Reinforcement Learning for The Frozen Lake Problem and Variations

ME5406: Deep Learning for Robotics

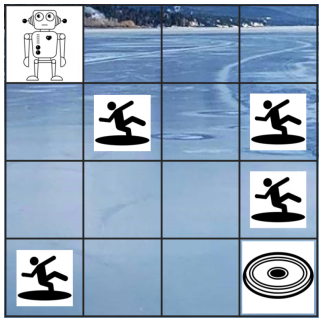
Chong Yu Quan

A0136286Y
e0321496@u.nus.edu
Faculty of Engineering
National University of Singapore
27 September 2020

1 Introduction

This report highlights the implementation of reinforcement learning (RL) techniques in solving a small scale grid world problem, called the Frozen Lake Problem (FLP), which can be further varied.

1.1 Frozen Lake Problem and Variations



(a) Illustration

```
time step: 0
r _ _ _
_ h _ h
_ _ _ h
h _ _ f
```

(b) Code implementation

Figure 1: The Frozen Lake Problem

Figure 1a illustrates the basic FLP. It is a 4 by 4 grid (frozen lake) with 4 holes covered in thin ice. A robot is to glide from the top left corner to pick up a frisbee in the bottom right corner. Figure 1b illustrates the code implementation of the grid world, where 'r', 'h', 'f', '_', represents the robot, hole, frisbee and empty state respectively. The robot is confined within the grid and given any state, it can only choose one of four directions (up, down, left, right) to move. When the robot reaches the frisbee, hole or empty state, it receives a reward of 1, -1 or 0 respectively. Lastly, an episode terminates terminates when the robot reaches a hole or the frisbee. Variations of the problem include extending the grid size while maintaining the same hole to state ratio of 0.25, distributing the holes evenly without blocking access to frisbee for the robot. An example of a 10 by 10 grid is shown in Figure 2 below.

```
time step: 0
r _ _ _ h _ _ _ _ _
_ h _ _ _ _ _ _ _ _
_ _ h h _ _ _ h h _
_ _ h h h _ _ _ _ _
h _ h _ _ _ h _ _ _
_ h h _ _ _ _ _ _ h
h _ _ h h _ _ _ _ _
_ _ _ _ _ _ _ _ h _
_ _ _ _ h h _ _ _ _
_ _ h h _ _ h h _ f
```

Figure 2: Code implementation of 10 x 10 frozen lake problem

2 Code Implementation

2.1 Grid

The grid is implemented using a Python class to initialises a class instance. The class instance contains all the necessary attributes to describe the grid (e.g. grid size, time step and location of robot, holes, frisbee) at any given state of an episode. Furthermore, the class instance also contains variables (e.g. discount rate, learning rate) and data structures that logs state, action and reward from an episode and store Q values to implement RL. Most importantly, there are useful class functions that allows for the implementation of an episode till termination. For example, the function add_valid_holes() randomly initialises and stores the location of holes in the grid, for variations of the FLP, in accordance to the hole to state ratio of 0.25 while ensuring that the robot is not denied access to the frisbee. The class function move() implements the robot's movement in the grid and raise Boolean flags when the robot reaches the a hole or the frisbee to indicate the termination of an episode. Other class functions include show_grid_state(), which prints the grid state with its corresponding time step for human readability and debugging as shown in the introduction.

2.2 Policy

An important function amongst class functions would be the policy function, which decides the action of the robot for a given state. The two policies utilised in this report are the greedy policy and ϵ-greedy policy.

$$A^* = argmax(q_\pi(s,a)) \tag{1}$$

For a greedy policy (π), the action that gives the largest Q value for a state-action pair, $q_\pi(s,a)$, (i.e. greedy action A^*) is always selected for a given state, as shown in equation 1 above. In the situation where there are multiple actions that gives the largest Q value, an action will be randomly chosen amongst the best actions.

$$\pi(s|a) = \begin{cases} 1-\epsilon+\frac{\epsilon}{|A(s)|} & \text{if } a = A^* \\ \frac{\epsilon}{|A(s)|} & \text{if } a \neq A^* \end{cases} \tag{2}$$

For an ϵ-greedy policy (π), which belongs to the class of ϵ-soft policies, each action at a given state has at least $\frac{\epsilon}{|A(s)|}$ chance of being selected, where ϵ is a parameter and $|A(s)|$ is the number of actions for a given state. This can be done by assigning $\frac{\epsilon}{|A(s)|}$ probability to each non-greedy action and the remaining probability to the greedy action, A^*, as shown in equation 2. In the situation where there are multiple actions that gives the largest Q value, an action amongst the best actions will be randomly given the higher probability. For both policies, the class function is_move_valid() is utilised to ensures that selected action is valid, i.e does not take it robot of the grid, by removing those actions from the list for selection.

2.3 Key Algorithms

The 3 key algorithms investigated are the First Visit Monte Carlo without Exploring Starts (FVMCES), Sarsa and Q Learning algorithms. The following sections further elaborates on the algorithm and their implementation.

