Finding Eulerian tours in mazes using a memory-augmented fixed policy function

Mahrad Pisheh Var[1], Michael Fairbank[2], and Spyridon Samothrakis[3]

[1] University of Essex,
mpishe@essex.ac.uk,
[2] University of Essex,
m.fairbank@essex.ac.uk
[3] University of Essex,
ssamot@essex.ac.uk

Abstract. This paper describes a simple memory augmentation technique that employs tabular Q-learning to solve binary cell structured mazes with exits generated randomly at the start of each solution attempt. A standard tabular Q-learning can solve any maze with continuous learning; however, if the learning is stopped and the policy is frozen, the agent will not adapt to solve newly generated exits. To avoid using Recurrent Neural Networks RNNs to solve memory-required tasks, we designed and implemented a simple external memory to remember the agent's cell visit history. This memory also expands the state information to hold more information, assisting tabular Q-learning in distinguishing its path from entering and exiting a maze corridor. Experiments on five maze problems of varying complexity are presented. The maze has two and four predefined exits; the exit will be randomly assigned at the start of each solution attempt. The results show that tabular Q-learning with a frozen policy can outperform standard deep-learning algorithms without incorporating RNNs into the model structure.

Keywords: Tabular Q-learning, Augmented memory in Q-learning, Maze Navigation, Augmented memory in Q-learning, Maze Navigation and Eulerian tours.

1 Introduction

A maze is a simple and discrete technique to demonstrate exploratory tasks. The Q-learning algorithm can solve the problem of finding a path through a fixed maze [1]. The traditional Q-learning method uses a Q-table, initialised with potentially arbitrary fixed values [2], to assign to each state a Q-value for each of the possible actions allowed from that state.

The exploration tasks become complicated when the exit locations of the maze are randomised at the start of every solution attempt. In this case, the agent is given a new exploration assignment at the beginning of the maze; this will require the agent to adapt.

Traditional tabular Q-learning methods with continuous learning can adapt and solve the same maze if the exit location is moved, provided the Q-learning algorithm continues to learn indefinitely. However, once the traditional tabular Q-learning algorithm has stopped learning, i.e. has stopped updating the Q-values, the algorithm can only handle one possible exploration task. The frozen policy learned by the traditional tabular Q-learning will prevent the agent from backtracking upon entering a corridor.

The agent must have a memory to remember that it entered a dead-end corridor and use that to backtrack and exit. Furthermore, once the agent enters a dead-end corridor in the maze, it requires the state information fed into the algorithm to contain information that helps the algorithm distinguish its path from entering and exiting the corridor. Recurrent Neural Networks RNNs are one approach to embed memory, which allows backtracking out of dead-end corridors [3]. However, the computation of this neural network is slow, and training can be difficult.

The idea of accomplishing adaptation with frozen policy generated by tabular Q-learning is illustrated in this study. This study aims to eliminate the need for recurrent neural networks [4] and replace them with a more straightforward system that allows the agent's fixed strategy to adjust to shifting reward conditions.

The system includes an additional table as a memory that records the history of the agent's cell visits. The memory is accessed by the agent's current location and action. In addition, the state is expanded to include the agent's present location and the memory value for the neighbouring cell visit, where it tracks the agent's history of cell visits.

This memory significantly helped the tabular Q-learning find all the environment's exits. Therefore, by including memory as part of the state vector and working with the tabular Q-learning architecture, it is possible to investigate the potential exits of the environment with the flexibility to avoid separate training Q-tables for each exit.

Our simple solution can solve all "perfect mazes". A "perfect maze" is one in which a single path can connect any two cells. Our result also extends to mazes with loops. However, although our simple solution can solve exit points being moved, it requires the maze structure and starting point to be fixed.

The structure of the rest of this paper is as follows: Section 2 consists of the introduction to related research on memory usage to complete memory-based tasks, followed by a discussion about the significance of memory in partial-observable environments. Section 3 includes the knowledge of discrete reinforcement learning and its application to the maze environment. Moreover, section 3 describes the memory modification we implemented and applied to the maze environments, and its functionality was demonstrated with tabular Q-learning. In addition, a proof of sufficiency for solving mazes using our method is included in section 3. Section 4 includes details about the experiment setup and reinforcement learning agents we prepared to demonstrate in the environment. The results are shown in section 5. This study will conclude with a discussion in

section 6 about what we achieved and how this simple system can bypass using complicated recurrent systems in environments with shifting reward conditions.

2 Related work

An extended memory management system, known as a memory-based learning system [5], divides the input space into either static or dynamic sub-regions to store and retrieve valuable information.

There are different key generalisation strategies used by memory-based learning systems, which are:

- Nearest-Neighbour searches are a form of a proximity search that is mainly used in optimisation to find the point in a given set closest to a given point [6].
- Space decomposition methods are solutions to various problems and the design of algorithms in which the basic idea is to decompose the problem into sub-problems [7].
- Hierarchical clustering HCA [8] is a cluster analysis method that aims to create a cluster hierarchy. Agglomerative strategies for hierarchical clustering are a "bottom-up" approach in which each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. All observations begin in one cluster in a divisive, and splits are performed recursively as one moves down the hierarchy.

The system we introduce in this paper is not embedded in the neural network and acts as an external memory. In comparison, Deep reinforcement learning (DRL) includes the capability to learn optimal policy in an end-to-end manner without relying on feature engineering [9].

