitive and unproductive tasks through optimization. By employing intelligent task sequencing and route optimization, cooperation among multiple robots can greatly improve throughput efficiency and reduce errors in job execution [3]. In contrast to traditional manufacturing, where automated guided vehicles (AGVs) are commonly used for material movement, AMRs offer advantages such as communication and collaboration among robots to avoid collisions and congestion, thereby enhancing operational flow and efficiency. Additionally, AMRs demonstrate flexibility in route planning and delivery by creating and adapting multiple routes in the mapping software, which enables the management system to dynamically select the optimal route for pick-up or delivery based on user requirements. In contrast, AGVs often rely on a fixed reference in the initial setup, requiring manual adjustments to the reference and route re-establishment when changes occur in the route or delivery scenario [4].

1.2 Motivation

Path planning is a crucial aspect of designing autonomous mobile robots, as it enables the robot to determine collision-free paths between its start and target positions amidst obstacles in the workspace. Various algorithms can be used to read a map of the environment or workspace and find a free path for the robot to traverse, even in complex environments with many obstacles. The choice of the appropriate algorithm is essential to ensure efficient and quick path planning for the robot [5]. In addition to path planning, acquiring information about the environment and perceiving objects and their relative positions is equally important for designing the movement system of the mobile robot. Determining the type of terrain, such as uneven ground, level ground, upstairs, downstairs, or impenetrable terrain, is necessary for proper navigation. This may require the use of environmental sensors, such as laser rangefinders, to accurately estimate the position of objects and obstacles in the robot's surroundings [6][7].

In some cases, sensing the environment through sensors can complicate the task, especially when performing tasks in known specific environments. In such situations, a map designer can be used to assist the autonomous mobile robot in path planning and decision-making in specific restricted areas. For example, the Mir Fleet is a software for controlling autonomous mobile robots that allows drawing or uploading maps, giving operational instructions, setting obstacles, and generating optimal paths to the endpoint [8]. However, it may be challenging for beginners to use, and there is a need for a more user-friendly map designer.

Furthermore, integrating various algorithms for computing optimal paths in the map designer and comparing their performance under different conditions through experimental results can provide valuable insights. This can help in gaining a comprehensive understanding of how different algorithms perform in different environments and serve as a basis for optimization and improvement of the path planning system. This approach would enable a deeper exploration of the project, facilitating a better understanding of the strengths and weaknesses of different algorithms and supporting further enhancements [8].

1.3 Aims and Objective

The aim is to design a map designer to help autonomous mobile robots with path planning and a range of decisions in specific restricted areas. Representing the complex reality of the environment intuitively allows the user to visually identify the best path. The critical tasks of the project are: firstly, developing software that allows the user to design the map arbitrarily; secondly, enabling the user to add tasks to the decisions; and finally, calculating the best path of movement for the robot.

When designing a map, users can select the objects they want to add and place them anywhere on the map they wish. The objects mainly include:

1. Walls: which prevent the robot from moving and can be represented by lines of different lengths.
2. Floors: areas that need to be set up in order to perform the task of changing floors, through which different maps are connected.
3. charging stations: areas that need to be set up in order to perform the task of charging, which the robot needs to reach.
4. Obstacles: the robot can not move around this area.
5. Preferred zone: the zone that the robot is considered to enter first in the route design.
6. Unpreferred zone: in route design, the robot first avoids this zone but may enter it if there is no other possibility.
7. Forbidden zone: the robot is forbidden to enter the zone.

8. cargo container: the zone that needs to be set up in order to perform the task of handling goods, the robot needs to reach the zone, and multiple cargo containers can be placed on the map.

The main tasks that can be added to the decision include:

1. move: set the start and end points. The robot needs to start from the start and end at the end.

2. Charging: the robot needs to reach a charging station in order to complete this command.

3. Changing floors: When designing a map, new maps can be created at the same time as the current map, and the robot needs to move to a different map.

4. Carrying goods: the robot needs to set the position of the container and the location where it will arrive. The robot needs to move to the location of the container and move to the next location.

Route planning is a critical aspect of the map design process. The user defines the starting point and endpoint for the robot, which then navigates through the map to complete assigned tasks, aiming to reach the endpoint in the shortest possible time. An algorithm calculates the optimal path and displays it on the map. In the map designer we will present the best paths, running times and cumulative rewards calculated with different algorithms in a professional manner. Such a presentation will facilitate a comprehensive comparison of the performance of different algorithms under various conditions and will provide strong support for the analysis of the project.

1.4 Contributions

1. We utilize a Markov Decision Process (MDP)-based mathematical model in conjunction with value iteration and policy iteration algorithms within the framework of reinforcement learning to compute the optimal trajectory for an autonomous mobile robot as it traverses a given map. The pertinent map, task, and parameter information are stored in .xlsx files, and the final output showcases the optimized path for the robot's traversal.

2. In addition to employing the value iteration and policy iteration algorithms from reinforcement learning for path optimization, we have also incorporated the A* algorithm, allowing for algorithmic switching within the map designer. This facilitates a comprehensive evaluation of the merits and demerits of multiple methods, enabling a comparative assessment of algorithmic performance.

3. We present a thorough analysis of algorithmic disparities in reinforcement learning by evaluating the performance of three distinct algorithms across multiple environments with regards to best path identification, runtime efficiency, and cumulative reward. Furthermore, we conduct comparative assessments against the A* algorithm to ascertain the relative strengths and weaknesses of these algorithms in terms of performance and generalizability.

1.5 Dissertation Outline

This thesis is organized into three main sections:

Chapters 1, 2, and 3: These chapters provide the technical background for this project, covering efficient map design, reinforcement learning concepts, and the utilization of value iteration, policy iteration, and A* algorithms for calculating optimal robot paths.

Chapters 4 and 5: In these chapters, we conduct experimental testing of the three algorithms under varying conditions and compare and analyze the results. We also present the algorithmic designs and implementation. By the end of these chapters, the reader will be proficient in selecting the most suitable algorithm for obtaining the shortest path in different environments and utilizing the map designer.

Chapter 6: This chapter highlights the contributions of the project and outlines future directions. It encompasses a discussion on the achievements, impact, and contributions of the project, as well as identifying potential areas for further research. By the conclusion of these chapters, the reader will have a comprehensive understanding of the project's accomplishments and the potential future prospects.

Chapter 2

Literature Review

In this chapter, two approaches to map design are concisely outlined in Section 2.1. Global planning and local planning are briefly discussed in Section 2.2. Section 2.3 presents an analysis of the distinctions between employing a map designer and sensors. Section 2.4 introduces several prevalent algorithms for reinforcement learning, while Section 2.5 introduces the concept of dynamic programming. The primary emphasis is on map design and optimal path calculation, while detailed descriptions of sensors and related algorithms employed by autonomous mobile robots are not provided.

2.1 Map Design

Two-dimensional maps and three-dimensional maps are distinct categories of maps. Two-dimensional maps possess certain advantages, including reduced information content, simpler construction and maintenance processes, and enhanced scalability. This section introduces two distinct approaches to map design.

2.1.1 Grid-based mapping

Grid-based mapping, originally proposed by Elfes and Moravec, is a widely used method for efficiently determining the spatial distribution of small features over large areas [9]. In this approach, a grid map is designed by dividing the environment into a regular grid of cells, with the initial point of the robot often set as the centroid of the grid. The size of each grid cell is determined by the chosen x- and y-axis resolutions, which can be adjusted to balance map accuracy and computational efficiency.

The resolution of the grid map plays a crucial role in the accuracy of the resulting map. A smaller resolution, corresponding to smaller grid cells, can provide higher map accuracy by capturing more detailed information about the environment. However, it also increases the computational burden and may require larger memory storage. Thus, the choice of an appropriate resolution is a trade-off between map accuracy and computational efficiency, depending on the specific requirements of the application. Each cell in the overlay grid can be assigned a simple identifier to represent specific terrain characteristics. As shown in the figure 2.1, in a typical grid map, the start point of the robot may be denoted by yellow grids, the endpoint by red grids, open space by white grids, and obstacles by black grids. This allows for easy visualization and analysis of the environment, facilitating decision-making for robot navigation or other applications. The raster map representation used in grid-based mapping is a common approach to represent the environment as a collection of cells, where each cell can be classified as free space or obstacle based on the assigned identifier. This representation enables efficient storage and processing of map data, making it suitable for real-time applications.

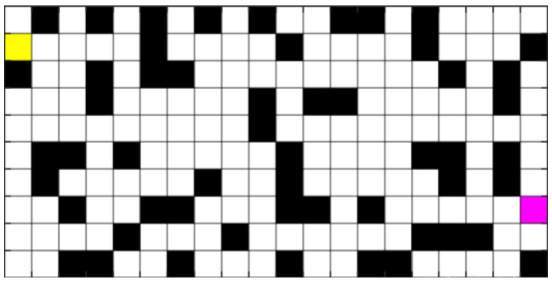


Figure 2.1: Grid-based mapping

When employing grid-based mapping techniques, careful consideration must be given to the order in which cells are completed [12], as it can significantly impact the effectiveness and efficiency of the mapping process. In this regard, three different methods are commonly used, as illustrated in Figure 2.2. The first method, referred to as "Adjacent Cell Completion," involves completing cells row by row sequentially. This approach ensures that cells in the same row are completed before moving on to the next row. The second method, known as "Random Completion," assigns a random number to each cell and completes them sequentially based on their assigned numbers [10]. This method introduces an element of randomness in the mapping process, which may impact the final outcome. The third method, termed "Coarse-Resolution Completion First," involves completing every third cell (depicted in dark green in C), allowing the results to be extrapolated to adjacent cells (light green) to generate a coarse-resolution raster. This method provides a quick initial approximation of the map, which can then be refined in subsequent iterations [11].

Compared to other mapping methods, raster maps offer several advantages, including ease of construction, representation, and storage, as well as unique location referencing, which can be advantageous in planning short paths. However, raster maps have limitations in terms of path planning efficiency and reliance on precise robot positioning. Nonetheless, these methods can be valuable in various mapping applications, such as robotic navigation, environmental monitoring, and terrain analysis.

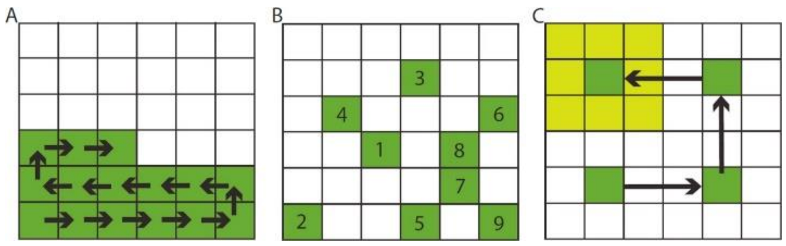


Figure 2.2: The order in which cells are completed.

2.1.2 Topological mapping

The concept of topological maps was first introduced by Kuipers and Byun, and later applied to mobile robot localization by Cassandra and Ulrich [13]. A topological map is a method of representing the external environment using a topological diagram composed of nodes and undirected edges, which capture the topology of the environment. Typically, a topological map consists of two fundamental elements, denoted as G = (V, E), where V represents the nodes in the graph, and E represents the interconnected paths between the nodes [14]. In this representation, each node corresponds to a specific location in the external environment, such as a corner, while edges are used to represent the connections or transitions between nodes. The topological map serves as a graphical abstraction of the environment, capturing the spatial relationships and connectivity between locations. The process of constructing a topological map is illustrated in the figure 2.3, depicting the robot's traversal from a starting point to a target endpoint, denoted as qG. The topological map provides a high-level representation of the environment's topology, facilitating the robot's ability to plan and navigate through the environment based on the connections between nodes. This abstraction of the environment into a topological map allows for efficient and robust robot navigation, as it eliminates the need for precise metric maps and enables the robot to rely on qualitative spatial relationships.

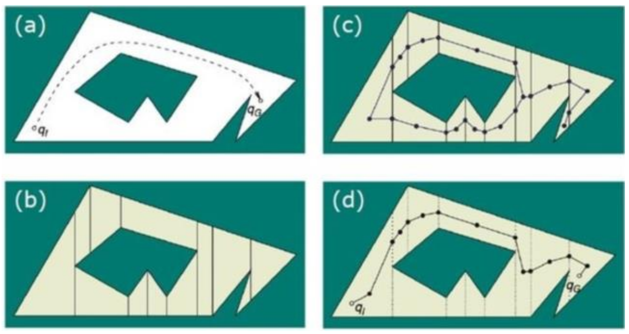


Figure 2.3: Topological mapping

2.2 Path planning

Path planning is the ability of a mobile robot to plan an optimal or near-optimal path from a starting state to a target state. It consists roughly of information acquisition, sensing, communication, decision-making, control, and execution. The implementation of path planning for mobile robots can be divided into global and local path planning. This section introduces two distinct approaches to path planning.

