LUISS

Degree Program in
Management and Computer Science

Course of Advance Coding for Data Analytics

Comparative Study of A* and Q-Learning in Grid-Based Maze Navigation

Prof. Alessio Martino

SUPERVISOR

Davoud Danish

CANDIDATE

Academic Year 2024/2025

Abstract

In this thesis, I compare traditional A* search and tabular Q-learning in their performance for the task of finding routes in grids. I adopted useful ideas from heuristic and reinforcement learning to design and carry out both algorithms in custom maze environments set up with OpenAI Gym. I analyze performance across different sized mazes, both with standard setup and with the 'Plus' variations which include added portals. Path length, computation time, how much of the state-space is covered and resource use are tracked throughout the test process. Results confirm that A* routinely outperforms Q-learning regarding both efficiency and finding the optimal solution, but the latter's results can be greatly enhanced with the aid of reward shaping. Still, Q-learning is not suitable for running in real-time due to being computationally complicated and less trustworthy. The investigation concludes by looking at Deep Q-Networks (DQN) as a viable way to solve the issues of tabular methods, pointing out that deep reinforcement learning may help in dynamic and high-dimensional applications.

Acknowledgements

I am deeply thankful to Professor Alessio Martino for his patience, careful analysis and willingness to share knowledge, as these qualities shaped this research from the start. I believe I am lucky to have met and worked with him.

I owe great thanks to my friends from all around the world and tutors who stayed by my side and were more than just tutors for me through this academic journey. Having them around, sometimes silently and always reassuringly gave me great moral encouragement. Talking late at night, venting about problems and sharing funny stories helped my colleagues give me unrealized support.

I owe a special thanks to Giulia D'Amati, Carlo Della Rocca and Michele Gradoli. Because they believed in what I did and encouraged me, their warm welcome and sincerity made me able to keep going forward. I appreciate all the good they did for me and the times they sacrificed, no matter their own busy schedules.

I am grateful to my family more than anyone else. I am so thankful to my father, Reza Danish, whose hard work and dedication to its family continues to guide me, to my aunt Mina for placing her faith in me and to my brother Daniyal for his insightful suggestions and assistances. All the important work here is made possible by your love, your giving and your unwavering support.

Table of Contents

Abstract i
Acknowledgements ii
Table of Contents iii
List of Figures v
List of Tables vi
1. Introduction 1
    1.1 Motivation 1
    1.2 Objective 2
2. Background 3
    2.1 Grid Systems 3
    2.2 Pathfinding Algorithms in Game Development 4
        2.2.1 A* Algorithm 5
    2.3 Markov Decision Process (MDP) 6
        2.3.1 The Markov Property 6
        2.3.2 Action Policy 7
        2.3.3 Value Functions 7
    2.4 Reinforcement Learning 8
        2.4.1 Q-Learning 8
3. Methodology 10
    3.1 Environment 10
    3.2 Experimentation Process 12
        3.2.1 Implementation of A* 12
        3.2.2 Implementation of Q-Learning 12
    3.3 Evaluation Metrics 14
        3.3.1 Path Length 14
        3.3.2 Time to Solve 14
        3.3.3 Unique Visited States 14
        3.3.4 Steps per Episode (Q-Learning) 15
        3.3.5 CPU and Memory Usage 15
4. Results 16
    4.1 Random Environments 16

4.2 Random Plus Environments 17
4.3 Random Environments with Reward Shaping 18
5. Discussion and Conclusion 19
5.1 Performance Comparison 19
5.2 Conclusion 20
5.3 An Alternative for Tabular Q-Learning 20
References 22

List of Figures

Figure 1. Interest over time in "Artificial Intelligence" (Google Trends, 2025) 1
Figure 2. Examples of Standard Neighbors for grids (Goldstein, Walmsley, Bibliowicz, & Tessier, 2022) 3
Figure 3. Most Popular Algorithms (Rafiq, Asmawaty, Kadir , & No, 2020) 4
Figure 4. A* Algorithm Demonstration (Pardede, et al., 2022) 5
Figure 5. The Subfields of Machine Learning 8
Figure 6. Symbolic interaction between the agent and environment 10
Figure 7. 10x10 Maze Figure 8. 10x10 Plus Maze 11
Figure 9. Q-Learning Performance on The Sample Maze at figure 7 13

List of Tables

Table 1. A* and Q-Learning Performances on 50 Random Mazes 16
Table 2. A* and Q-Learning Performances on 50 Random Plus Mazes 17
Table 3. A* and Q-Learning with Reward Shaping Performances on 50 Random Mazes..... 18

1. Introduction

1.1 Motivation

Since November 2022, after public release of ChatGPT, Artificial Intelligence has become more popular. It didn't take long for AI tools in Code and Development, Image and Video areas and AI Voice Assistants to become available in many countries (Google Trends, 2025). At some stages, how fast the job market changed led to concerns about being replaced in work. This worry may be due to what abilities AI tools have. In some jobs, creativity is most important, while in others it depends on being accurate and precise. AI can now be developed in artistic areas in which come up with fresh thoughts and learns. Before, machines had different aspects to what was important to us in living creatures. Because of AI, the two were able to be matched.

Figure 1. Interest over time in "Artificial Intelligence" (Google Trends, 2025)

Our view of AI was again brought into question with the growth of autonomous driving. Many companies are now developing delivery and taxi services without human drivers (Amazon Prime Air, Wing, Waymo one, AutoX).

Being a Management and Computer Science student and a gamer, I was interested in exploring how smart algorithms show up in games. After noticing AI's performance in the games I play, I have become even more drawn to this area, because these technologies introduce new, much larger approaches to problem-solving.

Trying to figure out the best way to go from where you are to where you want to be is a basic issue. A good pathfinder in video games is important for making characters that can travel around the map. Commonly, developers have depended on algorithms like A* to quickly find short, effective paths for many years. Meanwhile, Q-Learning is a different approach used for solving these problems. With Q-Learning, the computer will find out the best movement by testing different choices and getting rewards when it chooses wisely.