2.3.1 First Visit Monte Carlo without Exploring Starts

Monte Carlo (MC) control method is a model-free method for finding an optimal policy from Q values by random sampling. It learns directly from complete episodic experiences, implying that all episodes must terminate.

Paremeter: A small $\epsilon > 0$
Initialize: An arbitrarily ϵ-soft policy π
- $Q(s,a) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
- Returns(s,a) as an empty list, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
- Generate an episode with T steps following π:
  - $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
- $G \leftarrow 0$
- Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
  - $G \leftarrow \gamma G + R_{t+1}$
  - Unless pair (S_t, A_t) appears in $S_0, A_0, S_1, A_1, \ldots, S_{T-1}, A_{T-1}$:
    - Append G to Return(S_t, A_t)
    - $Q(S_t, A_t) \leftarrow$ average(Return(S_t, At))
    - $A^* \leftarrow \text{argmax}_a Q(S_t, a)$
    - For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1-\epsilon+\frac{\epsilon}{|\mathcal{A}(S_t)|} & \text{if } a = A^* \\ \frac{\epsilon}{|\mathcal{A}(S_t)|} & \text{if } a \neq A^* \end{cases}$$

Figure 3: Pseudo code implementation of First Visit Monte Carlo without Exploring Starts

Figure 3 highlights the algorithm for FVMCES. The class constructor initialises all necessary instance attributes, including epsilon (ϵ), an array of Q values for every state-action pair initialised to zero (Q values array) and empty array to log returns for every state-action pair for every episode (G_{all}). In an iteration, an episode following the ϵ-greedy policy is generated till termination. All states, actions and rewards are logged in lists (S_{log}, A_{log}, R_{log} respectively) indexed by time step. Two arrays logging the returns are initialised with zeros, indexed by state-action pair ($G_{s,a}$) and time step ($G_{time\ step}$) respectively. Looping from the terminal time step till initial time step (0), the returns are calculated using R_{log} and $G_{time\ step}$.

$$G_t = R_{t+1} + \gamma G_{t+1} \tag{3}$$

Equation 3 above shows the calculation, where G_t is the return at time step t, R_t is the reward at time step t and γ is the discount rate. By checking S_{log} and A_{log} in earlier time steps, the algorithm determines if the state-action pair at the current time step is the first visit in the whole episode. Only when a state-action pair

is first visited then its return is appended to $G_{s,a}$. At the end of loop, $G_{s,a}$ is appended to G_{all}. Finally, the mean of G_{all} between all logged episodes, ignoring the zero values using masking, gives the new Q values array and the grid resets to its initial state for the next iteration. Theoretically, the process is iterated forever for the best approximation of optimal policy. However, given limited time and computational resources, the number of iterations will be a finite parameter to be adjusted.

2.3.2 Sarsa and Q Learning

Unlike MC control methods that updates Q values per complete episode, Temporal Difference (TD) control methods update the Q values after one or more time steps within an episode using an biased estimate of the optimal Q values, a process known as bootstrapping shown in equation 4 below. $\delta_t = TD\ target - current\ value$, is also known as TD error.

$$updated\ value \leftarrow current\ value + learning\ rate \times (TD\ target - current\ value) \tag{4}$$

Hence, TD control methods are able to learn from incomplete sequences and work in non-terminating environments. TD control methods where the TD target is based on a one time step look ahead are referred to as TD(0) control methods.

Table 5.2: SARSA for estimating $Q \approx q_*$.

Parameters:	Step size $\alpha \in (0,1]$, and small $\epsilon > 0$
Initialize:	$Q(s,a)$ for all $s \in \mathcal{S} \cup \hat{\mathcal{S}}$ and $a \in \mathcal{A}(s)$, with $\hat{\mathcal{S}}$ being the set of terminal states; all initial values are arbitrary except $Q(s,a) = 0$ for all $s \in \hat{\mathcal{S}}$
	Loop for each episode:
	Initialize S
	Choose A from S using policy derived from Q (e.g., ϵ-greedy)
	Loop for each step of episode:
	Take action A; observe R, S'
	Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
	$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
	$S \leftarrow S'$; $A \leftarrow A'$
	until $S \in \hat{\mathcal{S}}$

(a) Sarsa

Table 5.3: Q-learning for estimating $\pi \approx \pi_*$.

Parameters:	Step size $\alpha \in (0,1]$, and small $\epsilon > 0$
Initialize:	$Q(s,a)$ for all $s \in \mathcal{S} \cup \hat{\mathcal{S}}$ and $a \in \mathcal{A}(s)$, with $\hat{\mathcal{S}}$ being the set of terminal states; all initial values are arbitrary except $Q(s,a) = 0$ for all $s \in \hat{\mathcal{S}}$
	Loop for each episode:
	Initialize S
	Choose A from S using policy derived from Q (e.g., ϵ-greedy)
	Loop for each step of episode:
	Take action A; observe R, S'
	$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_{a'} Q(S',a') - Q(S,A)\big]$
	$S \leftarrow S'$
	until $S \in \hat{\mathcal{S}}$