2.2.1 Global planning

Global planning refers to the process of generating a robot's route in a known environment, and the accuracy of the planned path is contingent upon the accuracy of the environment acquisition. A global weight map is generated based on the target location at a given point, by incorporating weight map information, which is then used to plan a global path from the starting point to the target location [15]. The process involves several steps, as depicted in Figure 2.4, including (a) obtaining a metric map of the environment, (b) generating an obstacle inflation map, (c) performing grid localization, (d) creating an occupancy grid map, (e) planning the global coverage robot path using the boustrophedon motion and A* algorithms, and (f) achieving obstacle area coverage through continuous reconstruction capability. However, this global planning method may encounter challenges when the environment undergoes changes, such as the emergence of unknown obstacles. In such cases, the planning process becomes anticipatory and does not require high real-time computing capabilities of the robot system. Despite the advantages of providing global and improved planning results, this method may exhibit limited robustness in the face of dynamic environment models. Further research in this area can contribute to addressing the limitations of global planning methods and improving their robustness in dynamic environments. This could involve developing more accurate environment acquisition techniques, refining the weight map generation process, and exploring alternative planning algorithms that are capable of adapting to changing environments in real-time. Additionally, investigating approaches to seamlessly integrate global planning with local planning and perception modules can enhance the overall performance of robotic systems in navigating in dynamic environments.

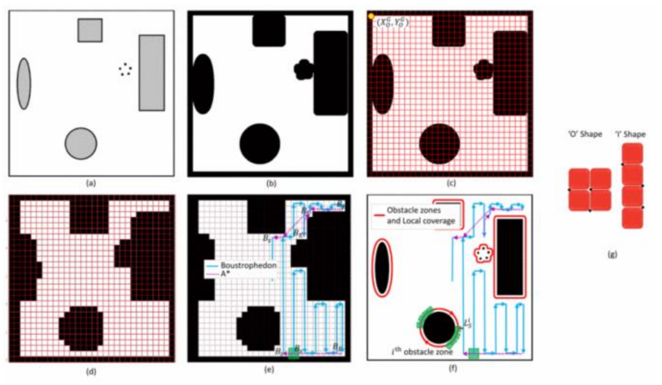


Figure 2.4: Global path planning

2.2.2 Local planning

Local planning refers to the process of enhancing a robot's obstacle avoidance capabilities by taking into consideration the current local environmental information when complete or partial knowledge of the environment is unavailable [15]. This involves utilizing sensors to detect the robot's working environment, which provides data on the location and geometric properties of obstacles. Local planning requires continuous collection and dynamic updates of environmental data, enabling real-time modeling of the environment and simultaneous search for an optimal path. This process demands a robot system with high-speed information processing and computational capabilities, robustness to environmental errors, and the ability to provide real-time feedback and correction of computational results. However, due to the absence of global environmental information, the resulting local path planning may not always yield an optimal path or identify the correct and complete path [16]. This limitation underscores the need for further research and advancements in local planning techniques. Efforts can be directed towards improving the accuracy and reliability of environmental data collection through sensor fusion, developing robust algorithms that can effectively handle uncertainties in the environment, and exploring methods for integrating local planning with other planning modules in a seamless and coherent manner.

2.3 Use of sensors

The conventional approach to autonomous mobile robot pathfinding often relies on sensor-based perception, scene analysis, and path planning. Laser SLAM and vision SLAM are commonly used methods for robust localization in indoor environments. Laser SLAM uses a laser rangefinder to accurately estimate the robot's position, but it may face challenges in large spaces or long corridors where accurate pose estimation can be difficult without sufficient observations, and depth information may be featureless and invariant over time, leading to potential navigation errors [17]. Vision SLAM, on the other hand, uses a monocular camera for environment mapping, but a single camera may not be sufficient to address the scaling problem, and changes in illumination can introduce uncertainties in robot localization [18]. Additionally, the complexity of hardware equipment and the uncertainty associated with environmental mapping pose challenges in using sensor-based approaches.

In contrast, employing a map designer allows for bypassing these challenges. Map designers provide a means to create maps of the environment using known data, eliminating the need for real-time sensor-based perception and scene analysis. This can result in faster and more reliable global path planning, as the map designer allows for incorporating known environmental information directly into the planning process, without being affected by limitations such as sensor accuracy, environmental changes, or illumination variations [18]. Map designers offer the advantage of efficiency and accuracy in global path planning for known environments, as the environmental information is preprocessed and readily available for planning purposes. Furthermore, map designers offer flexibility in incorporating additional contextual information, such as semantic maps or high-level features, which can enhance the planning accuracy and efficiency of autonomous mobile robots in various environments. By incorporating such contextual information, map designers can enable robots to make more informed decisions during path planning, taking into account semantic information, object recognition, or other high-level features that may not be directly available from sensor-based perception.

In conclusion, the utilization of a map designer in autonomous mobile robot pathfinding can offer advantages over traditional sensor-based approaches. It can lead to improved efficiency and accuracy in global path planning for known environments, as the map designer allows for incorporating known environmental information directly into the planning process, without being affected by limitations such as sensor accuracy, environmental changes, or illumination variations.

2.4 Reinforcement Learning

Reinforcement learning, a widely adopted paradigm in artificial intelligence, encompasses the fundamental components of Agent, Environment, State, Action, and Reward [20]. The Agent, also referred to as the intelligent entity, represents the entity that learns and interacts with the environment. After executing an action, the environment undergoes a transition to a new state, typically represented by environmental data, and provides a reward as a positive or negative feedback signal in response to the new state. The intelligent entity then utilizes a specific strategy to determine its next action based on the new state and the received reward, completing a cycle of interaction between the Agent and the environment. This iterative process of state-action-reward interactions forms the foundation of reinforcement learning, enabling the intelligent entity to learn and make optimal decisions to maximize its cumulative reward. Notably, the relationship between the intelligent entity and the environment in reinforcement learning exhibits similarities with the interaction between a human and its environment [40]. The intelligent entity gains knowledge of its current state and makes decisions on actions to be taken based on the observed state and associated rewards. Consequently, reinforcement learning serves as a versatile framework for addressing diverse artificial intelligence problems, providing a means for an intelligent entity to learn an optimal sequence of decisions while navigating its environment. By leveraging reinforcement learning, an intelligent entity acquires the capability to adaptively learn and improve its decision-making abilities through interactions with its environment, akin to the learning process observed in humans.

2.4.1 Gym

The Gym framework is a widely used tool for reinforcement learning, providing an environment in which datasets can be created and utilized for testing and learning reinforcement learning algorithms. Following the classic "agent-environment loop" [21], Gym facilitates the interaction between an agent and the environment, where the agent performs actions and observes how the environment responds, forming a timestep that captures the action-observation exchange. This loop is illustrated in Figure 2.5, depicting the flow of the reinforcement learning process. Over time, the environment may reach a terminal state, at which point it needs to be reset to a new initial state. When the agent encounters a terminal state, the environment signals the completion of a task to the agent.

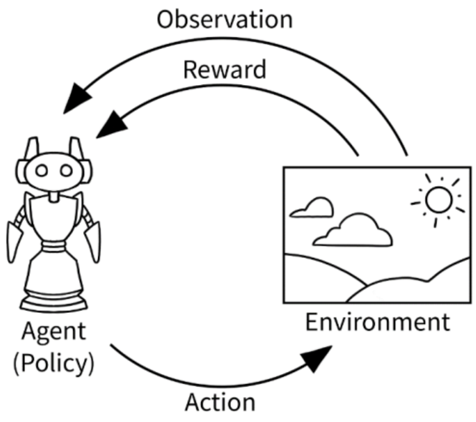
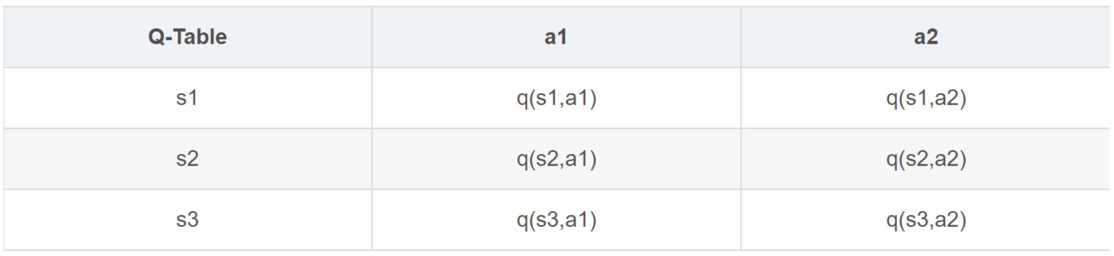


Figure 2.5: The flow of the loop

2.4.2 Q-Learning

Q-learning is a classical value-based reinforcement learning algorithm that is model-free and non-policy. It is commonly used to enable an agent to learn optimal actions based on its current state in an unknown environment. The algorithm constructs a Q-table to store the Q-values, as shown in Figure 2.6. These Q-values represent the expected cumulative reward for taking a certain action in a specific state. By updating the Q-values iteratively, the agent can optimize its decision-making process to maximize its long-term cumulative reward [41].



Q-Table	a1	a2
s1	q(s1,a1)	q(s1,a2)
s2	q(s2,a1)	q(s2,a2)
s3	q(s3,a1)	q(s3,a2)

Figure 2.6: Q-Table

The formula for updating the Q table is:

$$Q(s,a)^{new} = Q(s,a)^{old} + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)^{old}] \tag{2.1}$$

It can also be expressed as:

$$Q(s,a)^{new} = (1-\alpha)Q(s,a)^{old} + \alpha[r + \gamma max_{a'} Q(s',a')] \tag{2.2}$$

where Q(s, a) denotes the Q-value for taking Action a in State s, α is the learning rate, r is the immediate Reward obtained by the agent for taking Action a in State s, γ is the discount factor, s' is the next State the agent transitions to after taking Action a, and a' is the action selected in the next State s'.

Q-learning is considered model-free because the agent does not require prior knowledge about the dynamics of the environment and learns directly from its interactions with the environment. The agent uses the Q-values as a guide for selecting actions, without relying on a predefined policy [23]. The main objective of Q-learning is to find the optimal action for each state, given the current estimates of the Q-values. The algorithm iteratively updates the Q-values based on the observed rewards and transitions between states. The update rule consists of two terms. The first term α * (r + γ * maxQ(s', a')) represents the weighted sum of the immediate reward and the maximum Q-value of the next state, discounted by γ. This term reflects the agent's trade-off between the immediate reward and the potential future rewards. The second term Q(s, a) represents the current estimate of the Q-value, which is updated towards the updated estimate of the Q-value for the next state-action pair. By iteratively updating the Q-values, the agent gradually refines its estimates and improves its action selection process to make more informed decisions [22].

The implementation of Q-learning typically involves the following steps:

1. Q-table initialization: Before learning starts, the Q-table is initialized with small random values or zeros to represent the initial estimates of the Q-values for all state-action pairs.

2. Environment interaction: The agent interacts with the environment by selecting actions based on its current Q-values, and observing the rewards and next states as feedback from the environment.

3. Q-value update: After observing the rewards and next states, the Q-values are updated using the Q-learning update rule. The update is guided by a learning rate (denoted as α), which determines the weight of the new information compared to the existing Q-value, and a discount factor (denoted as γ), which determines the weight of future rewards in the Q-value update. This step is crucial for the agent to learn from its interactions with the environment and update its Q-values accordingly. Action selection: The agent selects the next action to take based on its updated Q-values, using a suitable action selection policy such as epsilon-greedy or softmax.

The Q-learning algorithm continues to iterate between environment interaction, Q-value update, and action selection until convergence, which refers to the point where the Q-values have converged to their optimal values. At this stage, the agent is able to make optimal decisions based on its current state, leveraging the learned Q-values to determine the best actions to take in order to maximize its rewards and achieve its objectives. In summary, the implementation of Q-learning entails initializing the Q-table, interacting with the environment, updating the Q-values, and selecting actions based on the updated Q-values. These steps are iteratively performed until convergence, allowing the agent to learn and make optimal decisions in the reinforcement learning process.

2.4.3 Deep Q-Network

The Deep Q-Network (DQN) algorithm is a deep neural network-based reinforcement learning algorithm for solving Markov decision process (MDP) problems in discrete action spaces. The DQN algorithm overcomes the capacity limitations and sample correlation problems of traditional Q-table-based methods by combining deep learning with reinforcement learning to approximate value functions using deep neural networks. The core idea of the DQN algorithm is to use a deep neural network to approximate the value function, which is represented by the Q-value function, to guide the choice of actions to be made by an agent in its environment. The Q-value function (also known as the action value function) represents the expectation of the cumulative reward for taking an action in a given state [24].