1.2 Objective

This thesis explores the ways in which reinforcement learning differs from the usual A* path finding algorithm. The results are used to judge how effective using reinforcement learning is in making decisions on pathfinding for games and to see what implementation issues arise.

The source code for this study and detailed graphs are freely available in the following link: https://github.com/itsDevo/thesis

2. Background

2.1 Grid Systems

Grid systems are used in fields such as robotics, video games, architecture and AI-driven navigation to discretize spatial environments. These systems rely on a set of vertices and are usually in square or hexagonal shapes; they allow spatial data to be managed through discrete graph-based or array-based traversal algorithms (Goldstein, Walmsley, Bibliowicz, & Tessier, 2022).
Grids can be considered as simplified spatial representations where each vertex corresponds to a location in space and neighbors are based on the geometric rules (e.g. for square grids either 4 or 8 based on diagonal movement and 6 neighbors for hexagonal or triangular grids).

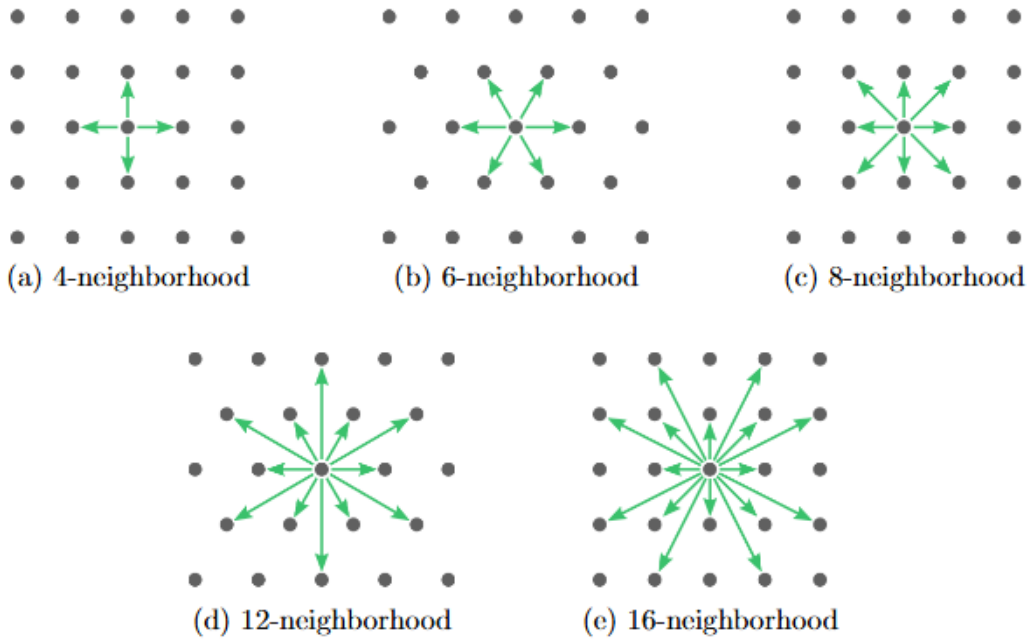

Figure 2. Examples of Standard Neighbors for grids (Goldstein, Walmsley, Bibliowicz, & Tessier, 2022)

In this study, in order to implement A* algorithm, a 4-neighborhood square grid is used.

2.2 Pathfinding Algorithms in Game Development

In this section, I will be focusing on the most used Pathfinding algorithms in video game development. Pathfinding algorithms are used to find the route to the goal for the agent or non-player characters. A game is particularly enjoyable when the characters from the story are as close to real people as possible. Therefore, a good pathfinding algorithm is crucial to ensure realism during gameplay. Nonetheless, determining the most efficient actions for non-player characters continues to be a big problem in video games. (Rafiq, Asmawaty, Kadir , & No, 2020)

According to a study in 2020, A* is the algorithm mostly relied on to discover the shortest path. Alternatively, Dijkstra, Ant Colony Optimization and Genetic Algorithms are applied to discover the lowest cost route through a graph (Rafiq, Asmawaty, Kadir , & No, 2020).

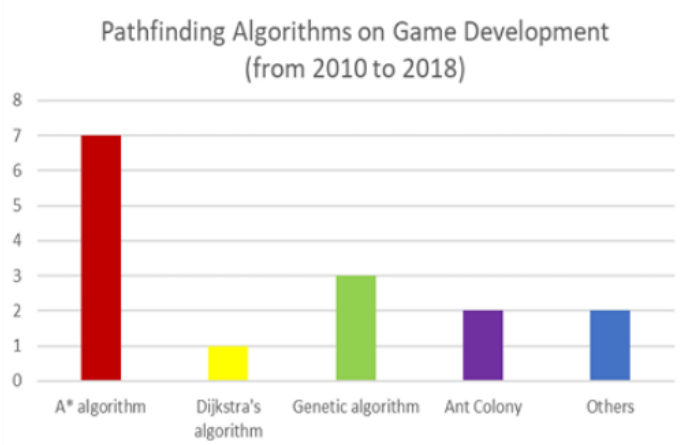

Figure 3. Most Popular Algorithms (Rafiq, Asmawaty, Kadir , & No, 2020)

Figure 2 shows the most popular algorithms which were used in game development based on Abdul Rafiq's study of 10 different game development related papers published between 2007 and 2018. The figure shows that A* was researched in 7 different papers.

2.2.1 A* Algorithm

A* was initially published as an extension of Dijkstra's algorithm. It is a best-first graph search algorithm used to find the shortest path between two nodes for pathfinding problems because of its accuracy and good performance. Heuristic techniques are used instead of classic methods when the performance is not good enough. Heuristic techniques focus on calculating the cost of the cheapest path which is useful for pathfinding problems because it never overestimates the cost (Cui & Shi, 2011)

2.2.1.1 Algorithm Principles

The A* algorithm uses a heuristic function $f(n)$ to determine the node. The function is:

$$f(n) = g(n) + h(n)$$

$g(n)$ is the cost from starting node to the node n (i.e. calculates the cost which was taken from the beginning node by far). $h(n)$ is the heuristic value, it represents the estimated cost from point n to the destination. In case of obstacles $f(n)$ will estimate another route which has the lowest cost to the goal. A* is very similar to Dijkstra's algorithm however it differs by the use of $h(n)$. A* guarantees to find a path from the start to the goal if a path exists. And it is optimal if $h(n)$ is always less than or equal to the actual cheapest path cost from n to the goal. (Pardede, et al., 2022)

There are different heuristics (e.g. Squared heuristic distance, Euclidean distance) for each type of grid system. For example, in 4 and 8 neighborhood systems Manhattan distance is one of the popular used heuristics. It provides a decent estimate of remaining distance without overestimating (Cui & Shi, 2011).

