Shortest Path Planning with Reinforcement Learning

Prepared by	: **FEYZA ORAK**
Student No	: **090200340**
Submission Date	: **January 25, 2024**
Course	: **MAT 4901E**
Supervisor	: **PROF. DR. ATABEY KAYGUN**

Abstract

In this study we investigate the use of reinforcement learning methods for the shortest path problem in two-dimensional maze environments. This study contributes to the fields of path planning, maze solving, and reinforcement learning while providing a scalable and adaptable solution for complex cases. To maximize path finding, we used basic algorithms such as Breadth-First Search for validation and Deep Q-Learning for path planning. We achieved a generalizable evaluation by using a random maze generation algorithm to generate a number of test cases to train our agent using Reinforcement Learning. We generated a dictionary of valid states to improve learning efficiency, which allowed the agent to concentrate only on feasible routes while reducing computational complexity. To balance exploration, we used state-of-the-art reinforcement learning approaches such as experience repetition and epsilon-greedy policies. We also used dynamic parameter change methods such as epsilon decay and step limits to further fine-tune the training process and to avoid infinite loops and accelerate convergence. Our experimental results show that the method we used reduced the training time for larger mazes while achieving a higher success rates.

Keywords: *Reinforcement learning, Shortest path problem, Maze solving, Q-Learning, Deep Q-Learning, Breadth-First Search, Epsilon-Greedy policy, Experience replay, Path planning.*

Acknowledgements

I would like to express my deepest gratitude to my esteemed teacher Prof. Dr. Atabey Kaygun for the invaluable guidance and support he gave me throughout my research process.
I would also like to express my gratitude to my mentor Orhan Öztürk for his valuable feedback and constructive criticism.
Finally, I am grateful to my family for their unconditional support, patience, and encouragement during my studies.

Contents

List of Figures

List of Tables

1. Introduction

The problem of finding the shortest path efficiently is an important practical problem in many areas. In uncertain, complex, and dynamic environments, proposed solutions for this problem become even more complex since the search space for optimal solutions grows uncontrollably large. Reinforcement learning plays an important role in reducing the size of the search space in such complex problems. Using reinforcement learning approaches to solve the shortest path problem in a two-dimensional maze setting is the main goal of the work.

In this study, we investigate Q-Learning and its deep learning variation, Deep Q-Learning, in order to find a solution for the shortest path planning problem. We also incorporate the fundamental breadth-first search method as a baseline control for maze path optimization. We use these techniques to train an agent to find an effective path-finding strategy by utilizing state-action-reward modeling.

In our experiments we effectively used reinforcement learning methods to train an agent to find optimized solutions. We also created a valid-states dictionary to streamline the learning process of the agent trained on randomly created mazes to guarantee a wide variety of testing scenarios. We also study uses of techniques such as epsilon-greedy policies and experience replay to improve exploration and learning effectiveness. We assess the effectiveness and success of the suggested strategies through a thorough analysis on how well they scale and adapt to larger and more intricate mazes.

In addition to using reinforcement learning methods for path planning, this study paves the way for future investigations into its practical applications.

2. Methodology

In this study, we used reinforcement learning tools and techniques to target shortest path planning in randomly generated maze environments. We also used search algorithms on trees such as the breath first search algorithm to control the maze paths. In this chapter we will outline all of the theoretical tools and techniques we used in our thesis.

2.1. Breath First Search Algorithm

Breadth-First Search (BFS) is a basic search algorithm that works on trees, or more generally, on graphs. It starts with a specified source node and explores nodes layer by layer, traversing its neighbors before moving to the next node level. BFS is often preferred for systematically analyzing the structure of a graph, for example for finding a shortest path in an unweighted graph.

The BFS algorithm works as follows: first, a source node s is selected, and then a queue Q is started to track the nodes that need to be located. The source nodes are added to the queue and marked as visited. As long as queue Q is not empty during the discovery phase, the procedure proceeds. Node v is taken out of the queue, and each of its unvisited neighbors, u, is added to the queue and marked as visited. Until the queue is empty, this process is repeated. When the queue is empty, meaning that every reachable node has been found, the procedure comes to an end. The pseudo-code for BFS, as presented by Cormen et al.[1], is shown in Algorithm 1.