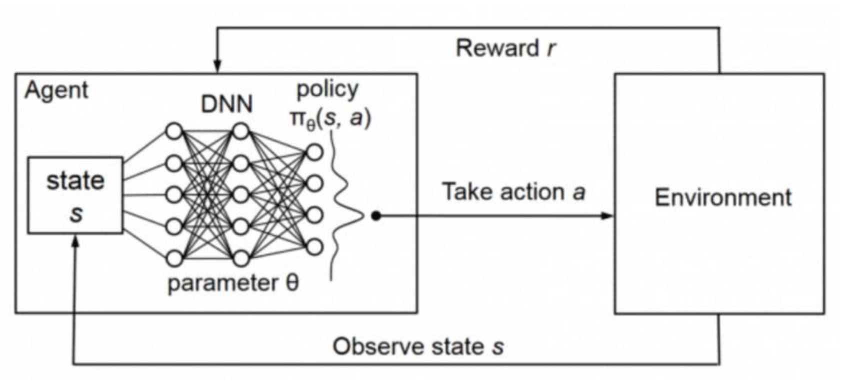


Figure 2.7: Deep Q-Network

The Q-value function is defined as:

$$Q(s, a; \theta_i) \tag{2,3}$$

The DQN algorithm updates the parameters of the neural network by minimising the mean square error of the Q-value function, thus gradually optimising the estimation of the value function. The loss function is used here:

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)}_{\text{target}} - Q(s, a; \theta_i)\right)^2 \tag{2.4}$$

The Target is calculated as the sum of the Reward obtained in the current step and the maximum Q value that can be obtained in the next state. This sum is then subtracted from the Q value in the current step, resulting in the difference between the actual Reward and the Reward value that the existing model predicts when action 'a' is taken under state 's'. Experience Replay and Target Network are often used to improve the stability and convergence of the algorithm [25]. The Experience Replay technique works by storing experience samples obtained from the interaction of the agent with the environment in an experience cache and randomly sampling them for updating the parameters of the neural network. This way, the DQN can learn from each experience sample independently, reducing the correlation between the samples. The fixed objective network technique improves the convergence of the algorithm by fixing the parameters of the objective function and reducing the impact of changes in the objective value on the update parameters.

$$Q(s_t,a) \leftarrow Q(s_t,a) + \alpha \left[r_{t+1} + \gamma \max_p \boxed{Q(s_{t+1},p)} - Q(s_t,a) \right] \tag{2.5}$$

The pseudo-code for training is represented as:

Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    **for** $t = 1, T$ **do**
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation $\boxed{3}$
    **end for**
**end for**
```

P-DQN (Prioritized-DQN) is the use of Experience Replay on top of the traditional DQN algorithm, thus improving learning efficiency and performance. In contrast, N-DQN (N-step DQN) is an improvement to DQN, aiming to enhance the performance and learning speed of the algorithm by introducing the concept of N-step reward. PN-DQN (Prioritized N-step DQN) is a combination of N-step DQN and Experience Replay, aiming to further improve the learning speed and performance [26]. As shown in Figure 2.8, the losses during training are displayed. Comparing P-DQN with DQN and PN-DQN with N-DQN, respectively, the model with the efficient learning strategy shows faster decrease in losses under the same conditions. The loss curves of PN-DQN and P-DQN are more stable and smaller, while the loss plots of N-DQN and DQN are very similar. However, due to the more efficient learning and better fitting of the models, N-DQN converges faster and has smaller losses. This demonstrates that the PN-DQN model is more stable, faster, and more accurate than the traditional model.

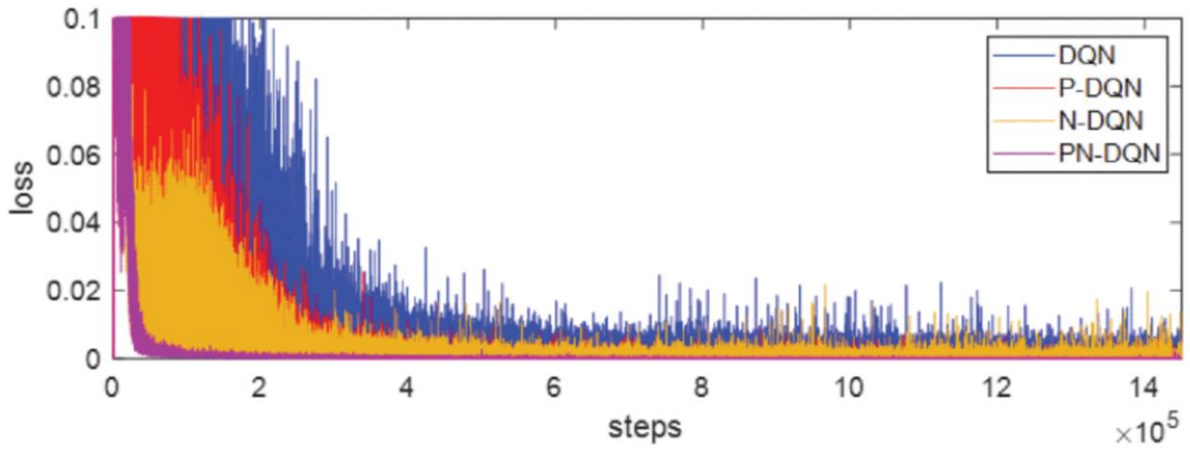


Figure 2.8: The loss of training process.

The DQN algorithm has the advantage of being able to handle high-dimensional, complex state spaces and to learn non-linear relations, which can be improved in combination with other deep learning techniques and is highly scalable. However, the convergence of the DQN algorithm is full, requiring a large number of training samples to achieve good performance, and the computational complexity of the process also limits its use in some environments.

2.4.4 Monte Carlo Method and Temporal Difference Learning

Monte Carlo methods encompass a class of random sampling techniques used to approximate results by performing calculations on random samples and refining accuracy as the sample size increases [27]. These methods simulate a large set of samples to approximate the actual problem being studied. Notably, Monte Carlo methods are characterized by their simplicity and flexibility, making them applicable to a wide range of problems, particularly in cases where the model is unknown or complex. Nevertheless, Monte Carlo methods do have limitations, including slow and inefficient learning due to the need to wait until the end of each trial to obtain results, as well as sensitivity to sample size and high computational costs for achieving high accuracy. In contrast, time-difference learning is a prediction method based on future values of a given signal [28]. Combining sampling from Monte Carlo methods with bootstrapping from dynamic programming methods, time-difference learning is well-suited for model-free algorithms and single-step updates, resulting in a faster learning rate. Unlike Monte Carlo methods, time-difference learning updates the value function at each time step, avoiding the need to wait until the end of a trial to obtain results. This allows for real-time updating and learning, making it advantageous in scenarios where the model is known and a faster learning rate is required. Time-difference learning obtains subsequent states through trials, similar to dynamic programming methods, but without the need for a model [29]. This makes it particularly suitable for real-world environments with real-time data, enabling updates and learning to occur in a shorter timeframe.

Both Monte Carlo methods and time-difference learning are critical technical tools in reinforcement learning, with specific applications in different scenarios. Monte Carlo methods are suitable for estimating environments where the model is unknown or complex, especially when a large number of samples are required. On the other hand, time-difference learning is well-suited for scenarios where real-time updating and learning are needed, particularly when the model is known and a faster learning rate is desired. A comprehensive understanding of the characteristics and application scenarios of these two methods can aid in selecting the appropriate reinforcement learning algorithm for practical problems.

2.5 Dynamic Programming

Dynamic Programming is a powerful mathematical optimization method that is widely used for solving multi-stage decision processes optimally. It involves transforming a complex multi-stage problem into a series of interconnected single-stage problems, which are then solved iteratively, one by one. This bottom-up approach allows for the construction of a global optimal solution by leveraging the optimal solutions of the sub-problems [30]. In the context of reinforcement learning, Dynamic Programming comprises two fundamental aspects: the state transfer equation and the optimal policy equation. The state transfer equation describes how an intelligent agent transitions from one state to another in the environment. It is a crucial component of the Markov Decision Process (MDP), which characterizes the dynamics of the agent's interaction with the environment. The state transfer equation is typically expressed as:

$$S_{t+1} = f(S_t, A_t) \qquad (2.6)$$

where S_t represents the state at the current moment, A_t denotes the action chosen at the current moment, and S_{t+1} represents the state at the next moment. The function f can take the form of a probability distribution function, capturing the stochasticity of the state transitions, or a deterministic function, capturing the deterministic relationship of the state transitions.

The optimal policy equation (OPE) is a mathematical formula that represents how an intelligent agent should choose actions to maximize the cumulative payoff in a reinforcement learning setting. In an MDP, the OPE can be expressed as:

$$\pi^{\wedge}(s) = \text{argmax}_{a}\, Q^{\wedge}(s, a) \qquad (2.7)$$

where $\pi^{\wedge}(s)$ denotes the optimal policy in state s, and $Q^{\wedge}(s, a)$ denotes the optimal value function after choosing action a in state s. The optimal policy equation guides the agent in selecting the action that maximizes the Q-value, which represents the expected cumulative reward when following a particular action from a particular state. The optimal policy equation enables the agent to make informed decisions to achieve its objectives effectively.

Chapter 3 Methodology

In this chapter, Section 3.1 introduces the concepts and methods of MDP, Section 3.2 introduces the Bellman equation, which is the main method used in this project to calculate the optimal path using reinforcement learning, and Section 3.3 introduces three algorithms for calculating the optimal path, including A*, value iteration and policy iteration.

3.1 Markov Decision Process

A state is considered to have Markovianity when the information about a state contains all the relevant history and the current state can determine the future as long as the current state is known and all the historical information is no longer needed. Expressed in the formula:

$$P(S_{t+1}|S_t) = p(S_{t+1}|S_1, S_2, \cdots, S_t) \tag{3.1}$$

The Markov Decision Process (MDP) is a mathematical model for describing how an intelligence agent chooses actions in a series of decisions to maximise cumulative rewards in a stochastic environment.The MDP is widely used in reinforcement learning and is a common formal framework for modeling and solving decision problems under uncertainty. The MDP consists of a five-tuple (S, A, P, R, γ), where: S denotes the set of states, representing the set of all possible states that the environment could be in. a denotes the set of actions, representing the set of all actions that the intelligence could choose. p denotes the state transfer probability, representing the probability that after an action is taken in the current state, the environment shifts to each possible state at the next moment, denoted as:

$$P(s_{t+1} = s'|s_t = s, a_t = a) = P[S_{t+1} = s'|S_t = s, A_t = a] \tag{3.2}$$

R denotes the immediate reward function, which represents the immediate reward received by the intelligence after an action is taken in the current state. Expressed in the equation as:

$$R(s_t = s, a_t = a) = E[R_t|s_t = s, a_t = a]$$

γ denotes the discount factor, which is used to weigh the importance of current rewards against future rewards. The discount factor γ lies between [0, 1] and is used to balance the trade-off between immediate and delayed rewards, with the closer γ is to 1, the greater the focus on future rewards, and the closer γ is to 0, the greater the focus on immediate rewards. the goal of the MDP is to find a strategy function π(s): S → A that allows the intelligence to choose the appropriate action in different states, thereby maximising cumulative rewards, denoted as：

$$\pi(a|s) = P(A_t = a|S_t = s)$$

The policy function π(s) represents the action chosen in the current state, which can be deterministic or stochastic.

3.2 Bellman Equation

The Bellman equation is a fundamental mathematical concept in dynamic programming, representing the equation for the relationship between adjacent states in a dynamic planning problem, which is necessary to achieve optimization in reinforcement learning. The Bellman equation is transformed into a subproblem for the next stage of optimal decision-making so that the optimal decision for the initial state can be solved iteratively from the optimal decision problem for the final state step by step [31].

The core problem of reinforcement learning is to optimize the policy function π(s) in order to maximize the value function. The value function is a measure of the expected cumulative reward that an agent can obtain from a given state if it follows a certain strategy [32]. The state value function, denoted as V(s), is one of the key criteria for evaluating the optimality of the strategy function, denoted as:

$$v(s) = E[G_t | S_t = s] \tag{3.3}$$

where Gt is expressed as the sum of the decaying returns of all rewards on a Markovian reward chain from moment t onwards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T \tag{3.4}$$

The value V in the Bellman equation can be expressed as:

$$\begin{aligned} v(s) &= E[G_t | S_t = s] \\ &= E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | S_t = s] \\ &= E[R_{t+1} + \gamma (R_{t+2} + R_{t+3} + \cdots) | S_t = s] \\ &= E[R_{t+1} + \gamma v(S_{t+1}) | S_t = s] \\ &= E[R_{t+1} | S_t = s] + \gamma E[v(S_{t+1}) | S_t = s] \end{aligned} \tag{3.5}$$

According to the Bellman equation, the value function V(s) has two components. One component is the expectation of the current reward received, denoted as R(s). The other component is the expectation of the state value at the next moment, which is the sum of the product of the expectation that possible states can receive a reward at the next moment and the probability of the corresponding state transfer, with a discount factor added at the end [33].

In addition to the state value function, there is a state-action value function denoted as Q(s, a), which goes beyond the state value function by fixing both the state and the action corresponding to that state. The state-action value function represents the expected cumulative payoff that an agent can obtain by taking a particular action in a given state and following a certain strategy thereafter. The benefits of acting in a given state come from two sources: the immediate reward R from the environment, and the delayed reward, i.e., the expected cumulative reward that an agent can obtain from subsequent states and actions. The formula is:

$$\mathrm{Q}_\pi(\mathrm{s}, \mathrm{a}) = \sum_{\mathrm{s}' \in \mathrm{S}} \mathrm{P}^{\mathrm{a}}_{\mathrm{s} \to \mathrm{s}'} \cdot \left(\mathrm{R}^{\mathrm{a}}_{\mathrm{s} \to \mathrm{s}'} + \gamma \mathrm{V}_\pi(\mathrm{s}')\right) \tag{3.6}$$

In a stochastic environment, the next state appears as a probability of a universal system, even if the action has been determined. The action value function Q(s, a) therefore needs to be expressed in terms of expectations, taking into account the probabilistic nature of the environment. The maximisation of the state value function V(s) involves finding the maximum expected value of the current state, taking into account all possible actions from that state. On the other hand, maximising the state action value function Q(s, a) involves finding the maximum value of the expected cumulative reward that can be obtained by taking a particular action in the current state and following a certain strategy thereafter [33].

Combining the two formulations above we obtain the Bellman expectation equation, which can be used to update the value function of the state and thus the decision making process of the intelligence in reinforcement learning, expressed as

$$\begin{aligned} v_\pi(s) &= \sum_{a\in A}\pi(a|s)\left(R(s,a)+\gamma\sum_{s'\in S}P_{(s'|s,a)}\cdot v_\pi(s')\right) \\ q_\pi(s,a) &= R(s,a)+\gamma\sum_{s'\in S}P_{(s'|s,a)}\cdot\sum_{a'\in A}\pi(a'|s')\cdot q_\pi(s',a') \end{aligned} \tag{3.7}$$

"In reinforcement learning, the Optimal Value Function (OVF) is the value function of the highest cumulative reward that can be achieved by an intelligent agent. The Optimal Value Function is usually represented symbolically as V*(s) (for a state-value function) or Q*(s, a) (for a state-action-value function) [34]. The optimal value function helps the agent to choose the optimal action to maximize the cumulative reward. By computing the optimal value function, the agent can learn the long-term value of taking different actions in different states and thus guide the agent's decision-making process. It takes the form of:"

For state value functions:

$$v_*(s) = max_a(R(s,a)+\gamma\sum_{s'\in S}P^a_{s's}\cdot v_*(s')) \tag{3.8}$$

For state-action value functions:

$$q_*(s,a) = R(s,a)+\gamma\sum_{s'\in S}P^a_{s's}\cdot max_{a'}q_*(s',a') \tag{3.9}$$

3.3 Optimal Path Algorithm

In this section, we present three algorithms used in optimal path planning for computer robot movement, based on the Markov Decision Process (MDP) framework of reinforcement learning. The classical reinforcement learning algorithms Value Iteration and Policy Iteration are primarily used, with the A* algorithm introduced as a control to compare the differences and generalization performance of heuristic search and reinforcement learning in path planning.

3.3.1 A*

The A* algorithm is a heuristic search algorithm commonly used in search problems to find a path around an obstacle from the start point to the goal point by continuously expanding the neighbouring nodes until the goal point is reached. The A* algorithm efficiently finds the best path during the search by using a combination of actual cost (the path cost from the start point to the current point) and heuristic estimates (the estimated cost from the current point to the target point) to select the next search direction [35].

f (n) is the combined priority of node n. When we choose the next node to traverse, we always pick the node with the highest combined priority (smallest value).

$$f(n) = g(n) + h(n) \tag{3.10}$$

g(n) is the cost of node n from the starting point, and h(n) is the expected cost of node n from the end point, which is the heuristic function of the A* algorithm. the A* algorithm selects the node with the smallest value of f(n) (highest priority) from the priority queue as the next node to be traversed during each operation. In addition, the A* algorithm uses two sets to represent the nodes to be traversed, and the nodes that have already been traversed, which are usually called open_set and close_set.

When calculating optimal paths, Geometric A* is often used, which is an extension of the geometric space-based A* algorithm to handle path planning problems on 2D grid maps where each grid cell can have different geometric properties such as obstacles. Unlike traditional A* algorithms, Geometric A* allows for more accurate and efficient path planning on maps with geometric information.

Algorithm: Geometric A-start