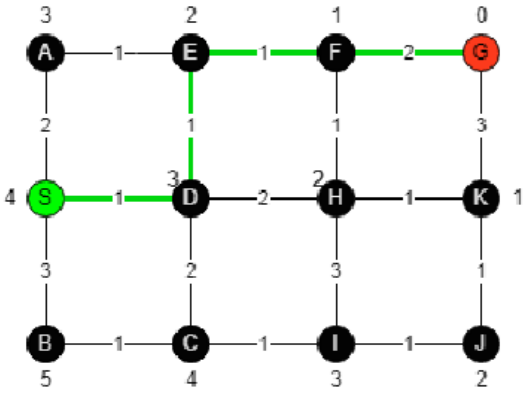

Figure 4. A Algorithm Demonstration (Pardede, et al., 2022)*

2.2.1.2 Strengths and Limitations

A* is a solid algorithm for single-agent grid pathfinding problems. As was mentioned previously, it will always find a solution if one exists and it is suitable for moderate map sizes, mainly thanks to heuristic functions. Besides that, A* is deterministic which allows reproducibility. It will always

produce the same path on the same map, which is desirable in games for predictability and fairness (Pardede, et al., 2022).

However, A* often faces some serious limits. The environment must stay the same and unchanging for the algorithm to work because A* relies on a fixed network and edge costs. Besides that, since it doesn't learn, each time a pathfinding query is made, one solution is provided independently. Computational costs and the use of memory can run into trouble on as grids size increase since A* places every node in memory as open and closed lists and as a result, the algorithm can run slowly or use too much memory. In fact, A* can be improved with hierarchical pathfinding or by pruning the search space, but the original A* can't handle extremely large grids (Cui & Shi, 2011).

2.3 Markov Decision Process (MDP)

Markov Decision Process (MDP) is a formal mathematical framework which forms the basis for numerous reinforcement learning algorithms. It aims to assist with decision processes within an environment where the outcomes are uncertain and only partially influenced by the agent. It serves as a theoretical base, and as a practical base in robotics and game AI (Sigaud & Buffet, 2013).

The structure is defined by a tuple (S, A, T, p, r), a set containing states, actions, time step needed to take a decision, transition probabilities and reward function. An MDP Environment usually consists of a set of states which the agent can be in, a set of actions that agent can take and the reward which agent will receive after realizing the action. Each transition provides a reward which is numerical feedback and the agent's goal is to find the optimal policy which maximizes the cumulative reward over time. MDPs are useful in pathfinding problems, since the agents must navigate from a start state to a goal state while avoiding the obstacles and optimizing the performance metrics such as distance, time or energy consumption.

2.3.1 The Markov Property

Markov property is the assumption behind MDPs which says that future outcomes in the process are affected only by the present state and the action performed, not by all the earlier states (Sutton & Barto, 2018). Simply, the state holds all important past data, so we can focus only on the present for future outcomes. Changes from one state to another always depend only on the present state and the action taken and not on how the agent reached the present state. Because of the Markov property, operations in MDPs and reinforcement learning are very efficient because the algorithms do not depend on past states. Applying this approach reduces the difficulty of analyzing and using learning algorithms in practical systems. When an agent updates the reward that it might get in the future, it will only need the current state which action is taken, the received reward and the next state.

2.3.2 Action Policy

Action policy in an MDP is the way the agent decides what action to take. It is often called π^* and it assigns an action to every state. Action policies can be deterministic or stochastic to, but in both cases the policy gives a clear method for selecting actions from the current state (Sigaud & Buffet, 2013).

The Goal of an MDP is to identify an optimal policy π^* that ensures the agent obtains the highest probable cumulative reward. The cumulative reward is equal to the rewards the agent gets starting from the current step into the future, in which future rewards are slightly discounted. As a result, by following π^* , the agent can obtain the most reward possible over the long run in the MDP. The task of finding an optimal policy is considered the solution to the decision-making problem with uncertainty, it shows the agent how to behave in each situation for the best outcome. A lot of MDP theory is focused on checking the effectiveness of a chosen policy and continuously improving it until the best one is found.

It is good to mention that in some cases, the best expected return can be reached by choosing among several policies. Nonetheless, MDP guarantees that there is always at least one optimal policy under usual conditions (Puterman, 2014). Optimal policies for the problems are computed with policy and value iteration techniques, relying on Markov property and the recursive structure. These techniques iteratively measure the success of a policy (policy evaluation) and adjust it to improve, until the policy reaches π^*.

2.3.3 Value Functions

Value functions are applied to evaluate and compare policies by showing how desirable states or state-actions are in long terms. Value function calculates how much reward or loss expect when it is following a certain policy and it provides the results of being in a specific situation or taking a specific action with numerical values (Sutton & Barto, 2018).

There are 2 main types of value functions in MDPs:

1. State-Value function v_π is the expected return if you start in state s and continue with policy π afterwards and this value is used to sort or rank states under a particular policy in which a higher $v_\pi(s)$ means a state is better for the agent in the long run (Puterman, 2014).
2. Action-Value function Q_π is the expected reward after decision-making from state s, taking action a while the agent uses policy π from that point onward. $Q_\pi(s,a)$ reflects the value of choosing action a in state s and then proceeding according to π. It shows us the value of a state-action pair from that policy. Using action values helps make better decisions. In case of states having the same state-value, the action-value function can help to see which action taken in that state will be more rewarding in the long term.

