Multi-Goal Pathfinding with Deep Q-Learning

Jazib Ahmad
Department of Electrical, Computer, and Biomedical Engineering
Toronto Metropolitan University
Toronto, Canada
jazib.ahmad@torontomu.ca

Riley Keays
Department of Electrical, Computer, and Biomedical Engineering
Toronto Metropolitan University
Toronto, Canada
rkeays@torontomu.ca

Aiyang Liang
Department of Electrical, Computer, and Biomedical Engineering
Toronto Metropolitan University
Toronto, Canada
yang.liang@torontomu.ca

Linas Gabrys
Department of Electrical, Computer, and Biomedical Engineering
Toronto Metropolitan University
Toronto, Canada
linas.gabrys@torontomu.ca

Cungang Yang
Department of Electrical, Computer, and Biomedical Engineering
Toronto Metropolitan University
Toronto, Canada
cungang@torontomu.ca

***Abstract*— Pathfinding, the process of finding a traversable route from one point to another, is an important field of research due to the implications it can have for many other fields and technologies. The purpose of this paper is to propose a new Q-Learning based pathfinding algorithm to solve mazes in which the algorithm ("agent") must find multiple subgoals before reaching a final destination, in fewer iterations than existing Q-Learning algorithms. The existence of multiple subgoals can result in 'reward loops' which cause an unnecessary increase in the number of iterations to solve the maze [1]. These reward loops can be eliminated with the use of Multiple Q-Tables [1]. However, Q-Table based methods require many iterations for large state environments (such as large mazes) to learn the Q-values for each state-action pair [2], [3]. Deep Q-Learning can be used to approximate Q-values for previously unseen state-action pairs and thus reduce the number of iterations to learn the optimal solution [3].**

Therefore, we propose an algorithm that adapts Multiple Q-Table theory to Deep Q-Learning to address the two weaknesses of basic Q-Learning at once: 1) The poor scalability to large mazes, and 2) The existence of reward loops. The proposed design is the use of Multiple Deep Q-Networks, each of which is responsible for finding the shortest path to the nearest subgoal or final destination (from the previous subgoal or starting point). We hypothesize that in addition to the elimination of reward loops, this results in other advantages including the balancing of exploration and of the replay memory allocation. In addition, we optimize our design with an improved Exploration Strategy, the addition of a Revisiting Penalty, as well as hyperparameter optimization. We test our solution on sample mazes of four sizes and compare it to the Multiple Q-Table and Single Deep Q-Network algorithms. Our results confirm our hypothesis and show that our solution outperforms the other algorithms in the number of iterations to find the shortest path, especially on larger mazes. Finally, we offer suggestions for alternative designs, future work, and improvements. (*Abstract*)

***Keywords*—Deep Q-Learning, Multiple Goal Pathfinding, Multiple Q-Tables, Exploration Strategy, Neural Networks, Reinforcement Learning** (*key words*)

I. Introduction

Multi-goal pathfinding problems, where an agent must reach several intermediate targets before arriving at a final destination, are common in robotics, logistics, and video game AI. This paper proposes a new algorithm for multi-goal pathfinding, with better scalability for larger mazes and elimination of inefficient behavior such as reward loops.

The proposed method utilizes reinforcement learning, specifically Q-learning, to solve mazes with multiple subgoals. Reinforcement learning is an approach to machine learning which represents the agent's environment using a Markov Decision Process [4]. This enables the agent to learn from the actions it takes in its environment so that it may determine the optimal actions to perform and when to perform them to yield the highest reward [4]. When performing reinforcement learning the agent will also make use of an exploration strategy which determines when the agent chooses to explore for new information or exploit the information it already knows [4].

Q-learning is one such method of performing reinforcement learning. It uses Q-values, derived from the Bellman equation, to represent the expected present and future reward for choosing a certain action in a given state [4]. These Q-values are then stored in a Q-table so that the agent can determine the best action in a given state by selecting the action with the highest Q-value [4]. When applied to maze solving, Q-learning is generally used to find the shortest path through the maze. For this paper it was used to find the shortest path to collect all of the subgoals before reaching the end of the maze.

Reference [3], builds upon the basic Q-learning process described above by replacing the Q-table with a neural network, thus enabling the prediction of Q-values after sufficient training and removing the need for the agent to determine every Q-value through direct experience. This improved process is called Deep Q-learning [3]. Reference [5], further modifies the Deep Q-learning process by splitting the single neural network into two, a Policy Network and a Target Network for improved stability.

Another paper which improves upon the basic Q-learning process is [1]. It uses multiple Q-tables and a modified Q-learning algorithm to solve a maze with multiple subgoals in fewer iterations by eliminating the presence of reward loops. This modified Q-learning process is called Multi Q-table Q-learning [1].

Our proposed Q-learning method adapts and combines the Multi Q-table Q-learning algorithm to be used with Deep Q-learning. It uses multiple Deep Q-networks instead of multiple Q-tables following a similar algorithm for managing the networks as used in [1]. Each neural network finds the path to the closest subgoal then switches to the next neural network after collecting it. This process then repeats until the end goal is reached. Our proposed Q-learning method is described in more detail in section III.

The paper is organized as described here. Section II covers some of the background theory our proposed method is based on in more detail, including the topics of reinforcement learning, Q-learning, Deep Q-learning, and

Multi Q-table Q-learning. Section III covers the details of our proposed Q-learning method as well as several other different versions created to compare its performance. Section IV details our experimental setup while section V details and discusses the results of our experimentation. Finally section VI concludes the paper.