However, in an environment where the state space is not fully shown, the DRL algorithms require external or internal processes to compensate for their lack of knowledge about whether the observation is complete enough [10]. Sensors can alone solve this problem. However, there are many issues with sensors, including sensor sensitivity limitation, noise and the quality of the data the sensors gather [11].

Whether the structure of the environment changes or there is a change in the reward conditions regime, any learning algorithm requires memory to adapt [12].

In previous studies, algorithms such as deep Q-learning were preferred in solving complex mazes. For instance, in [13], they compared different policy-picking algorithms for Q-learning to create a ratio of balance between exploration and exploitation. Moreover, they suggested using softmax instead of e-greedy to create that balance.

Similarly, in [14], they explored the same idea of maintaining the balance in behaviour policy by adopting a Bayesian approach for uncertain information. In conventional Q-learning, the uncertain observation information forces the Q-learning to choose a random action that will lead to higher exploration; however, the [14] authors used a Bayesian approach to compute a myopic approximation that maintained the balance between exploration and exploitation.

This approach resulted in Q-value distribution manipulation that controlled the agent's behaviour. The work done in [14] is quite similar to our idea, where we manipulated the dimension of our Q-values based on the problem description instead, giving the agent broader control based on the action it took at each state.

Architectures developed by [15] to tackle mazes combined convolutional neural net and auxiliary prediction to apply pixel control to the sensory data; the skewed replay buffer was used in reward prediction ahead of time to predict the rewards of the unobserved time step. In [15], authors created an unsupervised reinforcement and auxiliary learning agent. The convolutional neural network picked up pixels later used in an LSTM network to control the agent's movements. The incentives, such as images on the wall and 3-dimensional apple-shaped exits, were used to motivate the agent to apply pixel control to the data gathered by the observatory sensors [15].

Authors in [15] argued that adding memory to the agent is for remembering elements from the past and applying that to future decisions. However, the size of the trajectory to reach the exit directly affects memory. The [15] authors used their solution with sensory hints planned on the maze to solve similar mazes with sensory motors capturing the agent's point of view. In comparison, in [16], they tackled partially observable Markov decision problems (POMDPs) with recurrent policy gradient implemented as a model-free reinforcement learning method. Furthermore, in [16], they included a policy gradient for a recurrent neural network; this was only possible by a back-propagation algorithm.

Furthermore, Authors in [16] mentioned discrete control in settings of long-term dependency, a T-maze where the path leading to the T-junction was randomised from 10 to 100 cells. The results show that the recurrent gradient policy outperformed value-based methods.

Similarly, the [17] authors combined deep Q-learning with a recurrent neural network where they have used a data structure to store a chain of states, and the chain of states will be fed into the RNN to output the Q-values. Two different Q-networks are used as the "actual Q-values" and the predicted ones. Therefore, the agent must estimate from estimation to perform like Q-learning [17].

Previous studies used RNN to solve memory-required tasks. However, RNNs are computationally expensive, and their optimal hyperparameter tuning can be time-consuming. This paper introduces a simplified maze environment to demonstrate memory-required tasks and tackles it with standard tabular Q-learning. With continuous learning, a standard tabular Q-learning can solve any maze; however, if learning is halted and the policy is frozen, the agent will not adapt to solve newly generated exits. We designed and implemented a simple external memory to remember the agent's cell visit history and avoid using RNNs.

3 Discrete Reinforcement Learning in a Maze Environment

The maze we built is a standard discrete-valued reinforcement learning problem where the discrete state space is denoted by $\mathbb{S}$, and the discrete action space is denoted by $\mathbb{A}$.

The maze is represented by a matrix M of binary values 0 or 1. The matrix (maze) M is of size height $\times$ width. Each matrix element represents a maze cell (1=blocked or 0=open).

For example, the maze in Fig. 2b is represented by the matrix:

$$M = \begin{bmatrix} 1\,1\,1\,1\,1\,1\,1 \\ 1\,0\,1\,0\,0\,0\,1 \\ 1\,0\,1\,1\,1\,0\,1 \\ 1\,0\,1\,0\,0\,0\,1 \\ 1\,0\,1\,1\,1\,0\,1 \\ 1\,0\,0\,0\,0\,0\,1 \\ 1\,1\,1\,1\,1\,1\,1 \end{bmatrix}$$

At time step $t = 0$, the agent starts from a given fixed start point, e.g. as shown in Fig. 2b. Then, at each time step t, the agent has state $s_t \in \mathbb{S}$ and chooses action $a_t \in \mathbb{A}$. Finally, the environment responds by moving the agent to a new state s_{t+1} and giving a real-valued reward r_t.

The agent attempts to an open cell by applying a valid action a_t from $\mathbb{A} = \{\text{north}, \text{south}, \text{east}, \text{west}\}$. The new state's positional values will not be updated if the arriving cell's $M(y, x)$ returns the value of 1, indicating that the agent collided with the wall.

The arriving cell's value from $M(y, x)$ will determine the agent's new position. To illustrate this explicitly, we have the standard update equations for moving around a discrete-cell maze:

$$(y_{t+1}, x_{t+1}) \leftarrow \begin{cases} (y_t - 1, x_t) & \text{if } a_t = \text{North and } M(y_t - 1, x_t) = 0 \\ (y_t + 1, x_t) & \text{if } a_t = \text{South and } M(y_t + 1, x_t) = 0 \\ (y_t, x_t + 1) & \text{if } a_t = \text{East and } M(y_t, x_t + 1) = 0 \\ (y_t, x_t - 1) & \text{if } a_t = \text{West and } M(y_t, x_t - 1) = 0 \\ (y_t, x_t) & \text{otherwise} \end{cases} \tag{1}$$

At every time step t, on taking action a_t, the agent receives an instantaneous reward of:

$$r_t = -1, \tag{2}$$

regardless of whether the action led to bumping into a wall.