```
algorithm Geomtric A-start(start, n, goal)
if $reachAroundGoal$(start) $\neq$ goal then return $makePath$(start)
open $\leftarrow$ $closestPoint$(start)
closed $\leftarrow$ 0
final $\leftarrow$ 0
while open $\neq$ 0 do
  sort(open)
  n $\leftarrow$ open.pop( )
  if $reachAroundGoal$(n) $\neq$ goal then return $makePath$(n)
    neighbors $\leftarrow$ $expendFlexibleUnits(n)$
   end if
  for all the neighbors do
      if neighbor $\notin$ $Obstracle$
        neighbor.f $\leftarrow (n.g + g) + (n.p + p) + h$
        if neighbor $\cap$ closed=$\varnothing$ then open $\leftarrow$ neighbor
          else closed $\leftarrow$ neighbor
        end if
        closed $\leftarrow$ n
        if $crossPath(n)$ and $sawtoothPath(n)$ then
          final $\leftarrow$ closed
        end if
      end if
  end for
end
return 0
```

We have chosen to represent the map data in a grid format for this project. For a grid-based representation, the Manhattan distance can be used as a heuristic if movement is only allowed in four directions: up, down, left, and right. Alternatively, if movement in any direction is allowed, the Euclidean distance can be used, as shown in the figure. This choice of heuristics depends on the allowed directions of movement in the graph.

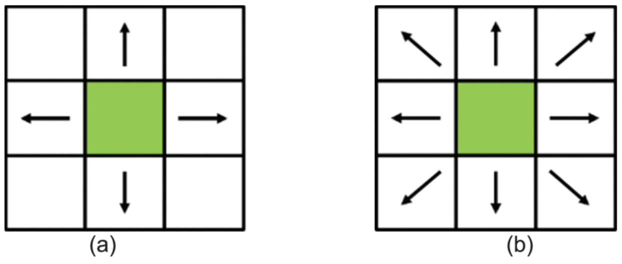


Figure 3.1: Search direction of two distance formulas: (a) Search direction of manhattan distance; (b) Search direction of euler distance

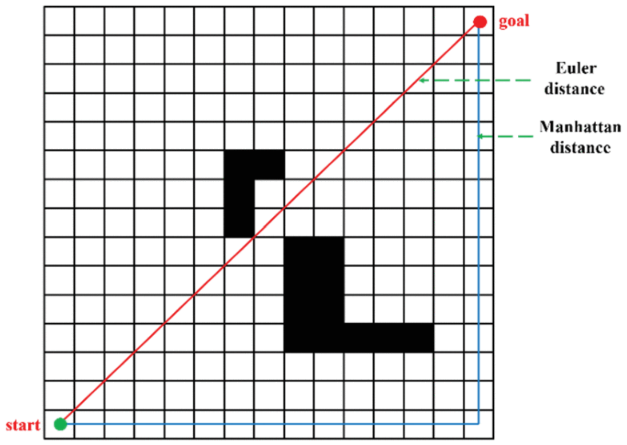


Figure 3.2: Paths generated by the Euler distance and manhattan distance

In this project, I have made some improvements to the A* algorithm. Since the traditional A* algorithm does not support the identification of preferred and non-preferred regions, I have added a findBestPolicy() function to the traditional algorithm to generate a policy for each successful path. This policy is based on reinforcement learning principles and determines the best action to take at each step based on the previous and current node. These policies are then evaluated by the performer object to calculate the total reward associated with each policy. This improvement incorporates reinforcement learning ideas into the traditional A* algorithm, introducing a more academic and intelligent solution to the path planning problem.

3.3.2 Value Iteration

Value Iteration is a classical reinforcement learning algorithm for solving Markov Decision Processes (MDPs) with the aim of finding the optimal policy. In Value Iteration, the agent learns the optimal value function by interacting with the environment, thus achieving the goal without fully understanding the environment [36].

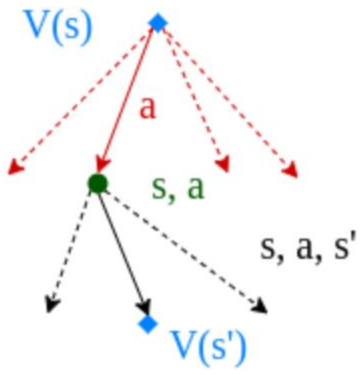


Figure 3.3: Value Iteration

The basic idea of Value Iteration is to iteratively update the value function until the optimal value function is found. The algorithm finds the optimal policy by calculating the optimal value function for each state from the state space by means of dynamic programming.The algorithm steps for Value Iteration are as follows:

1. Initialise the value function: For each state in the environment, a value function is initialised, either by random initialisation or by an initial value set according to a priori knowledge.

2. Update the value function: The current value function is updated by using the Bellman Equation. In Value Iteration, the current value function is updated by repeatedly applying the Bellman Equation until the value function converges.

$$V(s) = \max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')] \tag{3.11}$$

3. Policy improvement: After the value function has been updated, the policy is improved by selecting the action that has the maximum value in each state. This can be achieved by comparing the action value function for each state. The improved strategy may result in a change in the value function.

4. Repeat steps 2 and 3 until the value function converges: After each policy improvement, the value function update and policy improvement are performed again until the value function converges. Convergence of the value function means that the value function no longer changes and the resulting value function is the optimal value function.

5. Output the optimal strategy: After the value function converges, the optimal strategy is output, i.e. the action with the maximum value in each state

Algorithm 2: Value Iteration Algorithm

```
Data: $\theta$: a small number
Result: $\pi$: a deterministic policy s.t. $\pi \approx \pi_*$
Function ValueIteration is
    /* Initialization                                   */
    Initialize $V(s)$ arbitrarily, except $V(terminal)$;
    $V(terminal) \leftarrow 0$;
    /* Loop until convergence                           */
    $\Delta \leftarrow 0$;
    while $\Delta < \theta$ do
        for each $s \in S$ do
            $v \leftarrow V(s)$;
            $V(s) \leftarrow \max_a \sum_{s',r'} p(s',r|s,a)[r + \gamma V(s')]$;
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
        end
    end
    /* Return optimal policy                            */
    return $\pi$ s.t. $\pi(s) = arg\max_a \sum_{s',r'} p(s',r|s,a)[r + \gamma V(s')]$;
end
```

The advantage of Value Iteration is that the optimal policy can be obtained directly from the converged value function, and convergence is usually faster. In addition, Value Iteration requires less information about the environment model and does not rely on prior knowledge of the transfer probabilities and reward functions, making it more practical in some applications. However, the computational intensity of Value Iteration increases with the size of the state space, as it requires value function updates and policy improvements for all possible states and actions. Moreover, Value Iteration assumes complete awareness of the environment model, which may not be realistic in practical applications.

3.3.3 Policy Iteration

Policy Iteration is another classical reinforcement learning algorithm for solving Markov Decision Processes (MDPs), aiming to find the optimal policy. The basic idea of Policy Iteration is to optimise the policy and update the value function iteratively until the optimal policy is found. The algorithm searches for the optimal policy in the policy space by means of dynamic programming and updates the value function at the same time [37].

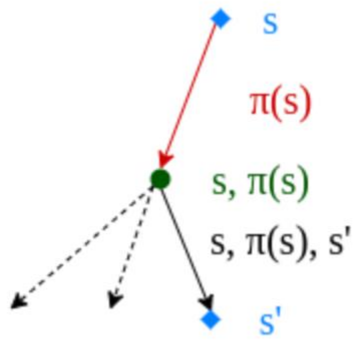


Figure 3.4: Policy Iteration

The algorithmic steps of Policy Iteration are as follows:

1. Initialise the policy: for each state in the environment, initialise a policy, either a random initialisation or an initial policy set according to a priori knowledge.

2. Policy evaluation: the current policy's value function is evaluated by computing a state value function. In Policy Iteration, the value function of the current policy is evaluated by iteratively applying the Bellman equation until the value function converges.

$$V(s) = \sum_{s',r'} p(s',r|s,\pi(s))[r+\gamma V(s')] \tag{3.12}$$

3. Strategy improvement: After the evaluation of the value function, the strategy is improved by selecting the action that has the maximum value in each state. This can be achieved by comparing the action value function (calculated by the Bellman equation) for each state. The improved strategy may lead to a change in the value function.

$$\pi(s) = arg\max_{a} \sum_{s',r'} p(s',r|s,a)[r+\gamma V(s')] \tag{3.13}$$

4. Repeat steps 2 and 3 until the strategy converges: after each strategy improvement, the strategy evaluation and strategy improvement are performed again until the strategy converges. Convergence means that the policy no longer changes and the resulting policy is the optimal policy.

5. Output the optimal strategy: After the strategy has converged, output the optimal strategy, i.e. the action with the maximum value in each state.

At the beginning, we are not concerned with whether the initial policy is optimal or not. During execution, we focus on improving it by repeating the policy evaluation and policy improvement steps in each iteration. Using this algorithm, we produce a policy chain in which each policy is an improvement on the previous one:

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_* \tag{3.14}$$

The advantage of Policy Iteration is that the optimal policy is obtained directly after the policy has converged and convergence is usually faster. Unlike Value Iteration, Policy Iteration updates only one policy in each iteration and therefore requires less computational resources

Algorithm 1: Policy Iteration Algorithm