II. Background Theory

A. Reinforcement Learning

Machine learning is a subset of Artificial Intelligence which seeks to enable computers to successfully perform tasks of varying complexity without explicit instructions or programming to do so [4]. Machine learning techniques make use of different algorithms and/or mathematical models to recognize patterns and information in the training data which are then used to perform the required task [4]. There are 3 main branches of machine learning which are referred to as supervised learning, unsupervised learning, and reinforcement learning respectively [4].

Within the branch of reinforcement learning there are both model based and model free algorithms [4]. Both methods seek to maximize the agent's cumulative reward thus allowing it to learn what action to take or not to take in a given state based on their reward [4]. Our research used Q-Learning, a model free off policy reinforcement learning algorithm [4] discussed further in the next section. Reinforcement learning typically uses Markov Decision Processes (MDPs) to represent the environment that the agent will learn [4]. An MDP consists of an agent, who takes actions within the environment and receives rewards based on those actions, and the environment itself which consists of a number of states and the actions that can be performed in each of those states [4]. Each time an agent takes an action in a given state it will receive the corresponding reward and then transition the environment to the next state [4]. This process repeats as the agent explores, which allows the agent to learn about its environment. Fig. 1 shown below shows a block diagram of a typical MDP [4].

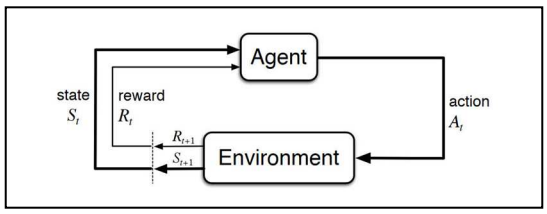

Fig. 1. A typical Markov Decision Process [4].

While an agent is training on its environment it must have some policy for deciding which actions to take. An agent can choose an action with the highest predicted reward, called exploitation, or it can choose another less optimal action randomly, called exploration [4]. To effectively learn about its environment an agent must have sufficient exploration to be able to find and learn about all the potential rewards in the environment while also possessing sufficient exploitation to take advantage of the rewards it knows about and develop an optimal solution to maximize its cumulative reward [4]. How the agent balances exploration and exploitation is determined by the exploration strategy that the agent uses.

B. Q-Learning

As mentioned in the previous section, Q-Learning is a model free off policy reinforcement learning method [4]. Q-Learning makes use of two fundamental concepts. The first is the Q-Value. A Q-Value is a numerical value which belongs to a state-action pair and indicates the quality of taking a particular action in the current state and then continuing to select the action with the highest Q-Value in all subsequent states [4]. By always choosing the action with the highest Q-Value in a given state the agent can follow the determined policy and maximize its cumulative reward. To determine the Q-Value for a state-action pair Q-Learning makes use of the second fundamental concept, the Bellman equation. The Bellman equation, shown below in (1), updates a Q-Value, $q(s,a)$, based on the reward, R_{t+1}, for taking action a in state s and the maximum Q-Value available in the next state, $q(s',a')$ multiplied by a discount factor γ [4].

$$q(s,a) = E[R_{t+1} + \gamma maxq(s',a')] \quad (1)[4]$$

While the Bellman Equation will not provide the optimal policy immediately, it is guaranteed to eventually converge to the optimal policy once the agent has recursively determined the expected rewards for each state-action pair [4].

To perform Q-Learning the agent selects an action for a given state and then updates the Q-Value based on the reward the agent received for performing the action and the new state it transitioned to as shown in (2) below.

$$q_{new}(s,a) = (1-\alpha)q(s,a) + \alpha\big(R_{t+1} + \gamma maxq(s',a')\big) \quad (2)[4]$$

The value α is the learning rate which allows us to control how quickly the agent learns from new experiences and forgets old ones [4]. This process is repeated until the Q-Values have converged to the optimal policy. Traditionally the Q-Values are stored in a table, called a Q-Table, which contains one row for every state and one column for every action [4]. Thus an environment with N states and M action in each state would need a NxM sized Q-Table.

C. Deep Q-Learning

While typical Q-Learning using a Q-table can be quite effective, it has some limitations. The biggest is that as shown in (2) the agent needs to know all the Q-Values in the next state to accurately calculate the Q-Value of the current state-action pair. This means the agent must iterate over each state-action pair multiple times until the optimal policy can be determined. In the worst case this can cause the agent to have to iterate over every state-action pair in the environment multiple times which, depending on the state and action space of the environment, can be a time consuming and inefficient process [2], [3].

Deep Q-Learning seeks to address this issue by utilizing artificial neural networks to make predictions about the quality of a state-action pair including state-action pairs which the agent has not seen before [3]. This allows the agent to learn the optimal policy without having to iterate over every state-action pair in the environment since the

agent is able to make predictions about unseen state-action pairs based on the state action-pairs it has seen [3]. This significantly increases the efficiency of the Q-Learning process allowing the agent to find the optimal policy in fewer iterations [3]. To perform Deep Q-Learning a neural network is used in place of the Q-Table to approximate the Q function based on the agent's experiences in the maze [3], [6]. To store these experiences Deep Q-Learning makes use of a data structure called Replay Memory which is a queue of the last N experiences the agent has had, where N is a finite integer [6]. Then each time the agent takes an action and receives a reward the experience is saved to the Replay Memory and the neural network is trained [6]. The neural network receives the entire current environment state as an input and outputs the predicted Q-Values for each possible action [6]. During training, the neural network's weights are updated using equation (1), which computes target Q-values for each experience in the batch. These targets are compared to the network's predicted Q-values, and the weights are adjusted by minimizing the loss between them [6].