The episode terminates when the agent reaches the exit (where $(y_t, x_t) = (y_{\text{exit}}, x_{\text{exit}})$) or runs out of steps.

The agent repeatedly takes actions, accumulating rewards, until a terminal state is reached (i.e. until the maze's exit is found or the time expires). An

episode is a state transition sequence from the start to the terminal state. The total discounted reward accumulated by the agent is:

$$R = \sum_{t} \gamma^t r_t \tag{3}$$

where $0 < \gamma \leq 1$ is a discount factor.

As purely negative reward is accumulated over the episode (via (2) and (3)), it is to the agent's advantage to get to the exit of the maze as quickly as possible to maximise the total reward received.

Actions are chosen by a policy function $\pi : \mathbb{S} \to \mathbb{A}$. The learning objective is to find a policy function that maximises the expectation of R.

A deterministic policy function π will always specify which unique direction to move. Upon entering a dead-end corridor with a deterministic policy function, it is impossible to backtrack out of that corridor. We define a memory-based solution to this problem in the following subsection.

3.1 Maze environment with memory modification

For each maze cell (y, x), a memory vector $\vec{c}_{(y,x)}$ is defined and stored in a table C with dimensions $(\text{height}, \text{width}, 4)$. The memory vector $\vec{c}_{(y,x)}$ is represented in (4) where for each direction, $d \in \{\text{north}, \text{south}, \text{east}, \text{west}\}$, the value of $C(x, y, d)$ holds a one if the agent has previously travelled from (y, x) in the direction d, and a zero otherwise.

$$\vec{c}_{(y,x)} = \begin{bmatrix} C(y, x, \text{north}) \\ C(y, x, \text{south}) \\ C(y, x, \text{east}) \\ C(y, x, \text{west}) \end{bmatrix} \tag{4}$$

The table C is initialised with zeros. Then, as shown in (5), we update the table's vector value $C(y_t, x_t, a_t)$ after the agent takes action a_t. Our table C will retain the changed values throughout the episode, and when a new episode begins, the table values will revert to 0.

$$C(y_t, x_t, a_t) \leftarrow 1 \tag{5}$$

In the above section, the agent's state was described by two numbers, (y_t, x_t). In the case of a maze environment with memory modification, we extended the state to hold additional values so that the state is now described by:

$$s_t = (y_t, x_t, C(y_t, x_t, \text{north}), C(y_t, x_t, \text{south}), C(y_t, x_t, \text{east}), C(y_t, x_t, \text{west})) \tag{6}$$

In short, the state is described by:

$$s_t = (y_t, x_t, \vec{c}_{(y_t, x_t)}) \tag{7}$$

The policy function without the memory modification was defined by $\pi(y_t, x_t)$. The policy function for the environment with memory arguments is now expanded to hold $(y_t, x_t, \vec{c}_{(y_t,x_t)})$. The policy function for the environment with memory modification is described by:

$$a_t = \pi(y_t, x_t, \vec{c}_{(y_t,x_t)}) \tag{8}$$

The C table stores memory by remembering the corridors the agent previously explored. Entering a corridor from direction d_t updates that direction in $C(y_t, x_t, d_t)$ to 1. After reaching an already visited maze cell, the state is represented differently while the agent reverses out of the corridor.

For example, if the agent only moved east in the corridor, the table $C(y_t, x_t, \text{east})$ value in $\vec{c}_{(y_t,x_t)}$ is updated to hold 1. When the agent decides to exit the corridor, the state representation will now show $s_t = (y_t, x_t, 0, 0, 1, 0)$, which differs from when it entered the corridor $s_{t'} = (y_{t'}, x_{t'}, 0, 0, 0, 0)$.

Comparatively, more straightforward solutions might be more appealing, such as remembering if the agent visited the arriving cell before. However, that would only require a table of shape $(\text{height}, \text{width})$. If the agent accesses the same cell more than twice, the simpler, more appealing design will not function. The agent can only explore two corridors due to the restriction imposed by the binary value for each cell, and it will fail if the agent needs to return to the same cell to explore more junction pathways.

3.2 Proof of sufficiency for solving mazes with memory modification

Different structures of binary cell mazes can provide different paths for the player to take to reach its target. To traverse from one cell to its neighbouring cells, the player must apply a valid action to reach the neighbouring unobstructed cells.

Tree graphs can represent a perfect maze in which every cell (tree nodes) can be reached, consisting of just one direct route from one cell to any other (tree branches); a perfect maze can be fully explored by an Euler tour (illustrated in Fig. 1) [18].

In Fig. 1, we can observe a sequence of actions made by the Euler tour. This sequence of numbers shows that the agent traverses down until it reaches a dead-end, traverses back up to the cell where its path was divided, and then chooses from the available unexplored paths. Recursively, the agent follows this rule until it fully explores all the possible sub-paths, which leads to the agent visiting every leaf node in the tree.

Hypothetically, suppose an exit is switched between the leaf nodes shown in Fig. 1 at every episode reset where the agent starts at the root node. In that case, the Euler tour can eventually find the exit by exploring every single leaf of the tree. Similarly, in our proposed problem, a maze that can switch the exit between two or more cells can be represented in Fig. 1, where each leaf can contain the exit.