```
Data: $\theta$: a small number
Result: $V$: a value function s.t. $V \approx v_*$, $\pi$: a deterministic policy
        s.t. $\pi \approx \pi_*$
Function PolicyIteration is
    /* Initialization                                                    */
    Initialize $V(s)$ arbitrarily;
    Randomly initialize policy $\pi(s)$;
    /* Policy Evaluation                                                 */
    $\Delta \leftarrow 0$;
    while $\Delta < \theta$ do
        for each $s \in S$ do
            $v \leftarrow V(s)$;
            $V(s) \leftarrow \sum_{s',r'} p(s', r|s, \pi(s))[r + \gamma V(s')]$;
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
        end
    end
    /* Policy Improvement                                                */
    policy-stable $\leftarrow$ $true$;
    for each $s \in S$ do
        old-action $\leftarrow \pi(s)$;
        $\pi(s) \leftarrow arg \max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$;
        if old-action != $\pi(s)$ then
            policy-stable $\leftarrow$ $false$;
        end
    end
    if $policy - stable$ then
        return $V \approx v_*$ and $\pi \approx \pi_*$;
    else
        go to Policy Evaluation;
    end
end
```

Chapter 4

Experimental Result

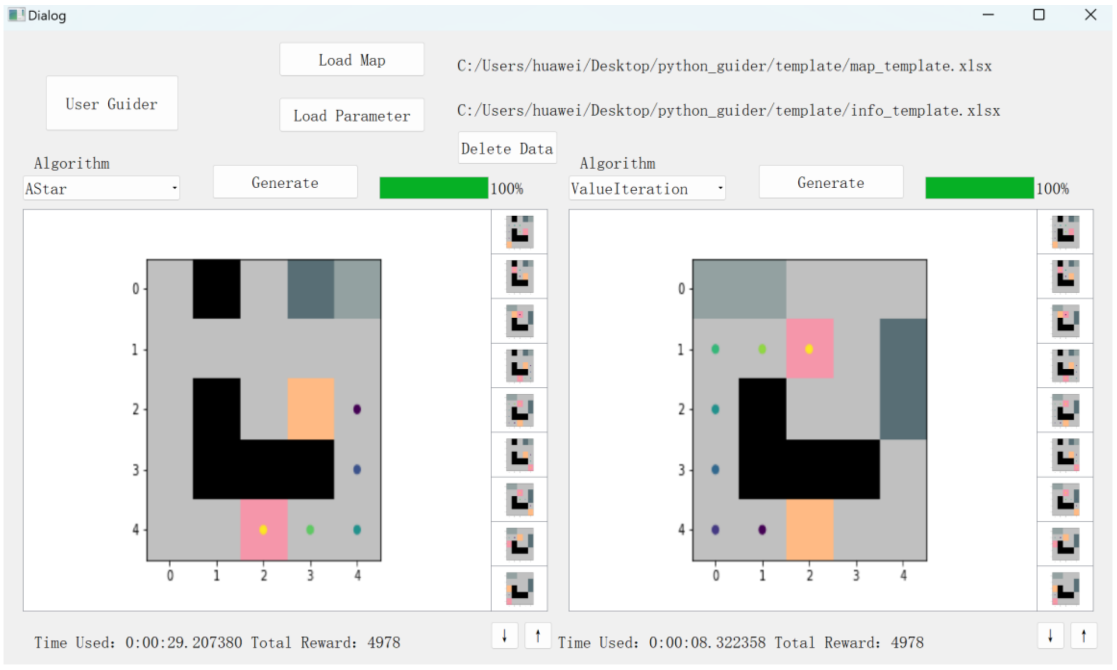
In this chapter, data processing and experimental results analysis are discussed. Section 4.1 examines the distinctions between the various algorithms, whereas Section 4.2 examines the results of the previous section and derives conclusions.

4.1 Algorithm Comparison

This section contrasts the experimental outcomes of three path planning algorithms: value iteration, policy iteration, and A*, and investigates additional algorithms based on the experimental outcomes. This section compares the outcomes in terms of used time, total rewards, and path.

4.1.1 Comparison Used Time

As shown in Figure 4.1, which is a 5x5 matrix, we first compare the difference in usage time between the A* algorithm and the value iteration algorithm. We observe that the A* algorithm requires more time than policy iteration and value iteration. This is due to the fact that the A* algorithm coordinates the search process by employing a heuristic function (also known as an estimation function) to select the search path that is closest to the objective. It must maintain a priority queue to select the next node in the search process, as well as calculate and update the value of the valuation function for each node, resulting in higher computational complexity and a longer runtime for the A* algorithm. On the other hand, Value Iteration finds the optimal strategy by iteratively updating the value function for each state until convergence. Iteration is more efficient than the A* algorithm because it does not require the maintenance of a priority queue or the calculation of the value of the evaluation function during the search process. However, Value Iteration requires iterative updates over the entire state space, which can result in higher computational complexity when the state space is large. Therefore, if the optimal path is computed in a map with a large space, the A* algorithm will take longer than value iteration.


Dialog

Load Map C:/Users/huawei/Desktop/python_guider/template/map_template.xlsx

User Guider

Load Parameter C:/Users/huawei/Desktop/python_guider/template/info_template.xlsx

Delete Data

Algorithm AStar Generate 100%

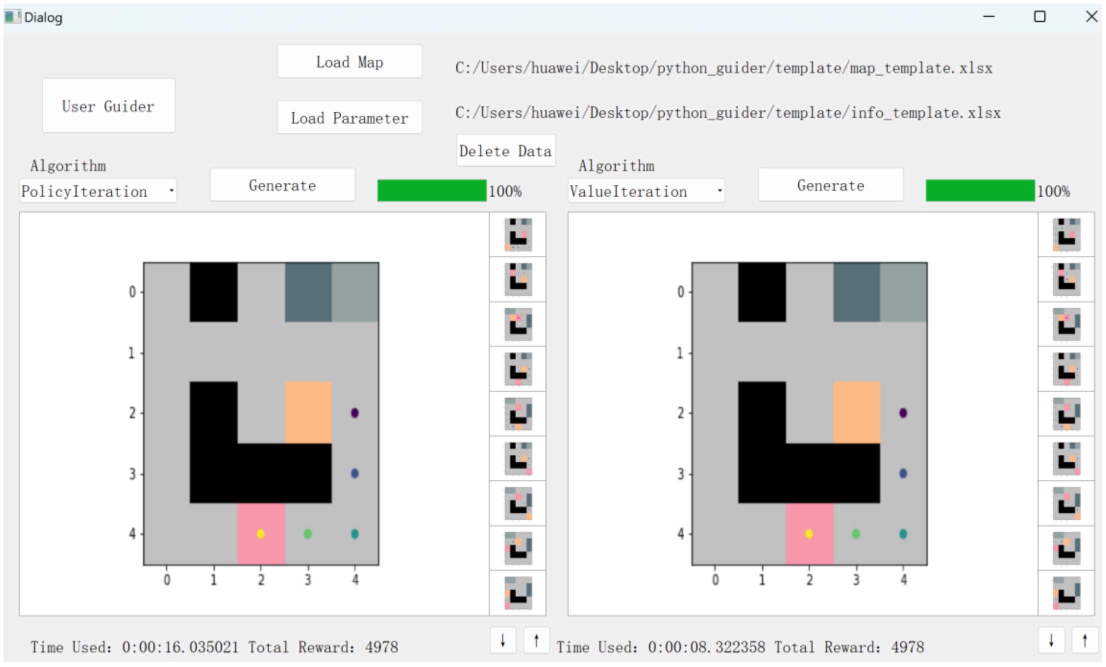
Algorithm ValueIteration Generate 100%

Time Used: 0:00:29.207380 Total Reward: 4978

Time Used: 0:00:08.322358 Total Reward: 4978

Figure 4.1: Time comparison between A* and value iteration

As shown in Figure 4.2, policy iteration requires more time than value iteration because the policy iteration algorithm must include both policy evaluation and policy improvement steps during each iteration, whereas the value iteration algorithm only requires an iterative update of the value function. In the value iteration algorithm, the value function can be directly modified at each iteration without explicit search or comparison of strategies. This may result in the value iteration algorithm being updated more rapidly at each iteration, thereby saving some time. Nevertheless, although the policy iteration algorithm may be slower, it is typically more stable than the value iteration algorithm because it explicitly evaluates and improves the strategy at each iteration, making it simpler to converge on the optimal strategy. In addition, the policy iteration algorithm typically guarantees an improvement in strategy with each iteration, whereas the value iteration algorithm only guarantees an improvement in the value function.


Dialog

Load Map C:/Users/huawei/Desktop/python_guider/template/map_template.xlsx

User Guider

Load Parameter C:/Users/huawei/Desktop/python_guider/template/info_template.xlsx

Delete Data

Algorithm PolicyIteration Generate 100%

Algorithm ValueIteration Generate 100%

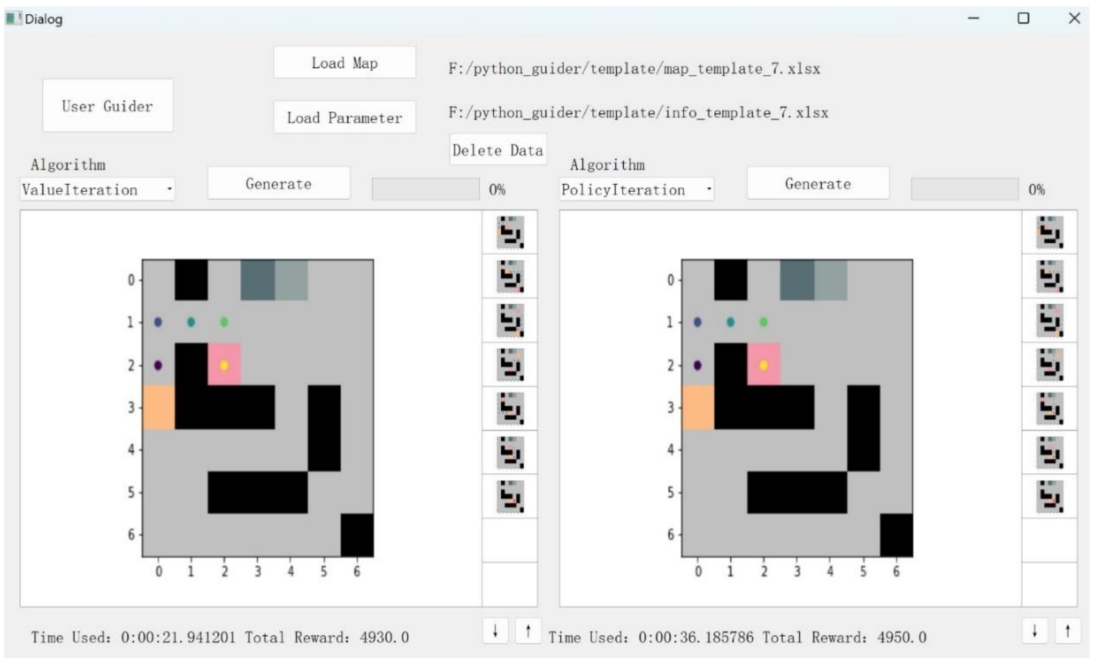
Time Used: 0:00:16.035021 Total Reward: 4978

Time Used: 0:00:08.322358 Total Reward: 4978

Figure 4.2: Time comparison between policy iteration and value iteration

4.1.2 Comparison Total Reward

After accounting for complexity, the total reward of the A* algorithm may be less than that of the other two reinforcement learning algorithms, due to the addition of a similar policy to the conventional A* algorithm for reinforcement learning, making the rewards comparable. Value Iteration is typically more appropriate for problems with a limited state space, due to the increased dimensionality and computational complexity of the value function. On the other hand, Policy Iteration optimizes the cumulative reward by iteratively updating the state's policy and determining the optimal policy. Policy Iteration performs policy evaluation and calculates the value function under the current policy in each iteration, followed by policy improvement and policy updates until convergence. Therefore, when the state space is large, Value Iteration may consider some unnecessary paths, resulting in a lower total reward, whereas Policy Iteration's update process is relatively complex, involving alternate updates between the value function and the policy, but since only the value function and the policy under the current policy are updated, it does not consider those states that will not be visited in the actual path planning. However, since only the value function and policy under the current policy are modified, states that will not be visited during actual path planning are not taken into account, and a superior path may be obtained. Comparing the three algorithms, Policy Iteration may result in the highest total reward when the state space is larger and more intricate.



F:/python_guider/template/map_template_7.xlsx

F:/python_guider/template/info_template_7.xlsx

Algorithm ValueIteration

Algorithm PolicyIteration

Time Used: 0:00:21.941201 Total Reward: 4930.0

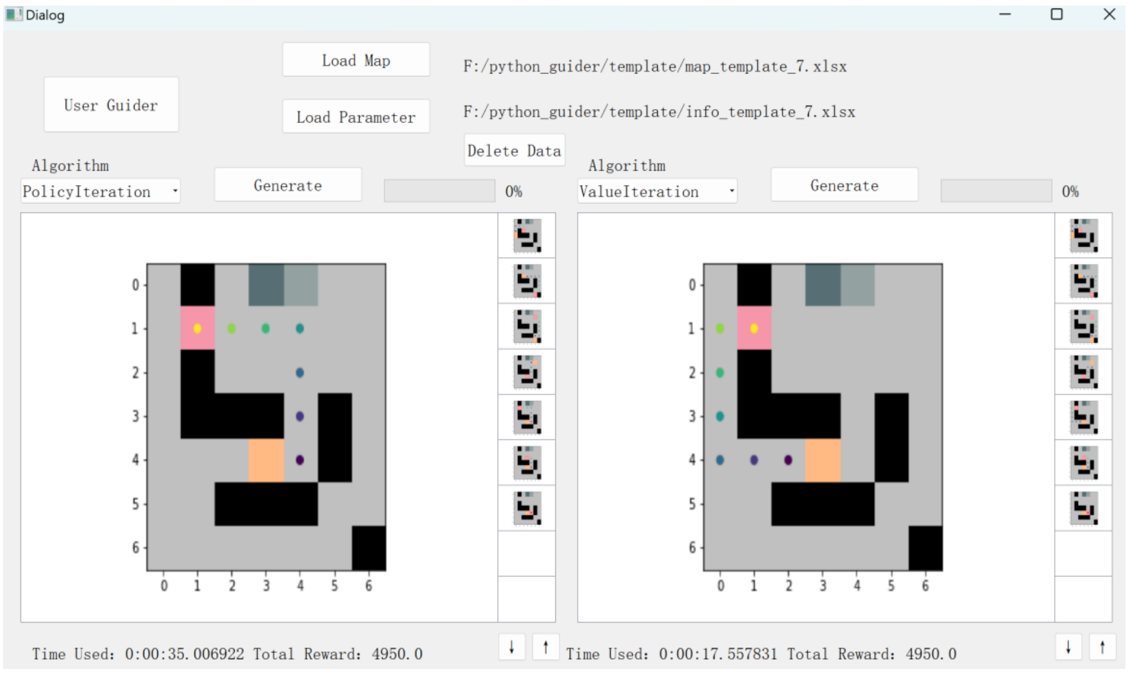
Time Used: 0:00:36.185786 Total Reward: 4950.0

Figure 4.3: Reward comparison between policy iteration and value iteration

4.1.3 comparison Path

As shown in figure 4.4, the A* algorithm utilizes a heuristic function (typically an estimated target distance) to guide the search process and efficiently identify the optimal path in the search space. Usually, the A* algorithm selects the node with the lowest total estimated cost in the next phase as the expansion node at each step, moving towards the target location. However, the A* algorithm only considers the distance between the current state and the goal state without taking into account other factors such as the value function of the state or the policy, which may result in generating a route that goes directly to the goal location without considering other factors such as the environment's reward or the long-term value of the state.

On the other hand, Value Iteration and Policy Iteration are reinforcement learning algorithms based on dynamic programming that aim to solve optimal value functions and policies. Both algorithms iteratively update a state's value function and policy to determine the optimal path. Value Iteration modifies the value function of all states during each iteration, while Policy Iteration compares the value function to the current policy and updates the policy accordingly. When updating the value function and policy, both algorithms take into consideration the environment's reward and the state's long-term value, which may result in a more integrated route.



F:/python_guider/template/map_template_7.xlsx

F:/python_guider/template/info_template_7.xlsx

Algorithm: PolicyIteration

Algorithm: ValueIteration

Time Used: 0:00:35.006922 Total Reward: 4950.0

Time Used: 0:00:17.557831 Total Reward: 4950.0

Figure 4.4: Path comparison between policy iteration and value iteration

4.2 Algorithm Analysis

In this section, an extended analysis based on the experimental results in Section 4.2 will be conducted to investigate the differences in the practical application of the majority of the algorithms involved in path planning and to analyse the advantages, disadvantages, and generalizability between the various algorithms in relation to the experimental data.

4.2.1 How to choose Algorithm

The selection of an appropriate path planning algorithm should be based on several factors, including problem difficulty, the characteristics of the environment, the availability of computational resources, and the performance requirements of the algorithm. The following principles can be taken into account:

1. Problem difficulty: Consider using a search-based algorithm such as A*, Dijkstra, etc. if the problem has a small and straightforward state and action space. These algorithms typically perform well on small-scale problems and can find optimal solutions.

2. Ecology of the environment: If the problem's environment is stochastic or characterized by incomplete information, such as partially observable Markov decision processes, reinforcement learning-based algorithms such as Q-learning, SARSA, etc. may be required. These algorithms can autonomously learn the optimal policy by interacting with the environment and are adaptive to the environment's lack of complete information.

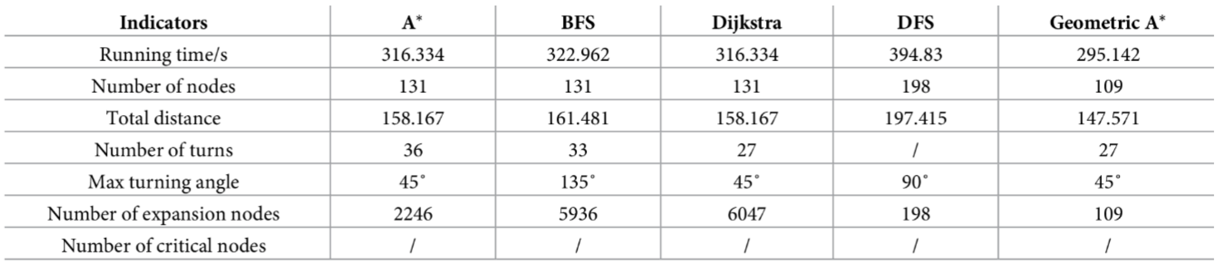
3. Availability of computational resources: Different algorithms have varying computational resource requirements. Some search-based algorithms may have low memory and computational power requirements and are suitable for use when computational resources are limited, whereas some reinforcement learning-based algorithms may require larger computational resources and sample sizes for training and are suited for use when computational resources are abundant.

4. Performance requirements of algorithms: The performance requirements of algorithms vary from one algorithm to another. For example, some algorithms may be more time-efficient but may not find the optimal solution, while others may find the optimal solution but require more time. Therefore, the nature of the problem and the algorithm's performance requirements must be considered when selecting an algorithm.