Because (1) requires the maximum Q-Value in the next state, which must be predicted from the neural network, the target and the predicted values will both be changing as the neural network learns. This can lead to the predicted values "chasing" the target values causing instability in the neural network [5]. To counteract this, a variant of the basic Deep Q-Learning process was developed which splits the neural network into two different neural networks. One network, called the Policy Network, is used to predict the Q-Values and train while the other, called the Target Network, is used to predict the max Q-Value of the next state that is needed for calculating the target values [5]. The Target Network is identical to the Policy Network and has its weights updated periodically by copying them from the Policy Network [5]. This allows the targets used by the Policy Network to train to be stable for a certain period of time thus improving the stability of the Policy Network and the Deep Q-Learning process [5].

D. Multi Q-Table Q-Learning

Basic Q-Learning also has other limitations beyond those already discussed in the previous sub-section. While basic Q-Learning can solve single goal mazes reasonably well it can take a large number of iterations to solve mazes with multiple temporary subgoals and one final goal [1]. The main cause of this is due to reward loops which occur when an agent tries to exploit a reward which is no longer there [1]. This can result in the agent repeatedly attempting to exploit a previously collected reward which is no longer present until it accumulates a large enough penalty to discourage it, resulting in a much longer training time [1].

To break these reward loops and prevent them from increasing the training time an alternate method of performing Q-Learning was proposed in [1]. This method uses multiple Q-Tables instead of a single one to solve the maze [1]. Every time the agent encounters a subgoal it checks if it has previously found this subgoal before [1]. If it hasn't it creates a new blank (all cells initialized to zero) Q-Table and then switches to the new Q-Table [1]. If it has found the subgoal before it switches to the matching Q-Table [1]. Each time the agent takes an action the current Q-Table is updated until that Q-Table has determined the shortest path to the nearest subgoal or the final goal if no other subgoals remain or are close enough. The algorithm for Multi Q-Table Q-Learning is shown below in Fig. 2[1].

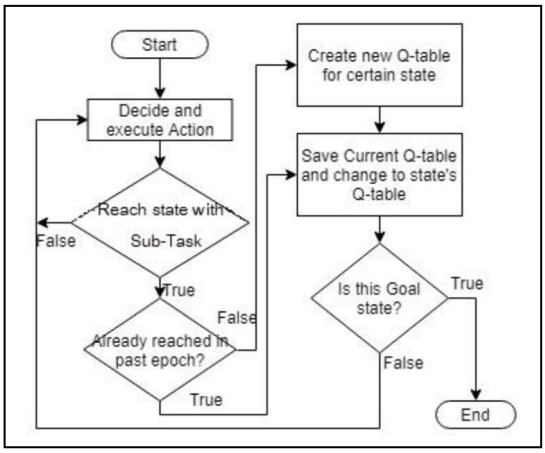

Fig. 2. Multi Q-Table Q-Learning algorithm [1]. Reprinted with permission from [1], ©2019 IEEE.

Practically speaking this results in each Q-Table finding the optimal path from one reward to the next, a much less complex task, until it reaches the end [1]. Switching Q-Tables also prevents a reward loop from occurring as each Q-Table has no knowledge of the reward it was created from and thus cannot attempt to exploit it [1]. Both of these properties combine to greatly improve the performance of the Multi Q-Table Q-Learning algorithm on multi-goal mazes when compared to the basic Q-Learning algorithm [1].

III. Our Proposal

Our proposal applies the Multi Q-Table Q-Learning theory to Deep Q-Learning. The goal is to improve the agent's ability to solve multi-goal mazes in fewer iterations, particularly in larger environments where Q-table methods become impractical. While the Multi Q-table approach helps prevent reward loops, it still suffers from scalability issues due to the need to explicitly learn Q-values for every state-action pair. By replacing Q-tables with deep neural networks, our method enables generalization to unseen states and reduces the memory and sample complexity. Each neural network is responsible for learning the shortest path to a specific subgoal or the final destination, thereby reducing the learning burden on each network and allowing the system to scale more effectively with maze size. To develop and evaluate our approach, we began by modifying existing Deep Q-Learning codebases and comparing different strategies, including single-network baselines, multi-Q-table setups, and our final proposed model using multiple neural networks. The following subsections describe each version in detail, starting with the base implementation.

A. The Base Code

Our implementation's base code is based on the Tour De Flags (referred to as TDFB, B stands for Base) code written by Samy Zafrany [7] while another version (referred to as TDFM, M stands for Modified) was created by applying some modifications of our own to the base code to create an additional version of the Single Deep Q-Network algorithm for comparison to our proposal.

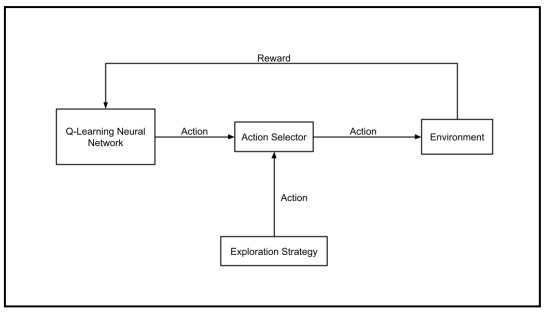

Fig. 3. The Single Deep Q-Network algorithm.