The main reason value functions matter is that they allow us to assess the quality of a policy indirectly. Once the policy π is known, we can compute $v_\pi(s)$ for all main states which means we learn how effectively it will do the job from each starting point. Policies can be improved by

taking actions with the highest action-value. The relationship between state and action values helps us to find an optimal policy, since it should take the action with the highest value at any state under the optimal policy.

2.4 Reinforcement Learning

This section gives a general overview and theoretical background of Machine Learning and Reinforcement Learning subfield. Machine learning has several subfields. In Supervised Learning the model receives labeled data as input and the model is trained to predict new data. In Unsupervised learning which the models receive unlabeled data and the models are trained to discover the patterns and clusters within the data (Murphy, 2012) and in Reinforcement learning, which is built on the interactions of the agent with the environment, the agent learns the optimal actions with rewards and punishments (Sutton & Barto, 2018).

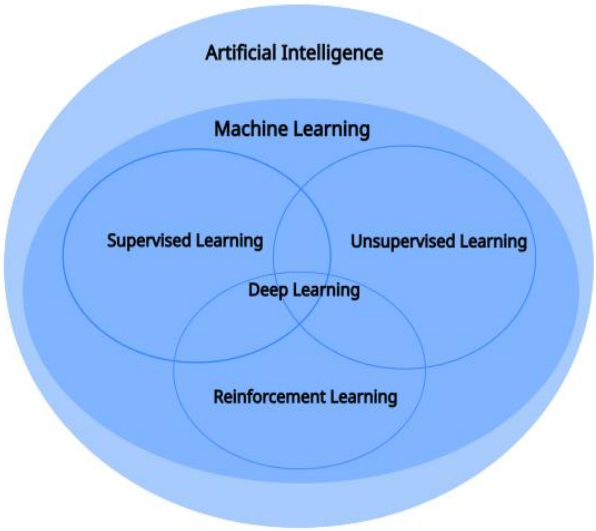

Figure 5. The Subfields of Machine Learning

This study was focused on Q-Learning which is in the Reinforcement Learning subfield.

2.4.1 Q-Learning

Q-learning is a model-free method. This means that the environment's details are irrelevant. It does not require prior knowledge regarding the MDP's transition probabilities or the reward function. It simply adapts and responds through trial and error. (Sigaud & Buffet, 2013)

2.4.1.1 Algorithm Principles

Q-Learning uses a q-table and functions by updating the table in which there is 1 value (q-values) from the action which was taken for the state as a pair. These values estimate the expected total reward for taking a particular action from a given state and then following the optimal policy. The values are updated repeatedly using the following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{\alpha} Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Here, α is the learning rate, γ is the discount factor, and the max term indicates we're pursuing optimal action value, not necessarily the action taken during the episode.

Learning rate determines how quickly the agent updates its knowledge based on new information and the discount factor affects the importance of future rewards.

The algorithm follows these steps:

1. Starts with a blank table
2. At each step
    a. The agent observes the current state s
    b. It selects an action a. It can be exploring (random) or exploiting depending on the exploration rate.
    c. It takes the action and receives a reward r, and then enters the next state s'.
3. The Q-Values are updated by applying the update rule
4. The process repeats on many episodes. And over time the Q-Values converge to the actual expected rewards

2.4.1.2 Strengths and Limitations

Q-learning is an off-policy algorithm. It learns from actions that weren't even executed, assuming they're optimal. That makes convergence simpler to prove and implementations less constrained. Of course, it's not flawless; early learning phases are often noisy, and convergence can be slow.

Large or complex environments bring exploration challenges. The agent might need to explore most of the space to learn optimal actions. Which turns into an issue since the rewards will be delayed and often the agent sticks at loops. So, in such scenarios more advanced strategies might be required. In such scenario, most state-action pairs remain unvisited or poorly estimated and it increases the training time (Sutton & Barto, 2018).

This is where reward shaping enters. Instead of waiting for sparse, delayed rewards, you inject informative signals that guide learning. The original MDP reward function $R(s,a)$ can be supplemented with additional feedback without changing the optimal policy. A trick often done using potential-based shaping. It makes learning faster and more stable, especially in larger or sparser environments (Chintala, Dornberger, & Hanne, 2022).

3. Methodology

3.1 Environment

Open AI Gym is a toolkit for creating and evaluating reinforcement algorithms. It was released by OpenAI in 2016, and it offers a standard interface to a wide range of environments.
Gym is an extensible and modular architecture. Environment class defines a basic interaction loop with reset and step functions. This structure standardizes the way agents interact with their surroundings, understand state transitions, and receive rewards.
Because of this consistency, the users can focus on designing algorithms rather than dealing with creating an environment (Brockman, et al., 2016).

Each environment in Gym is set up as a Markov Decision Process (MDP). As mentioned in the earlier chapters, after observing a state, the agent chooses an action from a continuous or discrete action space, gets rewarded, changes states and finding a policy that optimizes cumulative rewards is the aim.

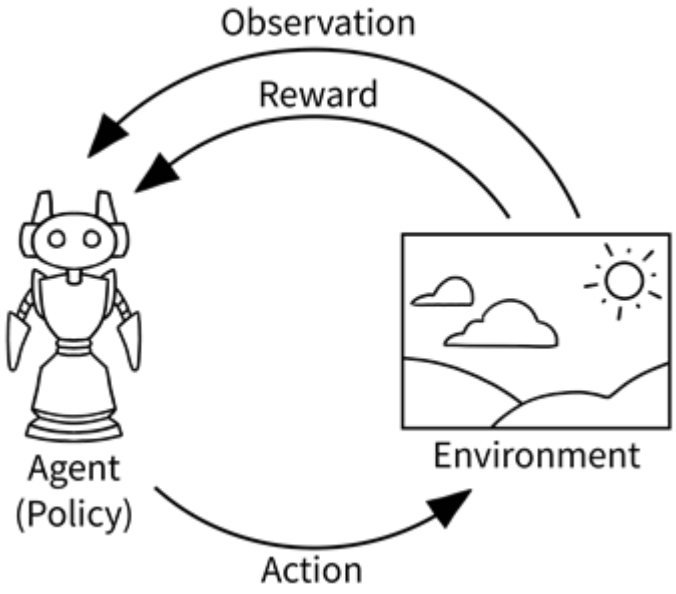

Figure 6. Symbolic interaction between the agent and environment

To evaluate the performance of the Q-Learning Algorithm, a custom 2D maze environment was used. The objective of this environment is showed by a red square, and the agent, visually represented by a blue circle. It must move from the top-left corner (0,0) to the right-bottom corner. The objective is to reach the goal in the fewest steps.