Algorithm 1: BFS algorithm.

```
Input: Graph $G = (V, E)$, source vertex $s$
Output: Shortest paths from $s$ to all other vertices
for each vertex $u \in G.V$ do
    $u.color \leftarrow$ WHITE;
    $u.d \leftarrow \infty$;
    $u.\pi \leftarrow$ NIL;
$s.color \leftarrow$ GRAY;
$s.d \leftarrow 0$;
$s.\pi \leftarrow$ NIL;
$Q \leftarrow \emptyset$;
ENQUEUE($Q$, $s$);
while $Q \neq \emptyset$ do
    $u \leftarrow$ DEQUEUE($Q$);
    for each $v \in G.Adj[u]$ do
        if $v.color == WHITE$ then
            $v.color \leftarrow$ GRAY;
            $v.d \leftarrow u.d + 1$;
            $v.\pi \leftarrow u$;
            ENQUEUE($Q$, $v$);
    $u.color \leftarrow$ BLACK;
```

2.2. Reinforcement Learning

Reinforcement learning is a field of artificial intelligence formalized as a Markov decision process in which an agent interacts with the environment by performing actions and learns to make decisions by receiving feedback. The agent receives rewards or punishments from the environment as feedback as a result of the action it performs. It then tries to maximize the reward it receives by shaping its decisions based on the rewards it receives [2]. The reinforcement learning loop is shown in Figure 1.

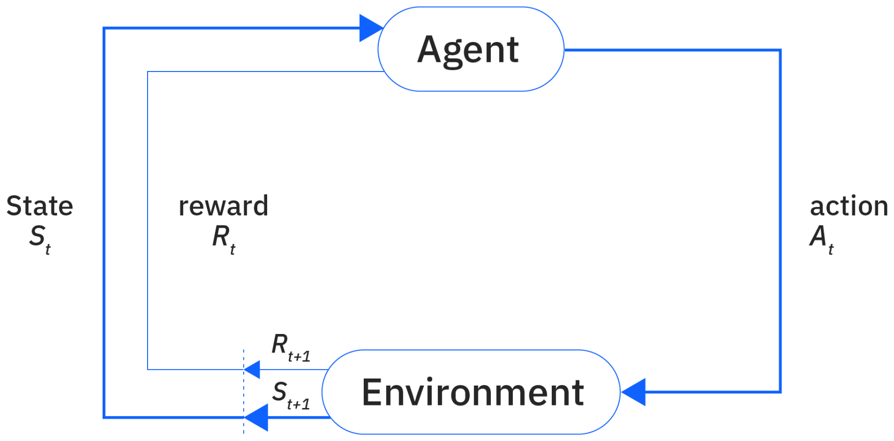


Figure 1: Reinforcement Learning Process.

The agent's goal is to learn a policy $\pi : S \rightarrow A$ that maximizes the reward by telling it which action to choose in which situation. The policy maps states to actions. The mathematical representation of the policy π is given in equation 1.

$$\pi(s) = a \tag{1}$$

The Markov decision process is defined as a tuple (S, A, P, R, γ) that includes the set of states S, the set of actions A, the reward function R, the state transition probability function P, and the discount factor γ, which indicates the importance of future rewards. The parameter $\gamma \in [0, 1]$ is defined as the discount factor, which reduces the future rewards by making the nearest reward more important than the rewards to be collected. The cumulative reward G_t that the agent is expected to collect is defined in equation 2.

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2}$$

The value function that depends on the agent's state is used to estimate future rewards. The value function is shown in equation 3. The value function represents the expected total reward following the π policy.

$$V(s) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right] \tag{3}$$

The total reward estimation of a certain state by taking a certain action according to the π policy is performed by the Q function. The Q function is shown in equation 4.

$$Q(s,a) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right] \tag{4}$$

2.2.1 Q-Learning

Q learning is a widely used reinforcement learning algorithm designed for agents to learn the optimal policy [3]. The main components are the Q value calculated from the Q function, the actions a of the states s, the discount factor γ, and the current reward r received. The agent iteratively updates the Q function with the current rewards it receives and the estimated future rewards obtained from the next states. This iteration is updated using the Bellman equation. Finding the expected value of action a at state s given the estimated future rewards yields the Bellman equation. Equations 5 and 6 illustrate the Bellman equation and the Q learning iteration, respectively.