A fixed policy function with no further learning, which receives inputs (y, x), cannot perform the series of actions shown in Fig. 1. For example, after finding

the exit at the far left of the tree, the agent cannot backtrack entirely to its branch-off nodes to explore other paths to other leaf nodes; the deterministic status of the policies limits further exploration.

On the other hand, $(y, x, \vec{c}_{(y_t,x_t)})$ input allows the agent to diversify states by training them separately with the help of the extended state values. This unparalleled access allows the agent to recursively backtrack and explore other branches until there are no unexplored paths; this behaviour is very similar to the Eulurean tree exploration shown in Fig. 1.

If the maze is not "perfect", containing loops, it is possible to imagine two different states and actions pair can reach a specific cell. For example, in Fig. 1, we can connect any randomly selected two leaves in the tree graph and create a looped maze. The fixed policy function will be stuck in the loop by entering it. Upon reaching the branch off node where it starts the loop, the agent with a fixed policy function cannot diversify the inputs (y, x), therefore, cannot exit the loop section of the maze.

The difference between actions to reach the same state will allow our method to update different table entries $C(y_t, x_t, d_t)$ every time it reaches the same cell in the maze, allowing different inputs to be provided. Moreover, similar to our explanation with perfect mazes, the agent will recursively explore down and then upwards towards branch cells, proving that the agent will explore every single leaf cell existing in the maze.

The above two sections demonstrate that it is possible for the greedy policy given by (11) to represent a full tour of the maze, which solves the maze even when the exit location is randomised. This demonstration is in contrast to the greedy policy in ordinary Q-learning. The Q-learning we mentioned in (10) cannot solve mazes with randomised exit points without further learning.

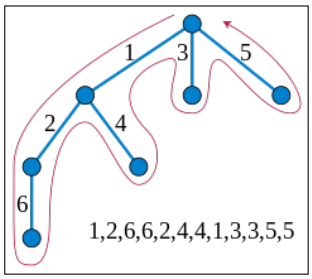


Fig. 1: Euler tour of tree graph, taken from [18]

3.3 Standard Tabular Q-learning

Over several discrete time steps, an agent is set to interact with an environment. For example, in standard reinforcement learning, this process is done at time t, where the agent receives a set of observation data o_t.

The agent chooses the optimal policy a_t from the available action list. Then, the action is applied to the environment, and the next state s_{t+1} and reward r_t is returned.

Q-learning [2], or deep Q-leaning [19] are standard value-based reinforcement learning algorithms.

The value function is expected to return feedback from state s at time-step t. when actions are selected $\pi(a|s)$, therefore, value function is calculated $V^{\pi}(s) = E[R_t : \infty | s_t = s, \pi]$.

If the agent takes an action a_t in state s_t and follows the optimal policy $\pi(s|s)$ the $Q^{\pi}(s, a) = E[R_{t:\infty}|s_t = s, a_t = a, \pi]$ is the expected value returned.

Q-learning uses the temporal difference (TD) to estimate the expected value. The TD is the agent's experience through episodes without prior knowledge of the environment. Therefore, mainly Q-learning uses a table of $Q[S, A]$ to hold the Q-values where each estimates the expected value. Commonly, the Bellman equation is used in returning the expected value $Q^{\pi}(s_t, a_t) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|s_t, a_t]$.

The update function uses the maximisation of the value function Q to return expected future reward $newQ_{(s,a)} = Q_{(s,a)} + \alpha[R_{(s,a)} + \gamma maxQ'(s', a') - Q_{(s,a)}]$. An episode is a state transition sequence from the start to the terminal state.

Actions are chosen by a policy function $\pi : \mathbb{S} \rightarrow \mathbb{A}$. The learning objective is to find a policy function that maximises the expectation of R. For example, in Q-learning, [2]. With a tabular representation for the Q-function ($Q(s, a)$), after each action, a_t is chosen from state s_t following the given policy π, after which the agent is observed to move to a new state s_{t+1} and to have received reward r_t, the Q table is updated by:

$$\Delta Q(s_t, a_t) = \alpha \left(r_t + \gamma \left(\max_{a'} Q(s_{t+1}, a') \right) - Q(s_t, a_t) \right). \tag{9}$$

The greedy policy on the tabular Q function is defined to choose action a from state s by:

$$a = \arg\max_{a'} Q(s, a') \tag{10}$$

3.4 Integrating memory in Tabular Q-learning

Integrating our memory in tabular Q-learning will use an ordinary Q-learning, which uses the new state vector $s_t = (y_t, x_t, \vec{c}_{(y_t,x_t)})$. The greedy policy on the augmented Q-function is defined with an action a_t from state s_t by:

$$a_t = \arg\max_{a'} Q(y_t, x_t, \vec{c}_{(y_t,x_t)}, a') \tag{11}$$

a' indicates all the possible actions available to the agent, and $\vec{c}_{(y_t,x_t)}$ returns all the four directions neighbouring cell history of visits from (y_t, x_t) shown in (4).