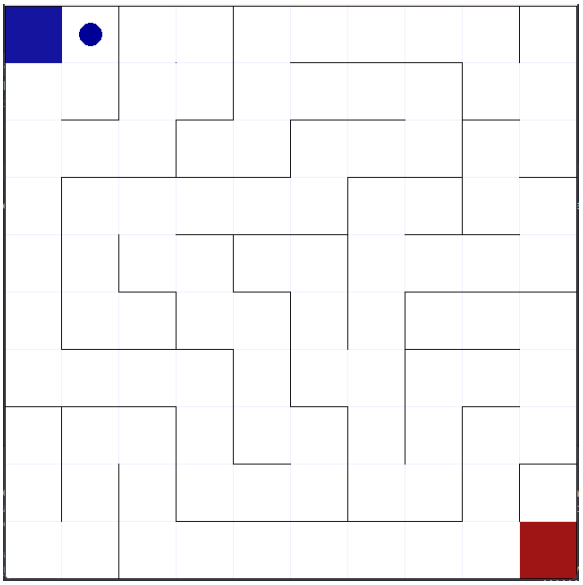

Figure 7. 10x10 Maze

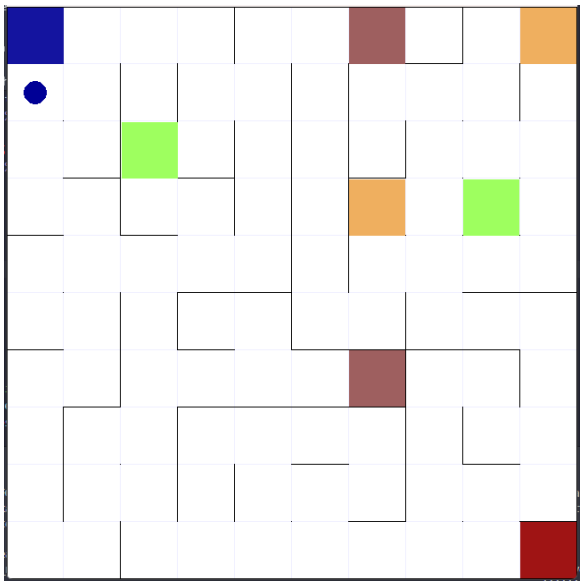

Figure 8. 10x10 Plus Maze

The four cardinal directions, ("N", "S", "E" and "W") designed as North, South, East and West, are the environment's defined discrete action space, and it uses a 4-neighbor grid system (no diagonal movement). If it runs into a wall or a boundary, the agent will stay wherever it is.

In the plus mode of the environment, random portals are located within the map. Each portal has two edges with the same colors, and it teleports the agent to the other edge when it comes to the teleport state.

The reward function uses a small shaped component. The agent gets a reward of +1 after achieving the objective. Every intermediate step is penalized with a small negative reward, which is determined as follows:

$$\text{Reward} = -\frac{0.1}{\#\ of\ Cells}$$

3.2 Experimentation Process

The experiments were run on a laptop device (plugged) with the following specifications: i5-1135G7; 8 GB DDR4 3200 MT/s Single Channel; Windows 11 Pro 64-Bit; Python 3.12.6.

The experimental process used to assess and compare the A* and Q-learning algorithms' performance in a supervised maze-solving environment. Each algorithm was evaluated on 50 randomly generated maze environments to guarantee comparability. The same custom environment that was previously described was used to create these mazes. Both A* and Q-learning were assessed sequentially for every scenario, which means that A* was executed first and then the Q-learning algorithm right after, using the same maze instance. This ensured that the structural obstacles that both algorithms faced (i.e. wall locations, potential routes and dead ends) were the same.

Each scenario's outcome is reported using the evaluation metrics at Section 3.3.3.

3.2.1 Implementation of A*

As it was explained in the previous chapter, A* is a deterministic, graph-based search technique that uses a heuristic estimate of the remaining distance to the goal in addition to the actual path cost. In this study, classic A* algorithm was used and since it complies with the environment's 4-directional grid movement restrictions, the Manhattan distance was chosen as the heuristic function.

The algorithm maintains an open list of states to explore, using a priority queue(min-heap), arranged according to their estimated cost $f(n) = g(n) + h(n)$ where $g(n)$ is the cost to reach state n and $h(n)$ is the Manhattan distance from n to goal.

Based on the wall structure of the maze, the algorithm looks for neighboring cells that are accessible for each state that is explored. Only when a shorter path is discovered, it updates the cost of each neighbor. And lastly, the route is reversed once the goal has been reached.

3.2.2 Implementation of Q-Learning

In this study, a Tabular Q-Learning algorithm with an epsilon-greedy exploration strategy was used. Each maze environment is considered as a Markov Decision Process (MDP), in which the agent chooses an action from a discrete set of options after observing its current coordinate (x, y).

Agent's exploration or exploitation decision is dependent on Epsilon (i.e. exploration rate) in which a higher epsilon has a higher chance of exploration and lower epsilon has a higher chance of exploitation. As the strategy, epsilon decays logarithmically over episodes and besides that to stabilize the convergence, the learning rate decays logarithmically as well. The implementation uses the Bellman update rule to learn the expected return of state-action pairs, keeping the regular off-policy formulation. As for the hyperparameters (e.g. learning rate,

discount rate, epsilon, decay factor, max step for episode, max steps for streak, minimum learning and minimum decay rate), they were set differently for different maze sizes in which larger mazes maintain flexibility and exploration for longer, while in smaller mazes decay more quickly.