$$Q^*(s,a) = E\left[r + \gamma \max_{a'} Q^*(s', a')\right] \tag{5}$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right) \tag{6}$$

In these equations, r is the reward received after taking action a in state s. s' represents the new state reached after action a, while $\max_a Q(s,a)$ is the maximum Q value calculated in state s'. The parameter α represents the learning rate and is the factor that determines how fast the updates will learn. Equation 6 shows that the Q function converges to the optimal value, and Equation 7 shows that the optimal policy π is obtained from the optimal Q function.

$$\pi^*(s) = \arg\max_a Q^*(s,a) \tag{7}$$

2.2.2 Deep Q Learning

Deep Q-Learning, which combine deep learning and Q-learning to allow agents to learn policies in high-dimensional state spaces, are a major breakthrough in the field of reinforcement learning. The concept of deep Q-network is based on approximating the Q-value function, which calculates the predicted utility of a specific action in a state [4]. To do this, a neural network is used, which receives the state as input and returns Q-values for every action that might be taken. $Q(s,a;\theta)$ is the representation of the Q-function, where θ is the neural network's parameters. The network weights are updated using the loss function, which is represented in equation 8. The target network weights, which are updated on a regular basis, are represented by θ^- in the equation, whereas θ stands for the current network weights. All things considered, deep Q-network does not require manually created features and enables agents to learn optimal tactics from unprocessed sensory information.

$$L(\theta) = E\left[(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2\right] \tag{8}$$

Throughout training, the target network provides consistent Q-values, lowering the chance of drift. The Q-network's update rule is presented in equation 9.

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta) \tag{9}$$

In equation 9, $\nabla_\theta L(\theta)$ is the gradient of the loss function with respect to its change. The deep Q-network algorithm stores past experiences in a replay memory and samples mini-batches from this memory during training.

2.2.3 Experience Replay

Experience replay, which increases agent training efficiency and saves past events for subsequent use, is a crucial reinforcement learning technique. In order for training to sample from past state-action-reward transitions, experience replay essentially involves recording them in a memory buffer. This strategy keeps learning from becoming unstable by decreasing the relationship between successive experiences. Every transition that occurs when an agent interacts with its surroundings is recorded as a tuple of (s_t, a_t, r_t, s_{t+1}), where s_t denotes the state at time t, a_t represents the action taken, r_t represents the reward received, and s_{t+1} represents the subsequent state [5]. Using algorithms like Q-learning or policy gradients, the agent then uses mini-batches sampled from these transitions to update its policy or value function.

2.2.4 Epsilon-Greedy Policy

An agent can interact with its surroundings in a variety of ways to maximize its learning process in the subject of reinforcement learning. The ϵ-greedy policy is one of the most popular of these tactics. At each step, the ϵ-greedy policy helps an agent balance between exploring its environment and taking the best possible action given the available information, allowing it to choose between exploration and exploitation. Exploitation occurs when the agent choose the optimal course of action, seeks to maximize the reward by using the Q-values it has learnt. Exploration is when the agent deliberately attempts various actions to learn more, selecting random actions based on the information at hand [6]. Finding potentially better long-term strategies requires the agent to investigate less familiar actions, which is made possible by this randomness. A determinant of the balance between exploration and exploitation in the ϵ-greedy policy is the parameter ϵ. Typically, the value of this option falls between 0 and 1. Equation 10 illustrates the policy's operation.

$$a = \begin{cases} \arg\max_a Q(s, a) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{10}$$

With probability ϵ, the agent selects a random action; with probability 1 - ϵ, it selects the action with the highest Q-value. The ϵ-greedy policy eliminates the need to model the environment in order to learn an optimal policy. For an agent that does not fully understand the environment, it is quite helpful. As it gains more knowledge, the agent's exploration gradually diminishes by lowering the value of ϵ. This method lowers the possibility that the agent may become trapped in local optima and works especially well in situations when the best course of action is not immediately apparent. ϵ can decay exponentially, as equation 11 illustrates.

$$\epsilon \leftarrow \epsilon \cdot \epsilon_{decay} \tag{11}$$

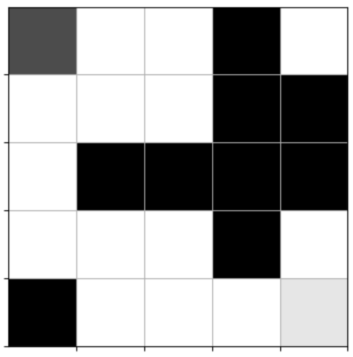
The combination of these strategies enables agents to effectively learn optimal policies in complex environments and ensures sufficient exploration during training.