The base code uses Deep Q Learning with a single neural network acting as both the Policy and Target networks. It was modified, in the TDFM version, to use two neural networks to improve the stability of the learning process as described in the Background Theory section. One neural network acts as the Policy Network and is used for training and making predictions. The other neural network acts as the Target Network and is used to generate the predictions needed to calculate the targets. Both TDFB and TDFM followed the same algorithm, shown above in Fig. 3, to determine the maze's solution. TDFM also features some parameter optimizations, a modified reward function, and an improved exploration strategy discussed further in the Experimental Setup section below.

B. The Multi Q-Table Code

For the sake of comparison, two versions of the base code were created which used multiple Q-Tables in place of the original single Deep Q-Network. The first version, referred to as MQTB (Multiple Q-Table Base), uses the same reward function and exploration strategy as the original TDFB code but replaced the single Deep Q-Network with multiple Q-Tables which are switched and updated according to the algorithm and equations from [1] as shown in Fig. The second version, referred to as MQTM (Multiple Q-Table Modified), uses the same algorithm as the first but uses the reward function and exploration strategy that the TDFM code uses with some slight changes. Since the MQTM code uses multiple Q-Tables while TDFM uses only a single neural network, the epsilon decay and the list of previously visited states is modified to be administered on an individual Q-Table basis. Each Q-Table maintains its own list of previously visited states, used for the reward function, and epsilon value, used by the exploration strategy. Otherwise the reward function and the epsilon decay function are the same as the TDFM code.

C. Our Proposal

Our proposal, referred to as MDQN (Multiple Deep Q-Networks), built off the base code by modifying it to use multiple neural network pairs along with a slightly modified version of the algorithm used in Multi Q-Table Q-Learning, shown in Fig. 4 below. Our design uses N+1 pairs of neural networks to solve mazes, where N equals the number of subgoals in the maze. The Replay Memory is also split such that each neural network pair maintains its own Replay Memory separate from all other pairs. These changes enable each neural network pair to focus only on finding the shortest path to the closest reward and thus reduce the complexity of the task which the agent needs to learn. These changes also prevent the possibility of any reward looping from occurring by ensuring that each pair of neural networks never trains on reward from the subgoal which they are linked to. One neural network in each pair acts as the Policy Network while the other performs the role of the Target Network to provide stability to the learning process. Our design also uses the same neural network architecture, learning rate, replay memory, and batch size as is used in the TDFM version.

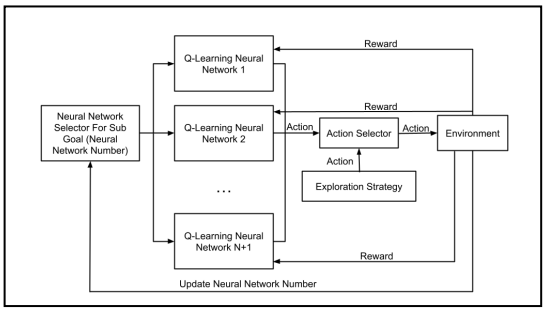

Fig. 4. The Multi Deep Q-Network algorithm.

Like in the Multi Q-Table Q-Learning theory, whenever the agent finds a subgoal, it switches to the corresponding pair of neural networks and continues training with them. The complete process for selecting neural networks is shown above in Fig. 4. Unlike the Multi Q-Table Q-Learning theory the neural networks are created and initialized before training begins. Lastly, our design also utilizes the same parameters, reward function, and improved exploration strategy as is used in the TDFM version with some slight changes. The epsilon decay and the penalty for visiting previous states are both modified to apply individually to each neural network pair rather than having a single epsilon value or list of previous states for all of them. This means each neural network has its own epsilon value based on the number of steps it has taken and its own list of previously visited states based only on the states it has visited. Otherwise the epsilon decay and penalty for visiting previous states works the same as in the TDFM code.

D. The Base Code vs Our Proposal

The most significant difference between the base code, TDFB, and our design, MDQN, is the use of multiple neural networks in place of TDFB's single neural network. This is reflected in the modifications made to the original algorithm, shown in Fig. 3, which resulted in our proposed algorithm, shown in Fig. 4. Specifically these modifications include the addition of multiple neural networks and a selector to determine which one should be used. As a result of these modifications each neural network is only required to find the path to the nearest subgoal. This is a much simpler task to complete than what is required of the TDFB algorithm, where the single neural network needs to find the optimal path to collect every subgoal. The use of multiple neural networks also allows each network to specialize in one local area of the maze while the single neural network used by TDFB needs to generalize to the entire maze. Assigning each neural network in the MDQN algorithm its own epsilon value also helps to balance out the exploration throughout the maze as each neural network explores its local area the most. In contrast, the single epsilon value used by the single neural network in the TDFB algorithm usually results in most of the exploration being done early on in the maze with later sections being poorly explored. As a result of these differences our design is able to more consistently

determine the optimal path to collect all the subgoals in fewer steps while the base code often fails to find any path to collect all the subgoals.

IV. Experimental Setup

The effectiveness of the proposed MDQN algorithm is examined by applying it to a shortest path problem and evaluating the results against other Q-Learning algorithms. The following is a short description of each of the algorithms tested. More information on each of the algorithms can be found in section III of this paper.

MDQN: The proposed Deep Q-Learning algorithm which uses multiple neural networks to divide up the task of finding the shortest path.