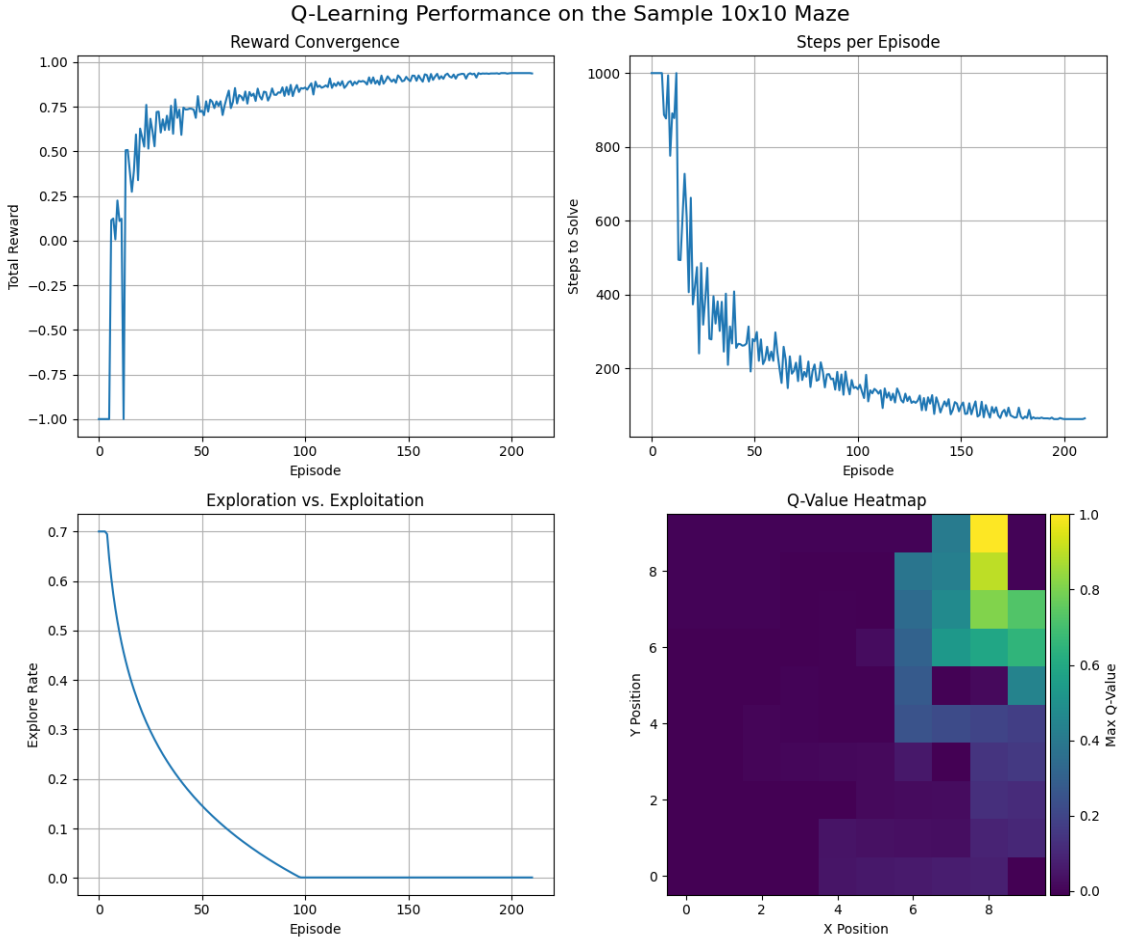

Figure 9. Q-Learning Performance on The Sample Maze at figure 7

Figure 9 shows the results of the implementation of Q-learning algorithm in a sample 10x10 maze environment which was shown in figure 7. As was expected, there is noise in the beginning for Reward convergence and Steps per episode which stabilizes as the agent proceeds. Q-Value Heat Map indicates higher Q-Values on the top-right corner of the map which means the agent has prioritized that corner during its exploration.

For scaling up larger mazes like 30x30 and 50x50, the default reward structure (where the agent received a reward of +1 for reaching the goal and a small constant penalty per step) was insufficient. Therefore, a shaped reward function was implemented to improve inefficiency. There are two main changes.

1. A terminal shaping component was introduced, which applied a negative penalty based on the agent's final distance to the goal if it failed to reach it within the maximum number

of steps. This penalty provides a scaled gradient that encourages the agent to follow shorter routes.
2. The agent receives a small reward boost if it reaches the goal in fewer steps than the predetermined maximum path length.

Training continues until a streak condition is met or a predetermined number of episodes are reached. The streak mechanism tracks if the agent reaches the goal in a feasible number of steps (defined as the maze's field). The agent will be considered as converged if it reaches the predetermined threshold by fulfilling the streak condition continuously. This dynamic stopping criterion makes the training process adaptive and computationally feasible by preventing overtraining in smaller mazes and providing efficiency in larger ones.

3.3 Evaluation Metrics

A set of evaluation metrics was set up to evaluate both algorithms' behavior and performance in a systematic and controlled way. These metrics are designed to measure a few features of the algorithms' performance, such as computational costs, execution time, search complexity and solution efficiency. The following metrics were collected for every experiment.

3.3.1 Path Length

The number of steps is taken along the chosen path from the starting point to the goal. This metric shows how effective the algorithm's chosen direction was. It is equivalent to the calculated optimal or near-optimal path for deterministic solvers such as A* and it represents the quality of the learned policy after training in learning-based agents such as Q-Learning

3.3.2 Time to Solve

The time needed to finish the maze for each episode. This metric was considered instead of execution time of the entire algorithm since A* would always outperform Q-Learning because of its training requirement. It measures the quickness of decision-making and general method responsiveness.

3.3.3 Unique Visited States

The total number of unique maze states (cell) visited in one run. This metric shows the search method of the algorithm and indicates whether it's exploratory and exhaustive or direct and efficient. While a lower value might suggest more concentrated movement, a higher value could suggest a wider search. It is good to mention that since the experiment is run on 50 random mazes, the highness or lowness of the value can be related to the complexity of the maze as well.