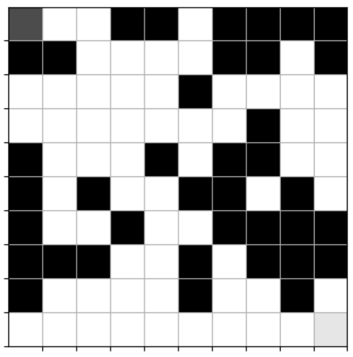
3. Experiment

In our numerical experiments, we considered a two-dimensional maze problem to solve finding the shortest path using reinforcement learning [7]. There are four possible actions in a two-dimensional maze problem: up, down, right, and left. The goal is to reach the target end point using the shortest path from any starting point in the maze. To evaluate the success of the algorithm used, we conducted experiments on different randomly generated mazes where mazes are represented by square matrices. We completed the experiments in three parts: (i) generation of a random maze, (ii) generation of valid states dictionary, and (iii) application of reinforcement learning methods.

3.1. Random Maze Creation

We needed different scenarios to verify the applicability of reinforcement learning techniques in a maze environment. These scenarios vary depending on the complexity and size of the maze. In the experiments, we designed a random maze generation algorithm to examine different scenarios. As per the designed algorithm, we searched for a path using randomly generated paths from the starting point eventually reaching the desired end point. While these sequences of random steps forming our random paths were recorded in a list, if a path traverses a vertex twice, we considered it as a cycle and remove this cycle from the list. Thus, we can find a path that reached the end point without a cycle containing random steps. After the path was found, we completed the maze by randomly assigning walls to the remaining areas with a certain probability. Thus, we made sure that there was at least one path from the beginning to the end and created the maze in accordance with the conditions. In the created matrix, 0 represents the wall and 1 represents the paths. We also took the size of the maze as input and thus allowed it to change. The created maze examples are shown in Figure 2. As a result, we performed each reinforcement learning training on a different maze and it became possible to examine different outcomes at different difficulties.


(a) 5×5 maze


(b) 10×10 maze

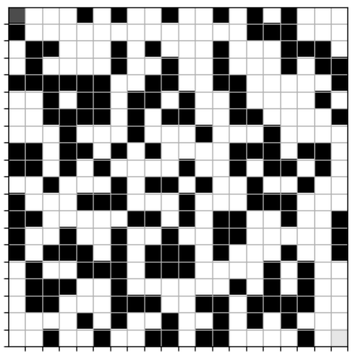

(c) 20×20 maze

Figure 2: Some Examples of Created Mazes.

3.2. Valid-States Dictionary

In order for the agent to reach the desired target point the penalties and rewards must be determined in detail using reinforcement learning techniques. There are some important points that the agent must pay attention to in order to find the shortest path. Of these, not trying to get out of the maze, not hitting the walls, and not re-crossing the path that it has passed. Such moves would need to have high penalty rates. The remaining paths should be scored with a small penalty in order for the agent to be encouraged to choose the shortest path. Finally, the end point should be determined as a reward, reaching it should terminate the training algorithm. Considering all these points, we defined the reward for the end point as 1.0, the penalty for the repeated paths as -0.55, and the penalty for the remaining paths as -0.04. In our training, we did not define any penalties for the walls and paths outside the maze in each step of our training algorithm. The agent's learning not to hit the walls and not to go out of bounds creates a computational load in the training process and negatively affects the time complexity.