The Q-learning update interacting with our augmented memory maze environment is shown in (12). In this case, The Q-learning algorithm accesses the Q-table according to its current state $s_t = (y_t, x_t, \vec{c}_{(y_t,x_t)})$.

$$\begin{aligned} &\Delta Q(y_t, x_t, \vec{c}_{(y_t,x_t)}, a_t) \\ =&\alpha_t(r_t + \gamma \left(\max_{a'} Q(y_{t+1}, x_{t+1}, \vec{c}_{(y_{t+1},x_{t+1})}, a')\right) - Q(y_t, x_t, \vec{c}_{(y_t,x_t)}, a_t)), \end{aligned} \quad (12)$$

where $\alpha > 0$ is the learning rate.

Equations (12), (5) and (11) show how an ordinary Q-learning interacts with our new state vector.

3.5 Simplified wall-following algorithm

It can be argued that known algorithms such as the "wall-follower" algorithm [20] can be used instead of memory-augmented tabular Q-learning. This method, also known as the left-hand rule method, can solve any non-looped maze with a guarantee of not getting lost and will reach a different exit if there is one. However, this method does not solve the maze structures with loops.

If we want to train a Q-table to have a "wall-follower" algorithm behaviour, we can modify the observation of the state to hold $\vec{o}_t = (w(\text{west}), w(\text{east}), w(\text{north}), w_(\text{south}), d_{entry})$; The w inputs are the sensory data which will return 1 if there is a blocked cell in the specified direction and 0 otherwise. The d_{entry} is the agent's direction entered the current cell.

In this representation, the policy is a function of the observation vector $\vec{o}_t$. The agent's observation is limited to its surrounding walls and the direction in which the agent entered the current cell; the only way to solve the maze is to follow the wall, which is more exploitation than finding a solution by exploration.

Therefore we can train a Q-table with shapes of $(2, 2, 2, 2, 4, 4)$ to represent the state given to the "wall-follower" algorithm.

We include this kind of agent in the experiments below for comparison against the memory-augmentation method.

4 Experiment Setup

Five distinct algorithms were intended to be compared to test on five different mazes; in each case, the augmented memory is added to the algorithm, and the results are compared. These algorithms are as follows:

- Conventional Q-learning algorithm.
- Proximal policy optimisation algorithm PPO.
- A synchronous, deterministic variant of Asynchronous Advantage Actor Critic (A3C) A2C.
- Deep Q Network (DQN) builds on Fitted Q-Iteration (FQI), using a replay buffer, a target network and gradient clipping.
- Conventional Q-learning trained to behave like a "wall-following" algorithm.

The PPO, A2C and DQN were taken from [21], and the specific hyper-parameters were set to their default values. The inputs to PPO, A2C and DQN will benefit from our external memory, adding more details to their neural network input.

For the conventional Q-learning algorithm, two inputs are acquired from the agent's current position (y, x) in case of no memory augmentation. However, for the memory augmented method, the input is expanded to contain six values to additionally hold a vector $\vec{c}_{(y_t,x_t)}$ shown in (7).

For the reinforcement learning algorithms such as PPO, A2C and DQN, the network architecture of Critic and Actor, where relevant for each algorithm, consisted of a multi-layer fully-connected neural network. In case of no memory augmentation, the observation will hold the position of the agent $o_t = (y_t, x_t)$, where it will be pre-processed into a one-hot encoded form. The input for each reinforcement learning algorithm included a one-hot encoded vector with the shape of the maze height added to the maze width.

In case of memory augmentation, the state is expanded to hold a vector $\vec{c}_{(y_t,x_t)}$ shown in (7) where contains four values in the range of 0 and 1 in addition to the agent's current position, therefore, after one-hot encode pre-processing, the input will hold the sum of maze's height and width added to eight extra values for the memory augmented part of the observation.

Two hidden layer and 64 nodes are designed for each algorithm's network architecture. The PPO and A2C algorithms used the tanh activation function; the DQN algorithm included the ReLU activation function in its network architecture.

The agent's x and y coordinates are one-hot encoded. Also, each element of the $\vec{C}$ shown in (4) is one hot encoded. Hence there are $height + width + 4 * 2$ inputs to the neural network.

The last layer's activation function for all reinforcement learning algorithms was an identity activation function.

The learning rate was set to 0.01 and the discount factor was set to $\gamma = 0.9$ The epsilon-greedy policy [22] was used to apply randomness in choosing the actions with a 0.1 chance to occur. The epsilon-greedy policy allows exploration in training, which was removed in the validation phase of our experiments. Each algorithm was given 500 time steps to find the exit in one episode. At the start of each episode, the exit is moved to its next possible cell.

Five different maze structure environments were created; these environments are shown in Fig. 2. The exit is indicated with a yellow circle and named "EXIT". The starting cell is drawn with a red square; the empty cross-hatched space shows the maze wall where the agent cannot move into these areas. Each maze environment contains two or four possible exit cells (exits) depending on the maze's structure.

Each maze has a set of possible exits, and one exit is chosen for each episode. Hence when the agent explores the maze, there is exactly one exit it is looking for. At the start of each different episode, the exit is rotated through the set of possible exits for that maze, so each time the agent starts a new episode, the exit will have moved from the last time it explored. For each episode, the agent

has 500 time steps to find the exit before the time runs out. After experimenting with all the possible exits, the total number of accumulated time steps to reach each exit is recorded.

These environments shown in Fig. 2 hold a common feature where the agent must decide to take one of the divided paths where only one can lead to the exit. The agent starts on the red square cell and will have to output actions to move into the allowed cells. In case of a collision with a wall, the agent will remain in the same cell, and one step will count towards the total steps.

The experiment was created to run on 10 different trials; in each trial, the agent was trained for 100,000 episodes, and the weights were updated after each action was taken by the agent during each episode.