In conclusion, the selection of an appropriate path planning algorithm should be based on the problem's requirements and resource limitations, taking into account theoretical and practical research, experimentation, and comparison of various algorithms."

4.2.2 Difference between Algorithms

We first evaluate the performance of several algorithms other than reinforcement learning. The following table compares A*, BFS, Dijkstra, DFS, and Geometric A*, which is based on the enhanced A* algorithm [38]. Depth First Search (DFS) is a blind search algorithm that focuses solely on the depth of the current path without considering the length of the path during the search. This explains why it performs the worst according to the table, as shown in Figure 4.5. This means that DFS may traverse deeper branches during the search process, resulting in paths that are not necessarily the shortest paths and may require more time to find. As described in Section 3.3.1, on the other hand, Geometric A* employs a pruning strategy during the search to reduce the search space by eliminating implausible paths.



Indicators	A*	BFS	Dijkstra	DFS	Geometric A*
Running time/s	316.334	322.962	316.334	394.83	295.142
Number of nodes	131	131	131	198	109
Total distance	158.167	161.481	158.167	197.415	147.571
Number of turns	36	33	27	/	27
Max turning angle	45°	135°	45°	90°	45°
Number of expansion nodes	2246	5936	6047	198	109
Number of critical nodes	/	/	/	/	/

Figure 4.5: Compares A*, BFS, Dijkstra, DFS, and Geometric A*

Five traditional map-based algorithms, including Dijkstra, A*, D*, LPA*, and D* Lite, are compared in terms of search direction, heuristics, incrementality, applicability, and practical applications [39], as shown in Figure 4.6. Here are some benefits and drawbacks of each:

Search Direction:

Dijkstra: Forward search. It works well in a static environment where global map information is known.

A*: Direct search. Similar to Dijkstra, it is effective in static environments where global map information is known.

D*: Reverse search. It can handle unknown maps in dynamic environments and update the optimal path from the current point to the target point based on node distance data from previous iterations.

LPA*: Forward search with incremental updates. It assumes that the remainder of the path is clear and incrementally modifies the path. It is beneficial when only partial knowledge is available.

D* Lite: Reverse search with iterative updates. Similar to LPA*, it incrementally adjusts the path and performs well in dynamic environments.

Heuristic and Non-heuristic:

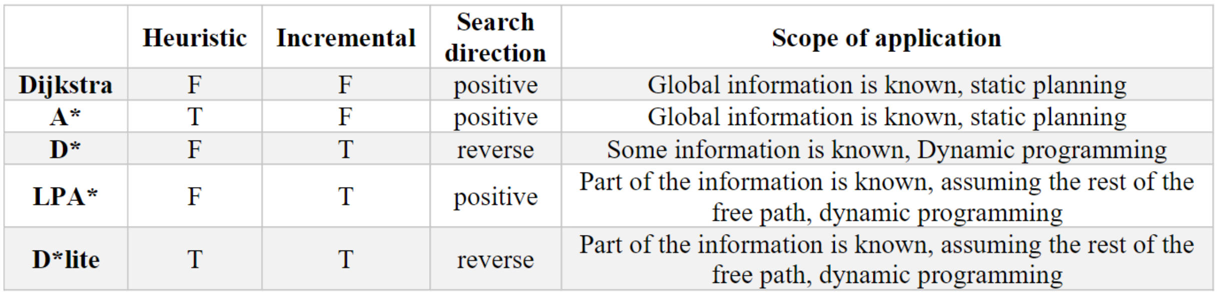
Heuristic algorithms (such as A* and Genetic Algorithms) use heuristic functions to guide the search, resulting in more efficient searches by avoiding global naive searches.

Non-heuristic algorithms (such as Dijkstra) do not use heuristic functions and may traverse the periphery without rules, which can sometimes result in decreased efficiency.

Heuristic and Incremental:

Heuristic search enhances time efficiency by using heuristic functions to direct the search towards the target location.

Incremental search re-uses previous search results for a more efficient search, reducing the search's scope and duration. Examples of incremental search algorithms are LPA* and D* Lite."



	Heuristic	Incremental	Search direction	Scope of application
Dijkstra	F	F	positive	Global information is known, static planning
A*	T	F	positive	Global information is known, static planning
D*	F	T	reverse	Some information is known, Dynamic programming
LPA*	F	T	positive	Part of the information is known, assuming the rest of the free path, dynamic programming
D*lite	T	T	reverse	Part of the information is known, assuming the rest of the free path, dynamic programming

Figure 4.6: Comparison of performance and application of traditional algorithms

Policy Iteration and Value Iteration, both utilised in reinforcement learning, differ primarily in their update methods: Policy Iteration and Value Iteration have different methods for updating policies and value functions. In Policy Iteration, the value function under the present policy is first calculated via a Policy Evaluation step, and then revised via a Policy Improvement step. In the value iteration, the optimal value function is derived directly from the Bellman Optimality Equation for the value function, from which the optimal policy is derived. Iteration: The manner in which strategy iteration and value iteration iterate differs. Strategy iteration is typically an iterative process that alternates between strategy evaluation and strategy refinement until the optimal strategy is reached. Value iteration, on the other hand, modifies the value function at every time step and obtains the optimal strategy when the value function converges. The value iteration typically converges faster than the strategy iteration due to the fact that the value iteration modifies the value function directly via the Bellman optimality equation for the value function, whereas the strategy iteration requires two stages of alternating iterations. However, the strategy iteration may acquire a better strategy with each iteration, whereas the value iteration may begin performing actions before the value function has completely converged, resulting in strategy instability.

4.2.3 Generalizability of Algorithm

In path planning, the generalizability of an algorithm refers to whether the path planning strategy it learned during training can be generalised to unseen environments, i.e., whether it can still generate plausible paths in a new map environment or start-end configuration. We can calculate the optimal route on a new map using the model. This substantially enhances computational efficiency, but if the algorithm is less generalizable, the obtained results will not be as excellent, i.e. the total reward will be lower than with other algorithms. Common path planning algorithms have the following generalisation properties:

Algorithms based on graph search (e.g. Dijkstra, A*, etc.): these algorithms typically lack an explicit model update and parameter adjustment process during training, so their generalizability is primarily determined by the algorithm's strategy and the selection of heuristic function during the search. If the heuristic function can estimate distances or costs reasonably well and is adaptable to various types of map environments, then these algorithms typically have excellent generalisation, but in general they do not perform very well.

Policy Iteration and Value Iteration are examples of reinforcement learning algorithms. These algorithms learn by interacting with the environment and choosing actions based on the current state to maximise cumulative returns. This learning process enables the algorithm to generalise to unobserved environments, but the algorithm's generalisation performance may be affected by the constraints of the training environment and the efficacy of the samples. For instance, if the training environment differs significantly from the test environment in terms of map characteristics, obstacle distribution, etc., the algorithm's generalisation performance may be affected; however, where map variability is negligible, this type of algorithm's generalisation performance is superior to that of algorithms based on graph search.

Deep learning algorithms (e.g. DQN): These algorithms perform path planning by acquiring knowledge of the mapping relationships between inputs and outputs. Typically, deep learning algorithms have robust fitting and characterization capabilities, but their generalisation performance may be hampered by data distribution and overfitting. In order to guarantee the algorithm's generalisation efficacy, it is necessary to take into account all possible test-environment scenarios during training.

The generalisation efficacy of algorithms in path planning is affected by a number of factors, such as algorithm selection, the design of the training environment, the quality and quantity of data samples, etc. In practical applications, these factors must be considered, along with algorithm selection, parameter optimisation, and model validation, to ensure that the path planning algorithm has excellent generalisation performance in unknown environments.

Chapter 5

Implementation

In this chapter, the implementation of this project will be primarily described, which is divided into three sections. The programming language, libraries, and runtime environment are introduced in Section 5.1. Section 5.2 introduces the program's front-end design, including the user guider. The back-end design of the program, including the info template and map template components that the user must upload, is described in section 5.3.

5.1 Environment, Languages, and Libraries

The operating system WINDOWS 10/11 has been selected as the running environment for this project to ensure compatibility with the requisite tools and libraries. Python will be used as the programming language for this project, as it is a well-structured, uncomplicated, and fast-running language. Python is more suitable than other languages for duties involving data processing. The following Python libraries are mainly used:

pandas: A library for data processing and data analysis, including operations such as DataFrame creation, data filtering, data merging, pivot tables, etc.

numpy: A library for numerical computation and array manipulation, including operations on arrays, indexing, etc.

matplotlib.pyplot: A library for data visualisation, including graphing, scatter plotting and other operations.

5.2 Front End Design

This section presents the front-end design of the program, which was developed using PyQt5. It helps the user to view the best paths, as well as presenting experimental data under different algorithms.

5.2.1 User Guider

This program is designed for calculating and presenting the optimal path for an autonomous mobile robot. It has been developed using PyQt5, allowing users to upload a map file and a parameter file in order to obtain the best path results and corresponding data using different algorithms.

Usage Instructions:

1. Load the map file: Click the "Load Map" button and select the map file to load. Once successfully loaded, the file address will be displayed on the interface.
2. Load Params: Click the "Load Params" button and select a parameter file to load. Once successfully loaded, the file address will be displayed on the interface.

3. Select and switch algorithms: Choose an algorithm from the drop-down bar in the "Algorithm" section, such as "AStar", "ValueIteration", or "PolicyIteration".
4. Calculate the best path: Click the "Generate" button to display the calculated best path on the interface.