3.3.4 Steps per Episode (Q-Learning)

It tracks how many steps the agent takes in each learning episode until termination. Specifically, Q-Learning, it helps in tracking the stability and convergence rate of the policy. With time, fewer steps often indicate better learning results.

3.3.5 CPU and Memory Usage

System resource usage is tracked during execution in terms of CPU usage (%) and memory allocation (MB). Although CPU percentage is hardware-dependent and may differ between devices, it still offers a relative indication of computational need.

4. Results

This chapter covers the results of tests that were done to see how well the A* algorithm and the Q-learning algorithm could find paths in maze environments. At first, the A* algorithm was compared to the traditional Q-learning method (without reward shaping) and after that, the comparison was made bigger by adding the modified Q-learning with reward shaping.

4.1 Random Environments

Maze Size	Algorithm	Path Length	Solve Time (s)	Unique States Visited	CPU Usage (%)
3x3	A*	4.12 ± 0.47	0.0004 ± 0.0002	7.58 ± 1.25	0.00 ± 0
	Q-Learning	4.12 ± 0.47	0.0018 ± 0.0007	4.31 ± 0.84	30.57 ± 19.27
5x5	A*	10.00 ± 2.08	0.0006 ± 0.0003	17.30 ± 4.31	1.95 ± 13.68
	Q-Learning	10.04 ± 2.06	0.0043 ± 0.0011	11.00 ± 1.99	21.84 ± 10.82
10x10	A*	33.52 ± 9.38	0.0024 ± 0.0011	63.42 ± 17.71	7.81 ± 32.92
	Q-Learning	49.28 ± 11.35	0.0209 ± 0.0043	38.06 ± 8.87	19.65 ± 5.44
20x20	A*	99.64 ± 25.83	0.0036 ± 0.0014	241.58 ± 67.64	5.86 ± 23.20
	Q-Learning	299.16 ± 84.09	0.0249 ± 0.0080	121.57 ± 19.07	30.77 ± 5.78
30x30	A*	221.76 ± 74.52	0.0090 ± 0.0024	603.90 ± 136.28	20.32 ± 40.66
	Q-Learning	598.38 ± 201.52	0.0652 ± 0.0125	269.82 ± 64.68	45.79 ± 4.72
50x50	A*	537.56 ± 149.55	0.0262 ± 0.0083	1695.28 ± 375.11	36.13 ± 36.39
	Q-Learning	4262.98 ± 8383.71	0.1545 ± 0.0448	969.75 ± 3.20	54.83 ± 3.20

Table 1. A and Q-Learning Performances on 50 Random Mazes*

4.2 Random Plus Environments

Maze Size	Algorithm	Path Length	Solve Time (s)	Unique States Visited	CPU Usage (%)
10x10 Plus	A*	19.36 ± 2.35	0.0013 ± 0.0005	55.90 ± 14.56	8.21 ± 27.84
	Q-Learning	22.66 ± 9.14	0.0062 ± 0.0015	24.99 ± 5.89	13.29 ± 6.14
20x20 Plus	A*	41.96 ± 4.01	0.0048 ± 0.0023	192.84 ± 84.84	15.89 ± 36.43
	Q-Learning	88.78 ± 37.96	0.0245 ± 0.0052	78.98 ± 16.05	27.22 ± 4.72
30x30 Plus	A*	63.92 ± 3.73	0.0135 ± 0.0068	485.10 ± 177.61	22.45 ± 45.71
	Q-Learning	187.02 ± 68.54	0.0468 ± 0.0102	132.46 ± 25.68	36.25 ± 3.88

Table 2. A and Q-Learning Performances on 50 Random Plus Mazes*

4.3 Random Environments with Reward Shaping

Maze Size	Algorithm	Path Length	Solve Time (s)	Unique States Visited	CPU Usage (%)
3x3	A*	4.32 ± 0.84	0.0001 ± 0	7.42 ± 1.23	0.00 ± 0
	Q-Learning	4.34 ± 0.89	0.0002 ± 0.0001	4.58 ± 0.81	4.31 ± 19.27
5x5	A*	10.08 ± 2.59	0.0002 ± 0.0001	17.18 ± 3.85	0.00 ± 0
	Q-Learning	10.08 ± 2.59	0.0007 ± 0.0002	10.95 ± 2.51	11.70 ± 16.37
10x10	A*	33.68 ± 9.49	0.0009 ± 0.0004	65.36 ± 20.99	2.08 ± 14.59
	Q-Learning	35.36 ± 11.30	0.0047 ± 0.0012	41.39 ± 11.09	23.36 ± 8.00
20x20	A*	106.52 ± 34.98	0.0034 ± 0.0010	256.00 ± 69.76	1.95 ± 13.68
	Q-Learning	121.24 ± 46.06	0.0290 ± 0.0049	135.25 ± 36.59	57.22 ± 4.64
30x30	A*	230.08 ± 83.36	0.0090 ± 0.0031	617.84 ± 177.23	36.89 ± 47.64
	Q-Learning	306.54 ± 150.85	0.0628 ± 0.0134	319.57 ± 94.27	70.50 ± 4.63
50x50	A*	490.28 ± 118.94	0.0219 ± 0.0068	1573.98 ± 449.30	59.71 ± 44.28
	Q-Learning	965.68 ± 573.61	0.1693 ± 0.0333	773.32 ± 149.66	83.51 ± 2.34

Table 3. A and Q-Learning with Reward Shaping Performances on 50 Random Mazes*

5. Discussion and Conclusion

5.1 Performance Comparison

The tests show that A* and Q-learning had similar average path lengths in small mazes (e.g. 3x3 and 5x5) which show that they were equally good at finding the optimal path at small sizes. But there was a big difference in computational resources and time to solve. A* was much faster than Q-learning. Also, Q-learning had a higher CPU usage, which shows that its learning process is very CPU-intensive, even at this smaller scale. But as Memory usage there was not any noticeable difference during the experiments except in extreme sizes.