The agent must devise a movement strategy to find and visit all possible environmental exits. For instance, in the small 3-cell maze shown in Fig. 2a, the optimal number of steps to reach the exit cell on the right side of the maze is 1. After reaching the exit cell, the agent's position is reset. Therefore, the total time steps to reach the exit on the left side of the maze through the exit we already visited is 3. To reach both exits, we get 4 as the optimal number of steps to reach both exits.

In Fig. 2, five different test mazes are shown and can be named and described in table 1. First, the table introduces each maze environment, followed by the number of open cells and optimal steps to reach all the exits planned for the maze.

Table 1: List of mazes followed by the number of cells capable of being traversed.

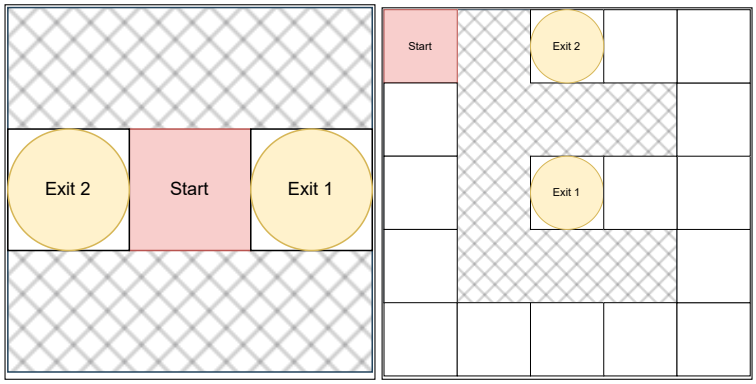
Maze Name	Total Traversable Cells
Small Corridor Maze (Fig. 2a)	3
Long Corridor Maze (Fig. 2b)	18
T-shaped Maze (Fig. 2c)	9
Cross Maze (Fig. 2d)	9
Complex Looped Maze (Fig. 2e)	188

Off-policy and on-policy are the standard methods used in Q-learning; off-policy was chosen for the Q-learning method because it changes Q-values independently from its previous policies [2].

Since the "wall-following" Q-learning algorithm does not need to be trained on each maze, we will only train it on the cross maze shown in Fig. 2d, and we will validate it on the rest of the mazes defined in table 1.

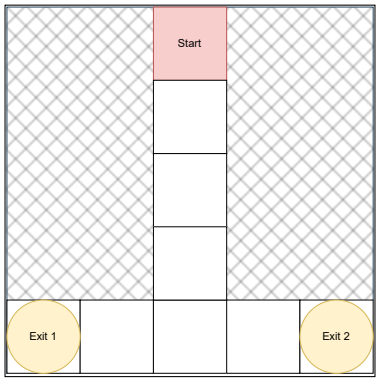
5 Results

Learning algorithms such as tabular Q-learning, DQN, A2C, and PPO were tested on each map represented in table 2; the agent performs well when it min-

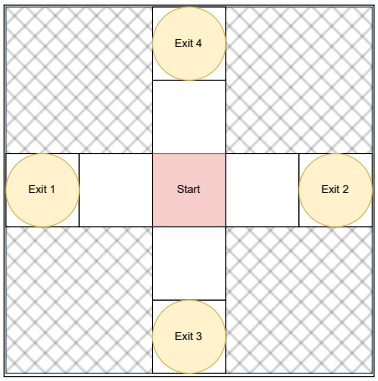


(a) Small corridor maze

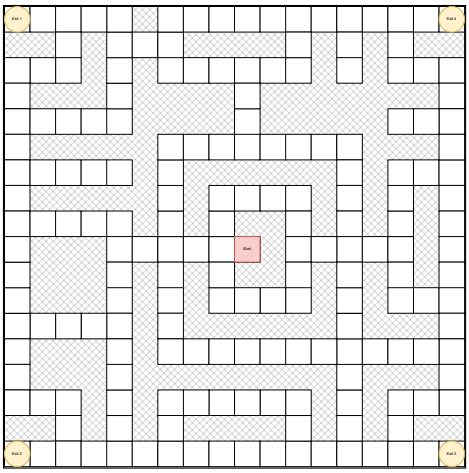
(b) Long corridor maze



(c) T-shaped maze



(d) Cross maze



(e) The complex looped maze with 20 by 20 overall size.

Fig. 2: Visual representation of the mazes purposed for the experiment.

imises the accumulated steps. The results are compared against the A^* searching algorithm to reach all the exits in the maze.

Fig. 3 shows the path the tabular Q-learning method with memory took to reach the exit. The white cells indicate the unvisited cells, and the red cells correspond to the agent's path; the cross-hatched areas indicate the blocked areas where the agent cannot enter. Each maze's exits will rotate during the 100,000 training iteration; the policies are frozen after one episode. A fresh evaluation dedicated episode starts with the policies frozen and no epsilon-greedy.

Table 2: Performance of each algorithm at 100,000 episodes with a frozen policy and no epsilon-greedy exploration. The errors indicate the standard error over 10 trials.