In order to side-step this incidental complexity, we found it appropriate to use only the paths inside the maze. In this way, the agent will be able to solve the shortest path problem by avoiding the walls by default. To implement this system, we created a valid state dictionary for each state the agent acquired, which in this case, is given by the position of the agent in the maze. For each position, we examined the feedback from the environment as a result of 4 separate actions, eliminated the results outside the wall and the maze. This effectively transforms the maze into a decision tree, implemented as a python dictionary (also known as a *hash map*), where the children nodes are labeled by valid moves. The dictionary contains only the possible directions (left, right, up, down) the agent can move to forbidding the walls and positions outside of the maze. The agent acquires moves to a different state as a result of one of these actions. Thus, when the agent's state is searched in this dictionary structure, the actions it can take and the states resulting from these actions will be found automatically. As a result, the agent would only make valid moves without being subjected to reinforcement learning training for illegal moves. With this preprocessing, we reduced the computational load of the training by including only the paths of the maze in the training.

3.3. Reinforcement Learning for Maze Solving

After converting the maze into a valid states dictionary, we designed the training process of the agent. We trained the agent to search for a shortest path using a Deep Q Network method [8]. In each epoch, the Q matrix values (which designates a *search-strategy* for the agent) are updated. To check whether the penalty function of the agent is minimized, or equivalently, the reward function is maximized, we determined the paths leading to the end point with the Breath First Search algorithm and examined whether these paths were in the best reward states. If the conditions were suitable, we ended the training as a *win*. Considering the penalty points that the agent could receive, we determined the maximum penalty limit, which is an indicator of the situation where the game is lost, as -0.5 times the maze size. When this limit was reached, we ended the training as *lose*. We also changed *the greedy epsilon value* to increase the training success. By changing *the greedy epsilon value*, we aimed to increase the training success by ensuring the agent's balance between exploration and exploitation. With these measures, we prevented infinite loop situations that may occur due to incorrect learning by allowing the agent to take random actions that are not dependent on learning, and aimed to increase its ability to explore and learn.

In all training cycles, we started the agent with different random states in order to improve the training speed and the success for each path chosen in the maze. In addition, similar to other *simulated annealing algorithms* [9], we gradually decreased the greedy epsilon value in each epoch. In this way, the epsilon value was kept high at the beginning of training in the system by focusing on the exploration strategy, and in the later epochs, the epsilon value was reduced and the exploitation strategy was prioritized. By adjusting the epsilon reduction rate, we ensured that the epsilon value continued to have a small effect in the later epochs. In training, we chose the starting value for epsilon as 0.9, gradually decreasing to 0.1 for the later epochs with the reduction parameter set as 0.99995. The epsilon graph formed in one training is shown in scaled form in Figure 3.

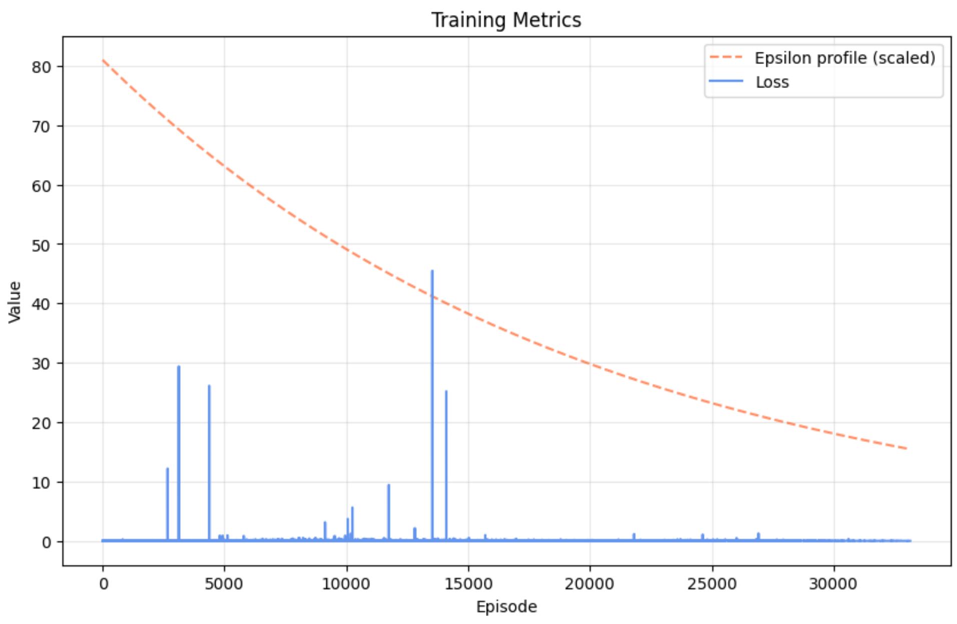


Figure 3: 15×15 Maze Training Epsilon Change Graph.