MQTB: A Q-Learning algorithm based on [1] which uses multiple Q-Tables to prevent reward loops and split up the task of learning the shortest path.

MQTM: A modified version of the MQTB code with an improved exploration strategy and reward function along with some parameter optimization.

TDFB: A Deep Q-Learning algorithm from the Tour De Flags code in [7] which uses a single neural network to find the shortest path to collect all the subgoals and reach the maze exit.

TDFM: A modified version of the TDFB code with an improved exploration strategy, reward function, and network structure along with some parameter optimization.

The 2D maze environment is modeled as a 2D array of ones and zeros where ones represent an open space the agent can move through and zeros represent a blocked space which impedes the agent's movement along. The list of subgoals is represented as tuples of integer values for the row and column of each subgoal's location. The agent starts in the top left corner of the maze each epoch and needs to find a path to collect all the subgoals and reach the end of the maze in the bottom right corner to win. Fig. 5, below, shows an example environment.

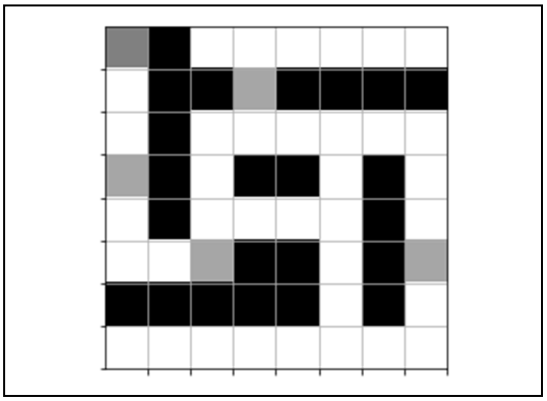

Fig. 5. Example maze environment with the agent in dark gray, the subgoals in light gray.

The agent chooses from four actions: up, down, left, or right. Should it try to move through a blocked space it is prevented from doing so and forced to remain in its current location.

The parameters for each of the algorithms are as follows. TDFB uses a learning rate of 0.001 and a discount rate of 0.97. MQTB and MQTM both use a learning rate of 0.5 and a discount rate of 0.99. TDFM and MDQN use a learning rate of 0.008 while using the same discount rate as TDFB (0.97).

The reward function provides a reward of one when the agent successfully collects a subgoal or reaches the final state. If the agent tries to move to a blocked or invalid state it receives a negative penalty which scales with the maze size. Finally, the agent receives a small negative penalty each time it moves to a valid state to encourage it to find the shortest path. This penalty is modified in the TDFM, MQTM, and MDQN versions to be doubled whenever the agent revisits a state which is already explored.

The original exploration strategy used by the base code in TDFB and MQTB is changed to better suit our design in the TDFM, MQTM, and MDQN versions. To make decisions about which actions to take the agent originally used a basic Epsilon Greedy strategy where the epsilon value started at 0.8 and decayed after each win to a minimum of 0.08. The agent chose to explore with a probability of epsilon and exploit with a probability of one minus epsilon. When the agent chose to explore, it picked an action with a uniform random distribution. In the TDFM, MQTM, and MDQN versions a hybrid exploration strategy consisting of Epsilon Greedy with an exponential decay and a modified form of epsilon greedy, created for this project, called Ranked Epsilon Greedy is used. In these versions the agent starts by using the same basic epsilon greedy exploration strategy used in TDFB but with an exponential decay for the epsilon value with each step, causing the agent to reduce its frequency of exploration quickly at first before eventually slowing down and allowing the agent to begin to converge to the optimal policy. Once the epsilon value has decayed sufficiently (less than 0.5 at a minimum) the exploration strategy switches to the Ranked Epsilon Greedy strategy and a linear epsilon decay with each step. The Ranked Epsilon Greedy strategy works identically to the basic Epsilon Greedy strategy, except when the agent chooses to explore, it ranks the actions based on their Q-Values and assigns a higher probability of exploration to actions with the highest Q-Values as follows:

If Action 1 Q-value > Action 2 Q-value > Action 3 Q-value > Action N Q-value:
Prob (Action 1) = 1- ε
Prob (Action 2) = ε (1-ε)
Prob (Action 3) = ε^2(1-ε)
Prob (Action N) = ε^{N-1}(1-ε)

This scheme enables the agent to use the information it has learned thus far to make more informed decisions when exploring. After the epsilon value decays below 0.35 the exploration strategy switches back to the basic Epsilon Greedy strategy described earlier. The TDFM, MQTM, and MDQN versions also use the same initial epsilon value of 0.8 but use a minimum value of 0.2 instead.

The Replay Memory is another parameter which was optimized for the MDQN algorithm. The Replay Memory size for TDFB is 4 times the number of states in the maze while the TDFM and MDQN algorithms use either 500 or 6 times the number of states in the maze depending

on which is larger. The Replay Memory batch size used for training the Deep Q-Networks was also modified. TDFB uses three quarters of the number of states in the maze to determine the Replay Memory batch size. In contrast, the TDFM and MDQN algorithms use the largest of 100 or two times the number of states in the maze as the Replay Memory batch size.

The neural networks used in the TDFB, TDFM, and MDQN algorithms all use the same architecture consisting of an input layer, 2 hidden layers, and an output layer all fully connected. The input layer and hidden layers are scaled to have as many neurons as there are states in the maze while the output layer always has four neurons, one for each action. The hidden layers use the Leaky ReLU activation function with an alpha value of 0.24.