5. Switch images: Use the up and down arrow symbols to navigate between the displayed thumbnail images.
6. View image: Click on a thumbnail image in the queue to view a larger image.
7. Clear data: Click the "Delete Data" button to clear the loaded map data and result data.
8. Display results: The "Results" column will show the "Time Used" and "Total Reward" results once the calculation is complete.

Caution:

1. Ensure that the map file and parameter file are in .xlsx format and comply with the program's required format when loading them.
2. Make sure to load the map file and parameter file and select the desired algorithm before performing image processing.
3. Before uploading new map data, clear the data or restart the program.
4. If the program encounters an exception or error, it can be resolved by closing and restarting the program.

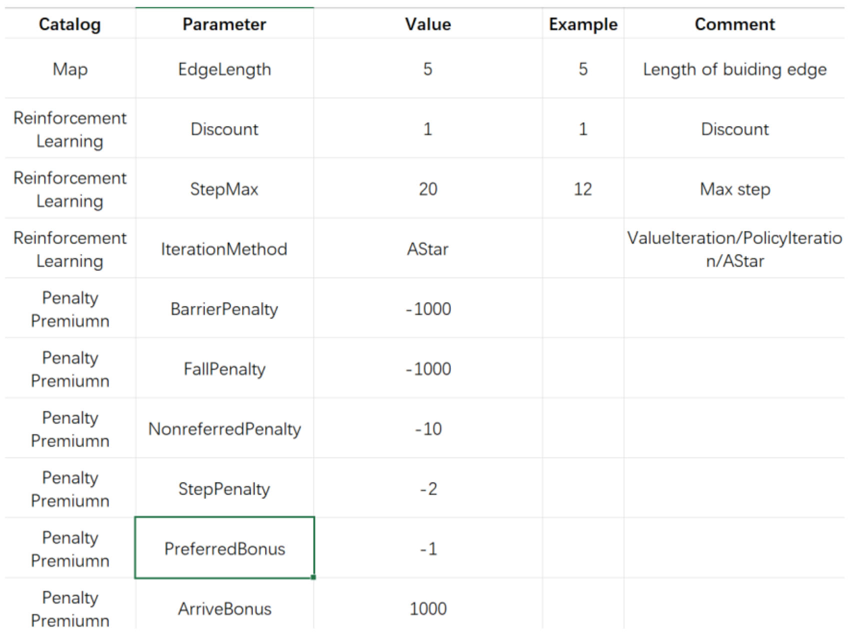
5.3 Back End Design

This section introduces the back-end design of the program, mainly introducing the two files that need to be uploaded by the user: info template and map template, both of which are .xlsx files with data set by the user according to their individual needs. The program needs the map data uploaded by the user in order to get the best route calculation results.

5.3.1 info template

In this .xlsx file there are 3 sections: paramter, point_role and role_color.

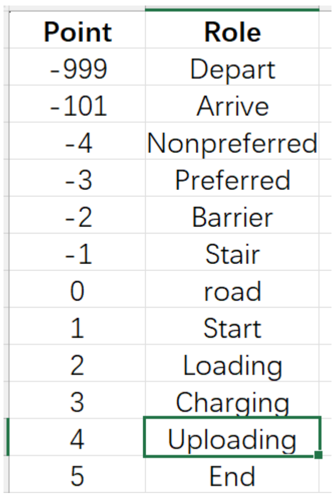
Parameter: sets and stores all the data parameters in the map, enter the number of cells per row of the map in the "EdgeLength" column, e.g. "5", to resize the map. IterationMethod indicates the algorithm used in the current environment. ArriveBonus are all set as defaults and the results can be changed by changing the value parameter assigned to them when changing requirements. The figure below is an example of a 5*5 map that performs the AStar algorithm.



Catalog	Parameter	Value	Example	Comment
Map	EdgeLength	5	5	Length of buiding edge
Reinforcement Learning	Discount	1	1	Discount
Reinforcement Learning	StepMax	20	12	Max step
Reinforcement Learning	IterationMethod	AStar		ValueIteration/PolicyIteration/AStar
Penalty Premiumn	BarrierPenalty	-1000		
Penalty Premiumn	FallPenalty	-1000		
Penalty Premiumn	NonreferredPenalty	-10		
Penalty Premiumn	StepPenalty	-2		
Penalty Premiumn	PreferredBonus	-1		
Penalty Premiumn	ArriveBonus	1000		

Figure 5.1: Parameter in info template

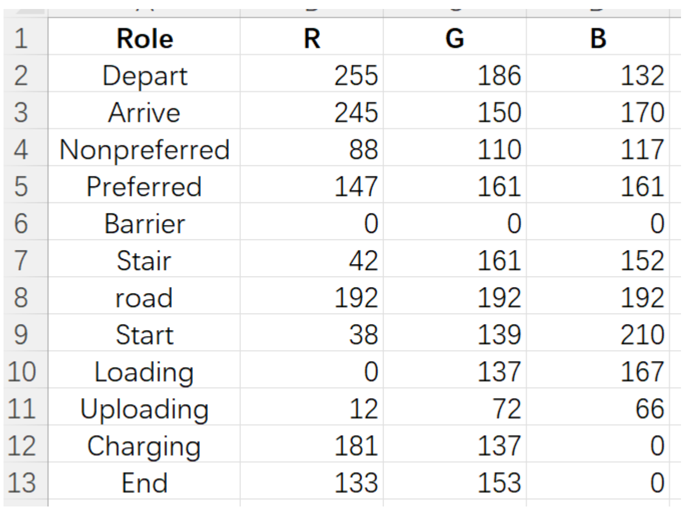
Point_role: Sets and stores the queue of tasks to be performed by the robot, including start, loading, uploading, charging and end.



Point	Role
-999	Depart
-101	Arrive
-4	Nonpreferred
-3	Preferred
-2	Barrier
-1	Stair
0	road
1	Start
2	Loading
3	Charging
4	Uploading
5	End

Figure 5.2: Print_role in info template

Role_color: Sets and stores the colour of the corresponding area of the map, including nonperferred, preferred, stair, road, barrier, etc.



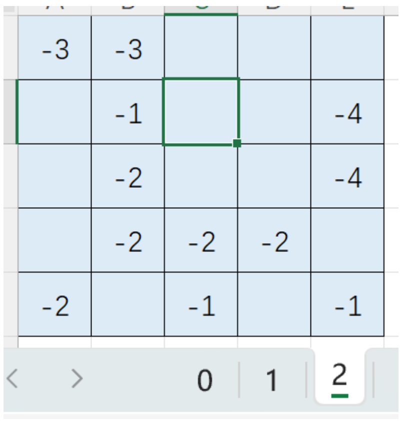
	Role	R	G	B
1	**Role**	**R**	**G**	**B**
2	Depart	255	186	132
3	Arrive	245	150	170
4	Nonpreferred	88	110	117
5	Preferred	147	161	161
6	Barrier	0	0	0
7	Stair	42	161	152
8	road	192	192	192
9	Start	38	139	210
10	Loading	0	137	167
11	Uploading	12	72	66
12	Charging	181	137	0
13	End	133	153	0

Figure 5.3: Role_color in info template

5.3.2 map template

The decision to use a grid map in this research is motivated by the desire to avoid dealing with an infinite state space that would arise with a continuous coordinate system. Although there are deep reinforcement learning models like DQN, A2C, and DDPG that can handle infinite state space, using such models may deviate from the goal of exploring classical reinforcement learning models like dynamic programming.

The map used in this research is a square area with a size of N*N, which is drawn by the user in the "map_template.xlsx" file. The robot queue is filled with numbers representing the locations on the map where the robot needs to perform tasks. As shown in the figure 5.4, the example map provided has 3 floors and a size of 5x5.



-3	-3			
	-1			-4
	-2			-4
	-2	-2	-2	
-2		-1		-1

Figure 5.4: Map template

Chapter 6

Summary and Reflection

6.1 Project Management

The development of the application will occupy the most crucial part of the project, as it will require a large amount of code to be written. The Gantt chart below depicts the project plan (Figure 6.1 and 6.2). I plan to complete more work in the first half of this year. The goal is to complete the central part of the plan by 2023. Complete the majority of the software development in the first semester. Start writing the final report in the second semester and improve and optimize the details of the software

A. Write and submit the proposal.

B. Prepare and submit ethics forms.

C. Reference to relevant literature to identify user needs.

D. Designing software prototypes.

E. Determining the best algorithm for computing path planning.

F. Development of main procedures.

G. Write the Interim Report.

H. Gather feedback on the software and refine the interface details.

I. Write and submit final report writing.

The project started on October 11, 2022, and is expected to be completed by April 20, 2023, for a total of 138 days. Some tasks can be carried out simultaneously. For example, the proposal and ethics form can be carried out simultaneously, and the deadline for this part is October 21, 2022. The next step is to define the user requirements for the software, identifying all the features that need to be implemented through communication with the tutor and reviewing relevant literature, which is expected to be completed by November 7, 2022. The next step is designing high prototypes of the software, which is essential for the construction of the software and requires consideration of the interface layout of the software, which is expected to be completed by November 14, 2022. The next step is to define the algorithms for calculating the route planning in the software, which is the core work of the software code and requires a comparison to get the algorithm that can calculate the best route, which is expected to be completed by November 21, 2022. The next step is to write the software code, and this part can be coupled with the completion of the interim report, due on December 9, 2022. Due to exam revision, there are no other tasks until the end of spring break. When I return to school in the spring, I will test the software, make improvements based on user feedback and further refine its functionality, and this part is expected to be completed by April 3, 2023. Also, I need to complete the final report by April 20.

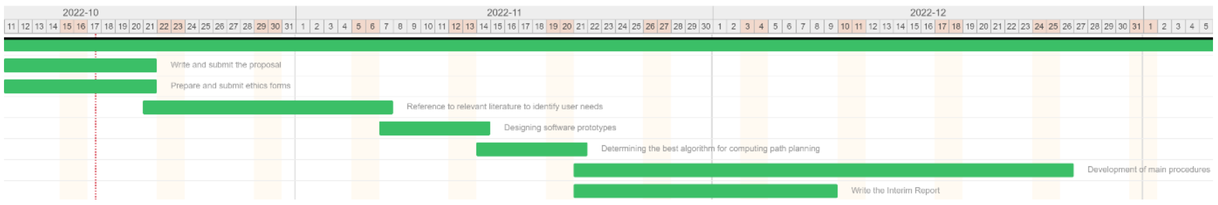


Figure 6.1: Project Schedule for Semester 1.

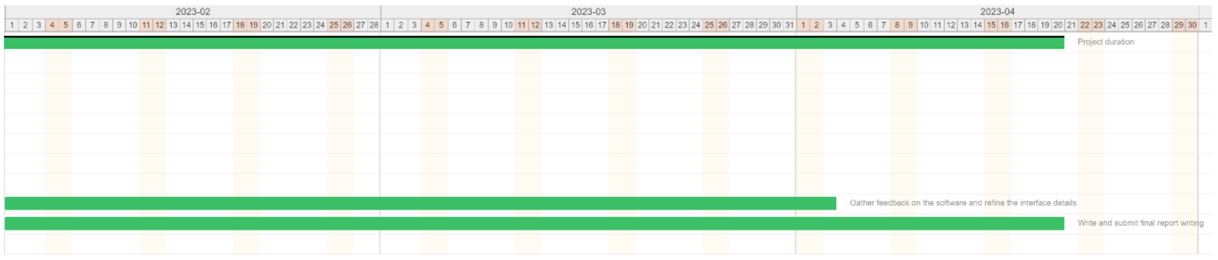


Figure 6.2: Project Schedule for Semester 2.

6.2 Reflection

This section presents the experiences and lessons learned during the development of the project, summarises the difficulties encountered and the directions for the development of the software, which will help in the project's future development.

6.2.1 How to face Failure

The process of developing software inevitably includes several failures and mistakes that can frustrate software developers. For example, in choosing the algorithm for the best route, my initial idea was to use traditional A* and Dijkstra algorithms. However, after actual testing, it became clear that using just a single algorithm was not sufficient for calculating the best route, and even using multiple algorithms at the same time did not yield satisfactory results. Especially after giving complex instructions, the software took a long time to generate results. This realization meant that I had not made the right choice of algorithm, and as a result, much of my previous work would be wasted, significantly impacting my project schedule.

In discussions with my supervisor, I found a new way to solve the problem, which was to use reinforcement learning intelligent algorithms. Faced with failure, I learned that it's important to adjust our attitude and approach the problem positively. It's also crucial to free our minds and think from multiple perspectives. Failure can be seen as an opportunity to improve the quality of research.

6.3 Time Management

Good time management is a prerequisite for completing a project and needs to be carefully considered. During my research, I realized that time is the most valuable and scarce resource for software developers, especially students. A time plan for the entire project was developed before the project started, and the research direction was clearly defined. However, the time plan outlined in the proposal turned out to be unsuitable for the actual research project. I encountered many detours during the development process, resulting in exceeding the estimated time at each milestone. Additionally, the project was further delayed due to the epidemic, which prevented me from conducting continuous research for an extended period. This delay required me to spend more time reviewing previous research and literature content, leading to an increased allocation of time for literature review.

After making several adjustments, I rearranged my schedule, which had a positive effect in terms of pushing and guiding the project. I encountered many unpredictable processes, errors, and challenges during the research process, as described in Section 5.1. For instance, using the wrong path algorithm early in the project resulted in wasting much time. Moreover, since I needed to learn reinforcement learning, I spent a significant amount of time understanding the work involved in reinforcement learning, which further delayed my project planning. Hence, it is crucial to start the project early and allocate enough time to deal with unexpected failures.

6.4 Importance of Pioneering Thinking

One of the most valuable qualities of a software developer is a pioneering mind. Researchers with innovative ideas find it easier to make significant research contributions. For example, using intelligent algorithms instead of being limited to traditional ones, learning through interaction and feedback with the environment. Therefore, I will continue to think outside the box and broaden my ideas to find optimal solutions in future research.

During my investigation of algorithmic differences and generalizations, I have realized the significance of maintaining an open mind. We can only comprehend the strengths and shortcomings of various algorithms, as well as their applicability and efficacy in different situations, with an innovative mindset. Path planning problems involve numerous factors, such as map dimensions, start and end point locations, distribution of obstacles, and others. Different algorithms operate under distinct assumptions and applicability conditions, and it is only through an open mind and extensive research that we can choose the optimal algorithm to solve the problem at hand.