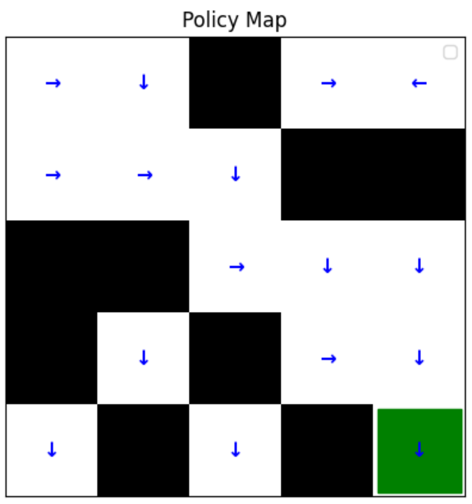
Later, we studied different approaches to shorten the training time. As a result of the studied approaches, we divided the problem into sub-parts by determining a *depth limit* for the number of steps the agent will take in each epoch. Thus, when we evaluated the problem piece by piece, we aimed to achieve success by obtaining fast results and reaching the global optimum from the combination of local optima. Considering the paths that the agent can traverse relative to the size of the maze, we added a certain depth limit to the loops, we run various training rounds with different depth limits. The results of these trainings rounds are shown in Table 1. Trainings with random start points without any depth limits are marked as '*Random Start Condition*', and the training rounds with random start points with a depth limit are marked as '*Fixed Step Size*'.

Maze Size	**Method**	**Epoch**	**Success Rate (%)**	**Time Taken (hr)**
2x2	Random Start Condition	138	100	0.002
	Fixed Step Size	312	100	0.007
3x3	Random Start Condition	554	100	0.031
	Fixed Step Size	541	100	0.032
4x4	Random Start Condition	686	100	0.09
	Fixed Step Size	365	100	0.03
5x5	Random Start Condition	1186	100	0.30
	Fixed Step Size	2814	100	0.91
6x6	Random Start Condition	4038	100	0.61
	Fixed Step Size	2150	100	0.30
7x7	Random Start Condition	5921	100	1.38
	Fixed Step Size	3325	100	0.51
8x8	Random Start Condition	3729	100	2.33
	Fixed Step Size	4006	100	1.23
9x9	Random Start Condition	29999	80	12.30
	Fixed Step Size	4711	100	2.22
10x10	Random Start Condition	29999	88	16.79
	Fixed Step Size	10110	100	2.34
11x11	Random Start Condition	12299	100	9.77
	Fixed Step Size	11625	100	4.83
12x12	Random Start Condition	29999	63	23.52
	Fixed Step Size	15317	100	5.35
13x13	Random Start Condition	29999	80	19.68
	Fixed Step Size	18465	100	6.54
14x14	Random Start Condition	29999	86	23.01
	Fixed Step Size	20384	100	7.85
15x15	Random Start Condition	29999	55	36.95
	Fixed Step Size	29999	67	9.16