Maze Name	A^* Accumulated Steps	Algorithm method	Average total steps reaching all exits	
			With external memory	Without external memory
Small Corridor	4	Tabular Q-learning	4.0 ± 0.0	501.0 ± 0.0
		PPO	77.22 ± 34.61	600.8 ± 124.47
		A2C	361.33 ± 72.94	800.4 ± 81.48
		DQN	445.88 ± 55.11	47.2 ± 10.83
Long Corridor maze	30	Tabular Q-learning	30.4 ± 0.4	902.6 ± 64.93
		PPO	215.0 ± 67.33	1000.0 ± 0.0
		A2C	786.44 ± 84.44	1000.0 ± 0.0
		DQN	1000.0 ± 0.0	948.4 ± 35.07
T-shaped maze	16	Tabular Q-learning	16.0 ± 0.0	703.6 ± 80.67
		PPO	47.66 ± 8.51	1000.0 ± 0.0
		A2C	230.22 ± 105.02	1000.0 ± 0.0
		DQN	1000.0 ± 0.0	738.4 ± 93.82
Cross maze	32	Tabular Q-learning	32 ± 0.0	1701.2 ± 81.32
		PPO	652.22 ± 162.91	1352.6 ± 149.4
		A2C	1452.77 ± 49.97	1452.2 ± 156.6
		DQN	1557.44 ± 55.32	887.0 ± 75.32
Complex Looped maze	232	Tabular Q-learning	948.0 ± 52.0	1951.7 ± 48.3
		PPO	2000.0 ± 0.0	2000.0 ± 0.0
		A2C	1950.55 ± 49.44	2000.0 ± 0.0
		DQN	2000.0 ± 0.0	1667.9 ± 77.10

The "wall-follower" results for each maze environment are shown in table 3.

6 Discussion

In table 2, we can observe each algorithm method's average total steps to reach all possible exits in each maze defined in table 1. The small corridor maze consists of 3 open cells, and the agent starts in the middle of the three open cells. To achieve an optimal accumulated step to reach both exits, the agent has to

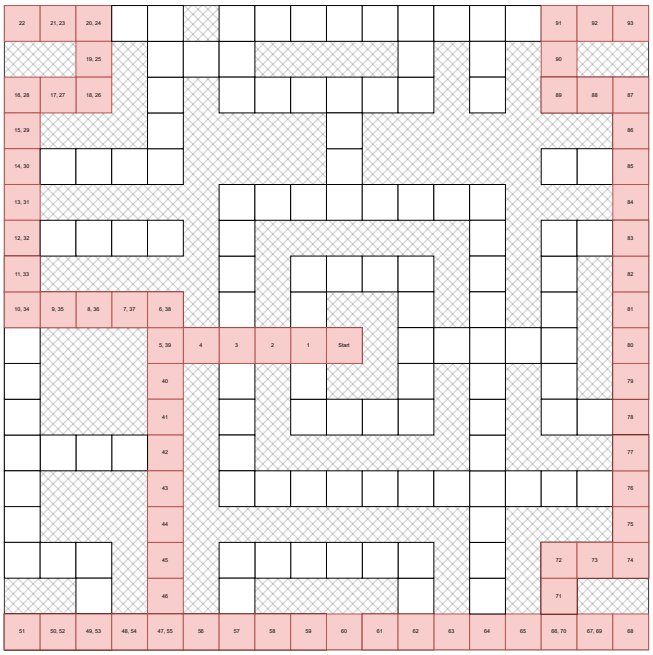


Fig. 3: Paths made by the Memory tabular Q-learning on the specified mazes.

devise a movement strategy to reach one exit, perform a return movement to the centre, and move to the exit at the other end of the corridor. 4 is the optimal accumulated step to reach both exits on the small corridor maze. It can be observed that the tabular Q-learning without memory accumulated 501 steps; this indicates that the agent did not find the second exit and ran out of time.

The accumulated total steps by the Q-learning with memory achieved better performance than other algorithm methods such as PPO, A2C and DQN, where the same memory architecture was used to help PPO, A2C and DQN algorithms.

The memory architecture we designed significantly helped tabular Q-learning training for $100,000$ iterations. It can be seen in table 2 that the tabular Q-learning with memory achieved 931.661 ± 0.678 average total steps to reach all exits in the complex looped maze, which shows that a maze as big as the complex looped maze requires more time to optimise.

Comparatively, PPO and A2C algorithms with our augmented memory performed better than those without external memory. However, DQN's performance suffered from our augmented memory method. The state representation $s_t = (y_t, x_t, \vec{c}_{(y_t,x_t)})$ added unparalleled access to different state representations due to $C(y_t, x_t, d_t)$ update method shown in (5).

The performance of PPO, A2C and DQN did not reach the optimal accumulated steps. It can be assumed that PPO, A2C and DQN needed further

Table 3: Performance of "wall-follower" algorithm learned by Q-learning at 100,000 episodes. The errors indicate the standard error over 10 trials.

Maze Name	A^* Accumulated Steps	Algorithm method	Average total of steps to reach all exits
Small Corridor	4	Wall-following Q-learning	4 ± 0.0
Long Corridor maze	30	Wall-following Q-learning	221 ± 0.0
T-shaped maze	16	Wall-following Q-learning	42 ± 0.0
Cross maze	32	Wall-following Q-learning	32 ± 0.0
Complex Looped maze	232	Wall-following Q-learning	1731 ± 0.0

adjustments, especially in their network structure, because these algorithms have proven to be sensitive to hyper-parameters [23].

Table 3 shows that the tabular Q-learning agent learned to perform like a "wall-follower" algorithm and solved perfect mazes. However, it can be seen that the algorithm struggled with the complex looped maze as expected. Comparatively, our Q-learning with an external memory solution performed better and reached all exits.