Open-mindedness not only has immense academic significance in the field of algorithmic studies, but it also serves as a governing principle in our daily lives. We must think beyond conventional boundaries in our daily endeavors to effectively adapt to changing circumstances and challenges. Creative thinking fosters a deeper comprehension of concepts beyond what is simply taught, encouraging independent thought and exploration of the essence and practical applications of knowledge among students. Open-mindedness requires continuous self-challenge and self-improvement, acceptance of new ideas and concepts, and ongoing learning and experimentation. To arrive at profound and comprehensive solutions, we must proactively consider multiple perspectives and possibilities, delving beyond superficial solutions to discover the underlying principles and mechanisms."

6.5 The Future of Project

Although I have made significant progress in preparing the core algorithms for the software, there is still ample room for improvement. Currently, the software can only be used on the computer side to help users understand the best route for the robot, but it cannot be connected to the robot and its hardware. In future planning, it will be necessary to establish a connection between the map designer and the robot, and integrate it with the sensors that the robot carries to scan its surroundings. The results of the sensor scans can then be uploaded to the map designer, enabling more accurate calculations and attempts to generate corresponding maps in the software for practical use by the user.

Another limitation of the current program is that it only supports switching between three algorithms, which may not be sufficient for a comprehensive comparison of generalizations between algorithms. In the future, more algorithm options could be introduced to provide a clearer comparison. Additionally, the front-end part of the program currently only supports the upload of .xlsx files. To improve the ease of use and user experience, maps could be designed in a more intuitive way in the program's interface.

In summary, future planning should focus on improving the connection between the map designer and the robot, introducing more algorithm options, and enhancing the front-end design of the program to further improve its functionality, performance, and user experience. This will provide more possibilities and convenience for research and applications in the field of robot path planning.

6.6 Conclusion

This project presents a comprehensive investigation into path planning for autonomous mobile robots by designing and implementing a map designer that generates optimal paths using the A* algorithm, Policy Iteration algorithm, and Value Iteration algorithm. The study examines and analyzes the distinctions and generalizations of these three algorithms in path planning, providing a comprehensive theoretical basis and experimental results for future research and applications of robot path planning algorithms.

Initially, a comprehensive survey of relevant research in the field of robot path planning is conducted through multiple literature reviews, providing an in-depth comprehension of the principles and application scenarios of the A*, Policy Iteration, and Value Iteration algorithms. The development and implementation of the map designer adhere to a stringent scientific methodology, employing appropriate experimental design and data processing techniques to ensure the map's reliability and validity.

The next section of this paper compares and contrasts the A* algorithm, Policy Iteration algorithm, and Value Iteration algorithm, discussing their advantages and disadvantages, application scenarios, and implementation methods in path planning. Through the implementation of the map designer, the efficacy of these algorithms in various scenarios is experimentally contrasted, primarily in terms of time, total reward, and path. The experimental results provide practical conclusions and a scientific foundation for choosing the optimal path calculation algorithm. In addition, the study examines the generalizability of the algorithms in depth and plans to compare their performance in the future, which is essential for optimizing the performance of path planning algorithms in practical applications, thereby enhancing the adaptability and intelligence of autonomous mobile robots in a variety of environments.

The study provides a comprehensive understanding of the benefits of reinforcement learning in path planning, as well as the applicability and efficacy of the A* algorithm, Policy Iteration algorithm, and Value Iteration algorithm in various situations. In addition to analyzing and comparing several other path planning algorithms based on the experimental results and other references, the study also recognizes the limitations of the experimental design and data processing, as well as the inadequacy of the performance evaluation of the algorithms, which must be continuously improved in future research.

References

[1] Mary B. Alatise. (2020). A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods. https://ieeexplore.ieee.org/abstract/document/9007654.

[2] Y. Jin-Xia, C. Zi-Xing, D. Zhuo-Hua and Z. Xiao-Bing. (2006). Design of dead reckoning system for mobile robot, vol. 13, pp. 542-547.

[3] "Autonomous Mobile Robots for Piece Picking Applications." https://www.guidanceautomation.com/autonomous-mobile-robots-amr/.

[4] "In-Position Technologies." https://iptech1.com/amr-technology/.

[5] Algorithms.N. Sariff. (2006). An Overview of Autonomous Mobile Robot Path Planning. https://ieeexplore.ieee.org/abstract/document/4339335.

[6] A. Saudabayev, F. Kungozhin, D. Nurseitov and H. A. Varol. (2015). Locomotion strategy selection for a hybrid mobile robot using time of flight depth sensor, pp. 1-14.

[7] D. Kortenkamp. (1994). Perception for mobile robot navigation: A survey of the state of the art, pp. 446-453.

[8] "Mobile industrial robots." https://www.mobile-industrial robots.com/solutions/mir-applications/mir-fleet/.

[9] M. Al Marzouqi. (2011). Efficient path planning for searching a 2-D grid-based environment map, in 2011 IEEE GCC Conference and Exhibition, pp.2-50.

[10] Jason D.Ramsdalea and Matthew R.Balme. (2017). Planetary and Space Science, pp.49-61.

[11] Thrun S. (1998). Learning metric-topological maps for indoor mobile robot navigation[J]. Artificial Intelligence, pp.21-77.

[12] "Geometric Feature-based Edge-Matching." http://www.geocomputation.org/1998/99/gc_99.htm/.

[13] Shiqiang Yang and Guohao Fan. (2020). MGC-VSLAM: A Meshing-Based and Geometric Constraint VSLAM for Dynamic Indoor Environments, IEEE Access (Volume: 8), pp.81007-81021.

[14] Shan-Shan Xue, Yong Zhang, Jin Cheng and Qin-Jun Zhao. (2014). Topological map building for mobile robots based on thining algorithm, in 2014 11th International Computer Conference on Wavelet Actiev Media Technology and Information Processing (ICCWAMTIP).

[15] S. M. Bhagya P. Samarakoon and M. A. Viraj J. Muthugala. (2022). Global and Local Area Coverage Path Planner for a Reconfigurable Robot, in 2022 IEEE Congress on Evolutionary Computation (CEC), pp.50-72.

[16] Ade Candra; Mohammad Andri Budiman and Kevin Hartanto. (2020). Dijkstra's and A-Star in Finding the Shortest Path: a Tutorial, in 2020 International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA), pp.33-124.

[17] Shao-Hung Chan; Ping-Tsang Wu; Li-Chen Fu. (2018). Robust 2D Indoor Localization Through Laser SLAM and Visual SLAM Fusion, in 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp.25-98.

[18] João Pedro Mucheroni Covolan, Antonio Carlos Sementille and Silvio Ricardo Rodrigues Sanches. (2020). A mapping of visual SLAM algorithms and their applications in augmented reality, in 2020 22nd Symposium on Virtual and Augmented Reality (SVR), pp.15-177.

[19] "How does a robot plan a path using RRT?" https://towardsdatascience.com/how-does-a-robot-plan-a-path-in-its-environment-b8e9519c738b.

[20] Le Lyu, Yang Shen and Sicheng Zhang. (2022). The Advance of Reinforcement Learning and Deep Reinforcement Learning, in 2022 IEEE International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA).

[21] "Gym Documentation." https://www.gymlibrary.dev/content/basic_usage/.

[22] Mayank Banoula. (2022). "What Is Q-Learning: The Best Guide To Understand Q-Learning." https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-q-learning.

[23] John Schulman and Filip Wolski. (2017). "Proximal Policy Optimization Algorithms." https://arxiv.org/abs/1707.06347.

[24] Mnih, Volodymyr. (2015). "Human-level control through deep reinforcement learning." ,pp.529-533.

[25] "Welcome to Deep Reinforcement Learning Part 1 : DQN." https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b

[26] Lv, L., Zhang, S., Ding, D., & Wang, Y. (2019). Path Planning via an Improved DQN-Based Learning Policy. IEEE Access, 7, 67319-67330. doi: 10.1109/ACCESS.2019.2918703.

[27] WILL KENTON. (2022). "Monte Carlo Simulation: History, How it Works, and 4 Key Steps." https://www.investopedia.com/terms/m/montecarlosimulation.asp.

[28] Andrew G. Barto. (2007). "Temporal difference learning." http://www.scholarpedia.org/article/Temporal_difference_learning.

[29] Baijayanta Roy. (2019). "Reinforcement Learning using Temporal Difference (TD) Learning." https://towardsdatascience.com/temporal-difference-learning-47b4a7205ca8.

[30] John Rust. (2006). "Dynamic Programming." extension://bfdogplmndidlpjfhoijckpakkdjkkil/pdf/viewer.html?file=https%3A%2F%2Feditorialexpress.com%2Fjrust%2Fresearch%2Fpapers%2Fdp.pdf

[31] Daniel Johnson. (2022). "Reinforcement Learning: What is, Algorithms, Types & Examples." https://www.guru99.com/reinforcement-learning-tutorial.html

[32] Bertsekas, D. P. (1976). "Dynamic Programming and Stochastic Control. Academic Press, Inc."

[33] Jordi TORRES. (2020). “The Bellman Equation.” https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7.

[34] Martin L. Puterman (1990). Handbooks in Operations Research and Management Science, Pages 331-434

[35] G. Tang, C. Tang, C. Claramunt, X. Hu, & P. Zhou. (2021). “Geometric A-Star Algorithm: An Improved A-Star Algorithm for AGV Path Planning in a Port Environment.” IEEE Access, 9, 59196-59210. doi:10.1109/ACCESS.2021.3070054.

[36] Aylin Tokuç (2023). “Value Iteration vs. Policy Iteration in Reinforcement Learning.”

https://www.baeldung.com/cs/ml-value-iteration-vs-policy-iteration.

[37] Sutton, R and Barto, A. “Chapter 6: Value Iteration. Retrieved.” http://incompleteideas.net/book/ebook/node44.html

[38] Huanwei Wang,Shangjie Lou (2022). “The EBS-A* algorithm: An improved A* algorithm for path planning.” https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0263841

[39] Zhuozhen Tang and Hongzhong Ma (2021). “An overview of path planning algorithms.”

[40] Daniel Johnson. (2022). “Reinforcement Learning: What is, Algorithms, Types & Examples.” https://www.guru99.com/reinforcement-learning-tutorial.html

[41] John Schulman and Filip Wolski. (2017). “Proximal Policy Optimization Algorithms.” https://arxiv.org/abs/1707.06347.