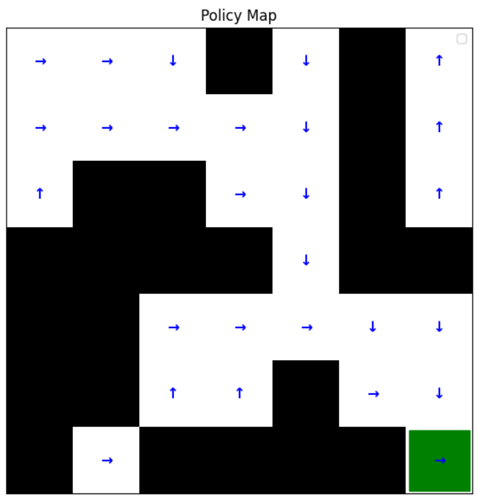
Table 1: Result Comparison Table

4. Results and Analysis

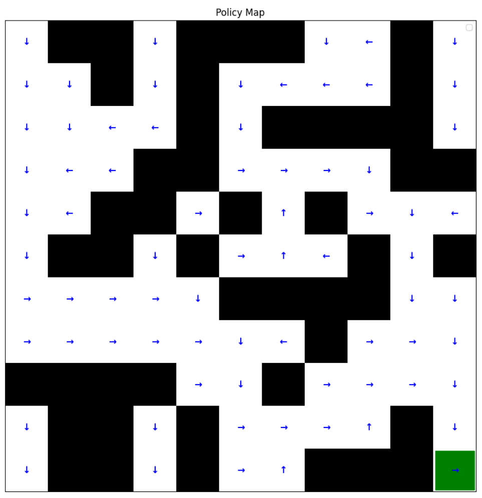
In this thesis, we examined the use of Reinforcement Learning techniques in developing strategies for finding shortest paths in a maze. In implementing the training phase, we also investigated the effects of limiting the depth of the search on the training time and success. In Table 1, we show the results of our experiments using the same random mazes of the same size. In all tests, we kept the epsilon initial value constant as 0.9, the decay value constant as 0.99995 and the max epoch value constant as 29999. We found that setting a depth limit was quite effective in large mazes, but it did not change the training times in small mazes. We think that the main reason for this behavior was the fact that we were able to divide large mazes into smaller pieces by choosing random starting points with depth limits. In the tests we made, we used policy maps, which are the training outputs whose success rates are shown in Table 1, to make the evaluation [10]. See Figure 4.



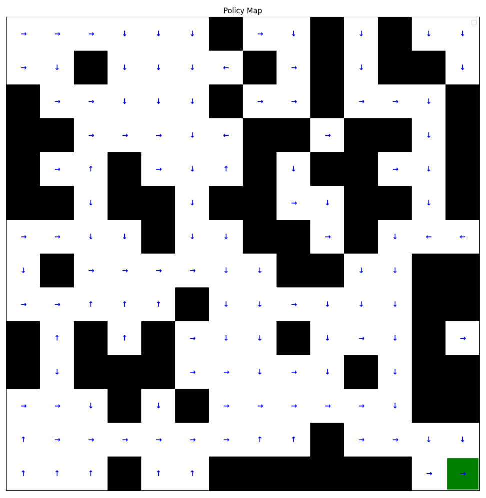
(a) 5×5 maze policy



(b) 7×7 maze policy



(c) 11×11 maze policy



(d) 14×14 maze policy

Figure 4: Result Policy of Some Mazes.

References

[1] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. MIT Press, 2009. URL: https://archive.org/details/introduction-to-algorithms-third-edition-2009/page/595/mode/2up.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd. Cambridge, MA, USA: MIT Press, 2018. ISBN: 0262039249. URL: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

[3] C. J. C. H. Watkins and P. Dayan. *Q-Learning*. In: *Machine Learning* 8.3 (1992), pp. 279–292. URL: https://doi.org/10.1007/BF00992698.

[4] Ashish Kumar Shakya, Gopinatha Pillai, and Sohom Chakrabarty. *Reinforcement learning algorithms: A brief survey*. In: *Expert Systems with Applications* 231 (2023), p. 120495. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2023.120495.

[5] Sergey Levine. *CS 285: Deep Reinforcement Learning*. 2023. URL: https://rail.eecs.berkeley.edu/deeprlcourse/.

[6] David Silver. *Lectures on Reinforcement Learning*. URL: https://www.davidsilver.uk/teaching/. 2015.

[7] Samy Zaf. *Deep Reinforcement Learning for Maze Solving*. 2025. URL: https://www.samyzaf.com/ML/rl/qmaze.html#Deep-Reinforcement-Learning-for-Maze-Solving.

[8] Hooman Ramezani. *rl-maze-pathfinding*. 2025. URL: https://github.com/HoomanRamezani/rl-maze-pathfinding/tree/main.

[9] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. *Optimization by Simulated Annealing*. In: *Science* 220.4598 (1983), pp. 671–680.

[10] Giorgio Nicoletti. *deep_Q_learning_maze*. 2025. URL: https://github.com/giorgionicoletti/deep_Q_learning_maze/tree/master.