Moreover, suppose there are two potential exits for a maze. In that case, reaching the potentially closer exit is more efficient. Our augmented memory tabular Q-learning follows this rule, whereas the "wall-follower" behaviour does not prioritise reaching the potentially more immediate exit first. The state representation given to the "wall-following" algorithm reveals the solution to the Q-table, and it will solve any given perfect maze with no loops. However, the state representation given to tabular Q-learning with memory only reveals the agent's location and its neighbouring cells history of visit.

For the RL algorithms such as DQN, A2C and PPO, we attempted to hot-one-encode the observation state into a large flattened maze representation. Unfortunately, we did not get improved results compared to the state vector, including the position of the agent and the memory we designed.

7 Conclusion

In deep learning, recurrent neural networks are essential when each piece of information, through time, is needed to solve a problem. For example, in our implemented environment, where mazes require the agent to turn back after reaching a dead-end, the agent must have this information looped back to its algorithm to be able to perform this movement strategy.

Recurrent neural networks can be computationally expensive and difficult to train; other difficulties, such as gradient vanishing, can be faced while using recurrent neural networks. In addition, it may become tedious to adjust all the activation functions in processing long sequences.

In this paper, we presented memory augmentation as a straightforward way to add memory to work with maze-solving algorithms so that it rivals recurrent nodes without the difficulties of training them. Moreover, we showed that this simple state representation significantly benefited the tabular Q-learning algorithm, where it could perform better than the "wall-follower" generic maze solver.

It will be helpful to change deep reinforcement learning algorithms such as PPO, DQN and A2C network structure in the future and try out different update methods. It will be helpful to integrate the solution we implemented in this paper to solve other discrete space environments to understand the limitations and advantages of this memory-augmented method.

References

1. D. Osmanković and S. Konjicija, "Implementation of q—learning algorithm for solving maze problem," in 2011 proceedings of the 34th international convention MIPRO. IEEE, 2011, pp. 1619–1622.
2. C. J. Watkins and P. Dayan, "Q-learning," Machine learning, vol. 8, no. 3-4, pp. 279–292, 1992.
3. R. Ilin, R. Kozma, and P. J. Werbos, "Efficient learning in cellular simultaneous recurrent neural networks-the case of maze navigation problem," in 2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning. IEEE, 2007, pp. 324–329.
4. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," nature, vol. 323, no. 6088, pp. 533–536, 1986.
5. J.-H. Lin and J. S. Vitter, "A theory for memory-based learning," in Proceedings of the fifth annual workshop on Computational learning theory, 1992, pp. 103–115.
6. L. Cayton, "Fast nearest neighbor retrieval for bregman divergences," in Proceedings of the 25th international conference on Machine learning, 2008, pp. 112–119.
7. C. Alexopoulos, "A note on state-space decomposition methods for analyzing stochastic flow networks," IEEE Transactions on Reliability, vol. 44, no. 2, pp. 354–357, 1995.
8. F. Nielsen, "Hierarchical clustering," in Introduction to HPC with MPI for Data Science. Springer, 2016, pp. 195–211.
9. L. Meng, R. Gorbet, and D. Kulić, "Memory-based deep reinforcement learning for pomdps," in 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE, 2021, pp. 5619–5626.
10. ——, "Partial observability during drl for robot control," arXiv preprint arXiv:2209.04999, 2022.
11. H. Y. Teh, A. W. Kempa-Liehr, and K. I.-K. Wang, "Sensor data quality: A systematic review," Journal of Big Data, vol. 7, no. 1, pp. 1–49, 2020.
12. Z. Wu, X. Wang, J. E. Gonzalez, T. Goldstein, and L. S. Davis, "Ace: Adapting to changing environments for semantic segmentation," in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 2121–2130.
13. A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for q-learning in random stochastic mazes," in 2016 IEEE Symposium Series on Computational Intelligence (SSCI), 2016, pp. 1–8.

14. R. Dearden, N. Friedman, and S. Russell, "Bayesian q-learning," in Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, ser. AAAI '98/IAAI '98. USA: American Association for Artificial Intelligence, 1998, p. 761–768.
15. M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu, "Reinforcement learning with unsupervised auxiliary tasks," CoRR, vol. abs/1611.05397, 2016. [Online]. Available: http://arxiv.org/abs/1611.05397
16. D. Wierstra, A. Förster, J. Peters, and J. Schmidhuber, "Solving deep memory pomdps with recurrent policy gradients," vol. 4668, 09 2007, pp. 697–706.
17. C. Chen, V. Ying, and D. Laird, "Deep q-learning with recurrent neural networks," Stanford Cs229 Course Report, vol. 4, p. 3, 2016.
18. "Euler tour technique," Dec 2020. [Online]. Available: \url{https://en.wikipedia.org/wiki/Euler_tour_technique/}
19. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through deep reinforcement learning," nature, vol. 518, no. 7540, pp. 529–533, 2015.
20. J. R. B. Del Rosario, J. G. Sanidad, A. M. Lim, P. S. L. Uy, A. J. C. Bacar, M. A. D. Cai, and A. Z. A. Dubouzet, "Modelling and characterization of a maze-solving mobile robot using wall follower algorithm," in Applied Mechanics and Materials, vol. 446. Trans Tech Publ, 2014, pp. 1245–1249.
21. A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," Journal of Machine Learning Research, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html
22. S. J. Russell and P. Norvig, Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,, 2016.
23. M. Olsson, S. Malm, and K. Witt, "Evaluating the effects of hyperparameter optimization in vizdoom," 2022.