The iterations to complete the maze is a discrete random variable, thus the distribution of the completion steps is represented as a Gaussian distribution. This hypothesis was tested using 50 runs of MQTB. Based on this assumption and the hardware limitations of the computing platform, a decision was made to limit the number of iterations the agent was allowed in each test run to find the solution to the maze – the step cap. The step cap per run is set as 1250 steps for the 5x5 mazes, 2450 steps for the 7x7 mazes, 3200 steps for the 8x8 mazes, and 4000 steps for the 10x10 mazes. Validation testing of the Gaussian model to represent the maze completion steps for the various algorithms is included in the results section.

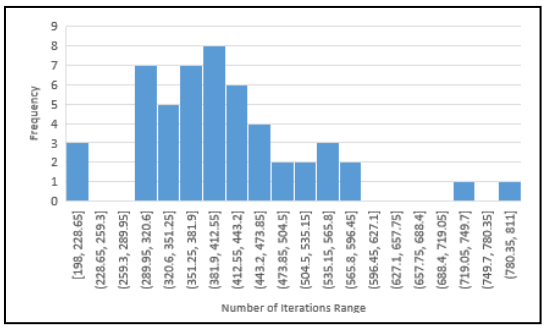

Fig.6. Histogram for 50 runs of MQTB Algorithm.

V. Results and Discussion

The results of our testing are summarized in Fig. 7 to Fig. 9 and Table I. The discussion of results is primarily centered around the number of iterations to complete the maze, as improving this was the main focus of the proposed algorithm. However, the real time to complete the maze as well as a note about the randomness in the results are also mentioned. Finally, the results are summarized, and the five algorithms are ranked from best to worst.

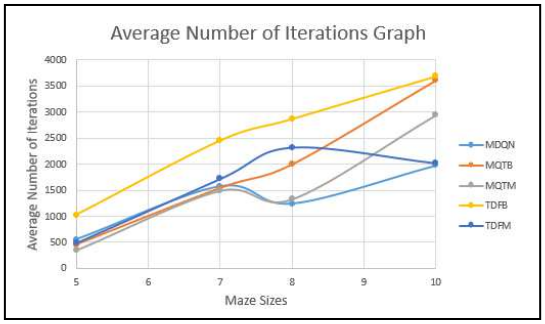

Fig.7. Average number of iterations to find the solution to the maze.

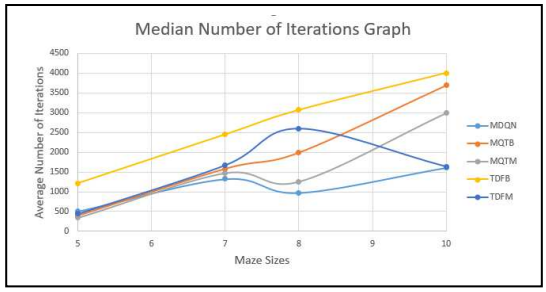

Fig.8. Median number of iterations to find the solution to the maze.

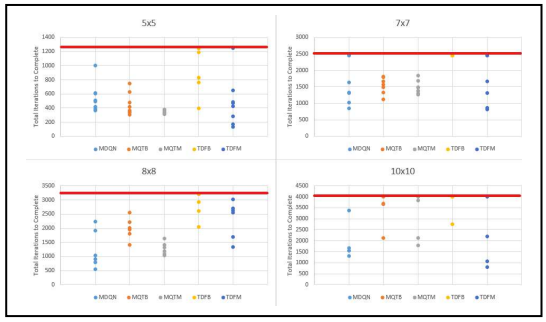

Fig.9. Number of iterations to find the solution to the maze

A. Number of Iterations

The first thing to notice from the Performance Measurement Results is that the base code, TDFB, performed worst among all algorithms and versions on all maze sizes. This is most obvious in Fig. 7 and Fig. 8. However, it is also apparent in Table 1 as many runs of TDFB did not complete training within the step cap. Fig. 8 (median number of iterations to completion) also shows that on the smallest mazes (5x5 and 7x7), all algorithms/versions excluding TDFB performed very similarly. This is in support of the theory that the DQNs do not have an advantage over Q-Tables for environments with fewer states (e.g., smaller mazes). The results of the testing on each maze are shown separately in Fig. 9. Looking at the results of the testing on the smallest maze, the 5x5, it seems that while the mean and median number of iterations shown in Fig. 7 and Fig. 8 are similar, from the individual runs it seems that TDFM can potentially perform better than the rest. This may be due to this maze being "too small" for the modified multiple model implementations (such as MQTM and MDQN) because, as discussed in Section III, these implementations use multiple epsilon values. Each of the epsilon values must decay individually, and thus may remain unnecessarily high (for mazes that require less exploration) for an extended number of iterations, increasing the number of iterations to solve the maze. That being said, this makes these implementations less reliant on good luck to solve the maze. As Fig. 9 shows, unlike TDFM, these implementations solved the maze in all test runs. Therefore, the modified multiple model implementations can be considered more robust. Furthermore, although MQTB does not use multiple epsilon values, it still uses a different exploration strategy that has a high exploration rate for an extended number of iterations. Therefore, TDFM and MQTB cannot be compared to say that the single model implementation TDFM outperformed the multiple model implementation MQTB. Rather, TDFB is the single model implementation with comparable exploration strategy to that of MQTB. Comparing these, it is clear that the multiple model implementation MQTB outperformed the single model implementation TDFB. The caveat here is that

MQTB is a Q-Table based implementation while TDFB is a DQN based implementation. This may have also contributed to the success of MQTB over TDFB since the 5x5 maze is considered small enough for Q-Table based implementations to succeed.