As the maze became harder the differences between the two algorithms became clearer. A* solved the problem in less time. As the size increases, inefficiencies in Q-learning were more noticeable in path lengths and times to solve. On the other hand, A* maintained faster solution times and shorter path lengths which were sometimes 3 or 4 times shorter than those found by Q-Learning.

A* had an optimal and efficient performance in Plus environments as well. This shows clearly that A* has better efficiency and accuracy in situations with more complex navigational parts.

In the third phase of the experiments, reward shaping was introduced and there was an improvement in Q-Learning's performance. Especially in terms of efficiency and computational resources. Despite the average path length not being reduced for small maze sizes, there is a reduction in average solving time. Even so, A* continued to show superior efficiency, especially in terms of resource stability and time to solve speed. As the size increases the effects of reward shaping are more visible. Although Q-learning showed lower path lengths and quicker results, still it couldn't maintain its results close to A*.

The results validate the improvement after reward shaping. However, A* continuously outperformed Q-learning even with the adjustments, showing its applicability for applications with limited resources and time. Besides that, A* continuously showed lower CPU usage for all maze sizes. So, we can conclude that Q-Learning has limitations for real-time or resource-constrained applications unless significant optimizations are done.

As extra, A* was tested for larger environment sizes like 100x100, 200x200 and 500x500 mazes. Unfortunately, it was not possible to run the tests for Q-Learning due to lack of computational resources.

Interested readers are recommended to look at the appendix for detailed plots of these larger benchmarks.

5.2 Conclusion

As conclusion, the results showed us the potential of AI. It is good to mention that Q-Learning is only one of the techniques in the AI world and these technologies are still in development. So, we can say that AI performs better in terms of quickness and efficiency than humans, however it doesn't perform as good as traditional algorithms for more complex cases now, but maybe it will in the close future.

5.3 An Alternative for Tabular Q-Learning

Deep Q-Networks (DQN) is the combination of classical Q-learning and a deep neural network (Vijay Ram Reddy et al., 2024). Instead of using a table for all $Q(s, a)$ values, a multi-layer neural network (typically a convolutional network) is used to estimate Q values. With this approach, DQN can handle high-dimensional inputs that were not feasible for tabular methods. This means that for the first time, DQN showed that learning policies from high visual information is possible directly with Q-learning. The authors' main work showed that their DQN agent could outperform previous approaches and even surpass human experts on several of the classic video games (Mnih, et al., 2013).

DQN adds some important changes to the Q-learning algorithm to train the network. Using sequential, correlated game frames with a nonlinear function approximator can cause Q-learning to perform unstable or divergent (François-Lavet, Henderson, Islam, Bellemare, & Pineau, 2018). DQN uses 2 techniques to solve this issue:

1. Experience replay, during training, the agent uses experience replay to randomly choose mini batches that contain elements (s, a, r, s') to overcome the issue of consecutive samples.
2. Target network is the secondary network which is used to compute stable Q-Values and its network weights are updated less often than the primary network's. It reduces the feedback loops which helps to prevent divergence.

These techniques help DQN to perform better in more complex and high dimensional environments in which tabular Q-Learning was insufficient.

DQN provides beneficial empirical results over traditional tabular Q-learning whenever situations are complicated. It is mainly scalable, since tabular Q-learning handles only smaller or discretized state spaces by storing and updating every entry vector (Lyl, Dazeley, Vamplew, Cruz, & Aryal, 2022). Dealing with this in situations with a huge number of states (for instance, an Atari screen with 160*210 pixels) is too hard to implement. As DQN generalizes across different states, the neural network reduces the need to memorize all states, helping it to solve the problems without storing every state.

As a result, DQN's network can handle large continuous spaces by providing useful estimates of state-action values, surpassing what a tabular method could achieve. In practice, DQNs have achieved results with input spaces that are much too large for any tabular agent to analyze.

Furthermore, using function approximation in DQN allows it to generalize better than a lookup-table agent which only works well for exact states it has trained on (Myhre, 2019).

Researchers found that a tabular Q-learning agent struggled when the environment was changed, while a DQN agent adapted easily to such small changes. Since DQN can be used with modern computations (GPU acceleration with mini-batch updates) instead of the simple table method, it can complete many learning tasks more quickly at large scale. Overall, the use of deep neural networks in Q-learning gives agents the ability to address much greater challenges than was the case with simple tabular methods (Lyl, Dazeley, Vamplew, Cruz, & Aryal, 2022).

References

Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym.

Chintala, P., Dornberger, R., & Hanne, T. (2022). *Robotic Path Planning by Q Learning and a Performance Comparison with Classical Path Finding Algorithms.*

Cui, X., & Shi, H. (2011). A*-based Pathfinding in Modern Computer Games.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., & Pineau, J. (2018). An Introduction to Deep Reinforcement Learning.

Goldstein, R., Walmsley, K., Bibliowicz, J., & Tessier, A. (2022). Path Counting for Grid-Based Navigation.

Google Trends. (2025). *Artificial Intelligence.*

Lyl, A., Dazeley, R., Vamplew, P., Cruz, F., & Aryal, S. (2022). Elastic Step DQN: A novel multi-step algorithm to alleviate overestimation in Deep QNetworks.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., & Riedmiller, D. W. (2013). Playing Atari with Deep Reinforcement Learning.

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective.*

Myhre, I. (2019). A Comparative Study of Reinforcement Learning.

Pardede, S. L., Athallah, F. R., Zain, F. D., Huda, Y. N., Nugrahaeni, R. A., Kallista, M., & Kusuma, P. D. (2022). A Review of Pathfinding in Game Development.

Puterman, M. L. (2014). *Markov Decision Processes: Discrete Stochastic Dynamic Programming.*

Rafiq, A., Asmawaty, T., Kadir , A., & No, S. (2020). Pathfinding Algorithms in Game Development.

Sigaud, O., & Buffet, O. (2013). *Markov Decision Process in Artificial Intelligence.*