On the 7x7 maze, the Q-Table algorithms (MQTB and MQTM) were more consistent, as seen in Fig. 9. None of the other algorithms completed training within the maximum number of iterations all times for this maze. Aside from the mentioned fact that the 7x7 maze is relatively small, another reason behind this may be that the specific maze design is well suited for Q-Table algorithms rather than DQN algorithms. The 'maze' in fact only has a single possible path, which means that a large portion of the maze is covered by walls. In other words, the number of actual state action pairs is limited. Therefore, the primary weakness of Q-Table strategies, exponential increase in the size of the Q-Table with maze size, is not well emulated by the 7x7 maze. In this way, the maze can be considered well-suited for Q-Table strategies. Nevertheless, MDQN still performed slightly better in the median number of iterations to completion, making this a significant achievement. Another observation for this maze is that while TDFM performed similarly to MDQN (especially relative to their difference in 8x8), MDQN still performed slightly better despite the fact that there was no possibility of a reward loop in this maze. This is evidence in support of the other benefits of MDQN, such as localized exploration rate, reducing the complexity of the path needed to learn for each DQN, etc.

Table I. Iterations to Complete Various Maze Sizes Across Algorithms.

Maze Size	Run	MDQN	MQTB	MQTM	TDFB	TDFM
5	1	1002	335	361	1250	471
	2	491	475	332	1250	1250
	3	419	363	383	763	171
	4	368	749	368	1190	134
	5	604	348	336	834	426
	6	512	627	340	1250	652
	7	385	310	340	1250	282
	8	615	417	314	398	489
7	1	2450	1666	1299	2450	1671
	2	1315	1488	1844	2450	1320
	3	1027	1328	1384	2450	2450
	4	2450	1118	1483	2450	2450
	5	1631	1793	1470	2450	817
	6	847	1808	1680	2450	871
	7	1323	1576	1274	2450	2450
8	1	913	2015	1088	3200	3023
	2	1917	2224	1038	3200	2552
	3	2235	1805	1638	3200	1687
	4	553	1411	1321	2931	2631
	5	1032	1953	1409	2050	1343
	6	801	2553	1183	2611	2702
10	1	1559	2118	3836	4000	2202
	2	1663	4943	4000	2741	809
	3	3373	3676	1795	4000	1076
	4	1312	3696	2133	4000	4000

The next maze size is 8x8. This is a very interesting maze from among our sample mazes, because it was handcrafted as a maze that would be better suited to Multiple Model algorithms (MDQN, MQTM, and MQTB). That is because of the placement of the subgoals such that they require the agent to backtrack and change directions for most subgoals. The subgoals are also dispersed more evenly throughout the maze and the agent must also avoid taking shorter paths that reach the final destination without collecting all of the subgoals. Therefore, it was expected that the Multi Model algorithms would perform better on this maze. That is exactly what Fig. 8 shows. MQTM and MDQN performed the best, with MQTM slightly edging out MDQN. In fact, it seems very clear that the median number of iterations to completion edged up out of trend for TDFB and TDFM (Single Model algorithms) while at the same time edging down out of trend for MDQN, MQTM, and MQTB (Multiple Model algorithms). This is very strong evidence in support of the theory behind our method to solve problem of reward loops.

The final maze size in our testing is 10x10. In Fig. 8, it can be seen that both of the modified DQN algorithms (MDQN, TDFM) significantly outperformed the Q-Table algorithms on this maze. That is in support of the theory that DQNs become more advantageous for environments with many states (e.g., larger mazes). MDQN and TDFM performed roughly the same. This may be partly because of the very limited number of direction changes required to solve this maze, making the MDQN less advantageous. Random chance also likely helped TDFM, and therefore more testing is needed to reduce the effect of chance and build more concrete evidence in support of MDQN. That being said, MDQN is the only algorithm that solved the 10x10 maze in all of the test runs, as seen in Fig. 9 as well as Table 1. The average number of iterations for TDFM to solve the maze is helped by the fact that the maximum number of allowed iterations was used to calculate it for the run in which it did not complete the maze. The actual number of iterations to solve the maze would have likely been much worse, had the maximum not been imposed. Therefore, the results of the testing on the 10x10 maze point to the superiority of MDQN over the competing algorithms, including TDFM. Furthermore, these results also clearly support the hypothesis that our proposal would perform better than Q-Table algorithms on large mazes. If the results are extrapolated to even larger maze sizes with more direction changes and subgoals, it is likely that the advantage of MDQN would become obvious. To do that, more access to high-speed computing resources is required. Computing optimization may also be performed to reduce the amount of time per episode and gather data more easily.

B. Other Comments

An important thing to note about the results is that while our results suggest that MDQN outperforms the Q-Table algorithms (MQTB and MQTM) in the number of iterations to complete the maze, the same cannot be said about the real time to complete the maze. MDQN is a vastly more complex algorithm, especially as the size of the maze increases. That is primarily because each episode requires back propagation on many samples from the replay memory to train the policy network. The number of epochs, the size of the replay memory, and the size of the network all depend on the maze size. Therefore, each iteration (or episode) takes significantly longer for DQN based methods (several seconds) compared to Q-Table based methods (a few microseconds). This causes the real time taken by MDQN

(and any DQN based method) to complete the maze to increase exponentially with maze size. That being said, this time depends on hardware and can be improved to some extent with parallelization. For this reason, improvement in the number of iterations rather than real time per iteration was chosen as the primary objective for this project.

Note some of the results in Table I. did not complete training within the step cap. Table II. shows the calculations for the test of normality. The data were compared to a Gaussian modeling based on the mean and standard deviation calculated, with respect to the maze size and algorithm, and compared to the Kolmogorov-Smirnov table. Most of the data translate to KS test statistics lower than the critical value, which results in the failure to reject the null hypothesis, indicating that the Gaussian distribution is an adequate modeling of the total number iterations to complete the maze. This means that for the majority of the algorithms, the step cap did not affect the results significantly, except TDFB on the 7 by 7 maze. In Table I**,** it shows that the TDFB algorithm reached the step cap of 2450 all times.

Table II. Test for Normality.

Maze Size	5	7	8	10
Sample Size	8	7	6	5
MDQN	0.19	0.20	0.18	0.24
MQTB	0.18	0.13	0.15	0.27
MQTM	0.13	0.15	0.14	0.28
TDFB	0.32	NA	0.27	0.28
TDFM	0.16	0.27	0.30	0.20
Confidence Critical Value	0.45	0.48	0.52	0.62

One final note about the test runs that did not complete training within the step cap is that some of them failed because of lack of sufficient exploration of the entire maze. While some of this can be attributed to the main algorithm for DQN based methods, it may be more of a result of the exploration strategy. That is because a lack of exploration of the entire maze means that on some runs the agent never experienced any wins (collecting all subgoals and reaching the final destination) for the algorithm to be able to learn the optimal solution.

C. *Summary of Results*

To summarize, our results verify our hypothesis and support our design, MDQN, being superior to the compared algorithms in the number of iterations to complete the maze. This is best seen in Fig. 7 and Fig. 8, while Fig. 9 and Table 1 support these conclusions. A simple way to rank the algorithms is to take the median steps to completion for each maze size and add them all to get the total median steps for each algorithm. The result is as follows:

MDQN: 4408< MQTM: 6046.5< TDFM: 6350 < MQTB: 7636< TDFB: 10735.5

Based on this, the algorithms can be ranked as follows from best to worst:

1) MDQN (Our Solution)
2) MQTM (Modified Reference Paper Algorithm)
3) TDFM (Modified Base Code)
4) MQTB (Unmodified Reference Paper Algorithm)
5) TDFB (Unmodified Base Code)

It should be noted that Q-table algorithms are far more superior to DQN based algorithms, including MDQN, in the real time per iteration. However, this was expected and not an objective for this proposal. Finally, it should also be noted that many of the incomplete runs were a fault of chance rather than the algorithm. To further reduce the impact of chance on the results, more test runs would be needed in order to collect more data.

VI. Conclusion

This paper sought to propose a new Q-Learning algorithm for the purpose of solving multiple goal mazes in fewer iterations. Multiple goal mazes sometimes cause the agent to experience reward loops, which can be eliminated by using a Multiple Q-Table algorithm [1]. However, due to the exponential increase in the number of states with maze size, Q-Table algorithms are not suitable for large mazes [3]. Rather, Deep Q-Networks (DQNs) can be used to reduce the number of iterations required for solving large mazes [3]. This paper proposed using Multiple Deep Q-Networks (MDQNs) to solve large, multiple goal mazes with fewer iterations.

The proposed algorithm was tested in an experiment comparing the performance of the Multiple Q-Table algorithm and the single DQN algorithm, along with their variants. The results support the hypothesis that MDQNs can solve large, multi-goal mazes in fewer iterations. Future work could focus on reducing the real computation time per iteration for DQN-based algorithms, enabling scalability to even larger mazes. Additionally, developing more efficient exploration strategies tailored to multi-goal environments may further enhance performance. These directions present promising opportunities for advancing multi-goal reinforcement learning.

References

[1] N. Kantasewi, S. Marukatat, S. Thainimit and O. Manabu, "Multi Q-Table Q-Learning," 2019 10th International Conference of Information and Communication Technology for Embedded Systems (IC-ICTES), 2019, pp. 1-7, doi: 10.1109/ICTEmSys.2019.8695963.

[2] D. Osmanković and S. Konjicija, "Implementation of Q—Learning algorithm for solving maze problem," *2011 Proceedings of the 34th International Convention MIPRO*, pp. 1619-1622, 2011.

[3] M. Hacibeyoglu and A. Ahmet, "Reinforcement learning accelerated with artificial neural network for maze and search problems," *3rd International Conference on Human System Interaction*, pp. 124-127, 2010.

[4] R. S. Sutton and A. G. Barto, Reinforcement learning: An introduction, Cambridge, Massachusetts: MIT press, 2018.

[5] V. Mnih*et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi: 10.1038/nature14236.

[6] S. Zafrany, "Deep Reinforcement Learning for Maze Solving," [Online]. Available: https://www.samyzaf.com/ML/rl/qmaze.html. [Accessed 10 02 2022].

[7] S. Zafrany, "Deep Reinforcement Learning: The Tour De Flags test case,"[Online]. Available: https://www.samyzaf.com/ML/tdf/tdf.html. [Accessed 10 02 2022].

[8] A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for Q-learning in random stochastic mazes," *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, Athens, Greece, Dec. 2016.