(b) Q Learning

Figure 4: Pseudo code implementation of Sarsa and Q Learning

Figure 4 highlights the algorithm for Sarsa and Q Learning respectively. Both algorithms are TD(0) control methods, but with different TD target. Like for FVMCES, the class constructor initialises all necessary instance attributes: Q values array, ϵ and the learning rate (α). In an iteration from initialisation, an action (A_t) is chosen using the ϵ-greedy policy from state (S_t) for a given time step. Using the move() class function, the robot takes A_t and ends up in another state (S_{t+1}) and receiving a reward (R_{t+1}) at the next time step. For Sarsa, another action (A_{t+1}) is chosen using the ϵ-greedy policy from S_{t+1}. The Q values array is then updated according to equation 5 shown below.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{5}$$

It can be observed that Sarsa uses the ϵ-greedy policy as the behaviour policy to generate the data samples for RL and follows an ϵ-greedy policy as the target policy when updating the Q values. Given that its behaviour policy is the same as its target policy, Sarsa is an on policy algorithm and will converge to the optimal Q values for its target policy giving rise to an optimal ϵ-greedy policy. On the other hand, the next action (A^*_{t+1}) selected from S_{t+1} is chosen using a greedy policy instead of the ϵ-greedy policy for Q Learning. Q Learning uses the ϵ-greedy policy as the behaviour policy to generate the data samples for RL but follows a greedy policy as the target policy. Given that its behaviour policy different its target policy, Q Learning is an off policy algorithm and will converge to the optimal Q values for its target policy, i.e the greedy policy, giving rise to an optimal greedy policy. The Q values array update for Q Learning follows equation 6 shown below.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A^*_{t+1}) - Q(S_t, A_t)) \tag{6}$$

For both Sarsa and Q Learning, $\alpha = \frac{1}{time\ step}$ and $\alpha = 1$ when $time\ step = 0$ in this specific implementation. This update process continues till the episode is terminated, where the grid resets to its initial state for the next iteration for a chosen number of iterations.

2.4 Additional Algorithms

Given that Sarsa and Q Learning are TD(0) control methods that are based on a one time step look ahead, TD control methods that look beyond one time step are explored and tested against TD(0) control methods.

2.4.1 Sarsa(λ)

A simple way forward would be to consider the n-step Q-return as the TD target as shown in equation 7 below.

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n}) \tag{7}$$

Hence the n-step Sarsa updates for Q values towards the n-step Q-return is a shown in equation 8 below. When n = 1, the previous one time step return from TD(0) control methods (e.g. Sarsa) is obtained. As n approaches infinity, the n-step Q-return converges to the conventional return (G_t) used in MC control methods.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(q_t^{(n)} - Q(S_t, A_t)) \tag{8}$$

However, instead of choosing a specific n-step Q-return, a weighted average across all time steps till termination could be used to combine all the n-step Q-return using the weight $(1-\lambda)\lambda^{n-1}$ as shown in Figure 5.

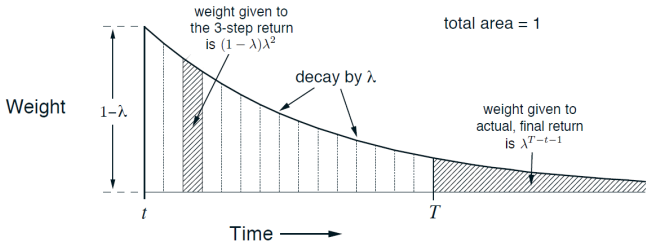


Figure 5: Weight given for each n-step Q-return

Such a particular method for averaging n-step Q-returns falls under the TD(λ) methods. This gives rise to the $q_t^{(\lambda)}$ return as shown in equation 9 below, where T is the terminal time step.

$$q_t^{(\lambda)} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)} = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} q_t^{(n)} + \lambda^{T-t-1} G_t \tag{9}$$

When $\lambda = 0$, $q_t^{(\lambda)} = G_t$, which is the conventional return in MC control methods. On the other hand, when $\lambda = 1$, $q_t^{(\lambda)} = q_t^{(1)}$, which is the one time step return from TD(0) control methods. Hence, λ can be used as a form of control between a continuum of MC to one step TD(0) (bootstrapping) returns. The forward view Sarsa(λ) update is obtained by applying the $q_t^{(\lambda)}$ return to forward view TD(λ) update methods as shown in equation 10 below.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(q_t^{(\lambda)} - Q(S_t, A_t)) \tag{10}$$

The update is forward view as it looks only into future rewards from the current state without considering the preceding states. However, given that the forward view Sarsa update requires complete episodes like MC, it loses the benefit of updating online at every time step from incomplete episodes that is originally present in the TD(0) control methods. Hence, the backward view TD(λ) is considered, which requires an additional memory variable called eligibility traces associated with every state-action pair. Eligibility traces records states that have "recently" been visited to indicate the degree each state-action pair is eligible for an update. Figure 6 illustrates the types of eligibility traces with their respective equations shown in equation 11 in identity indicator notation.

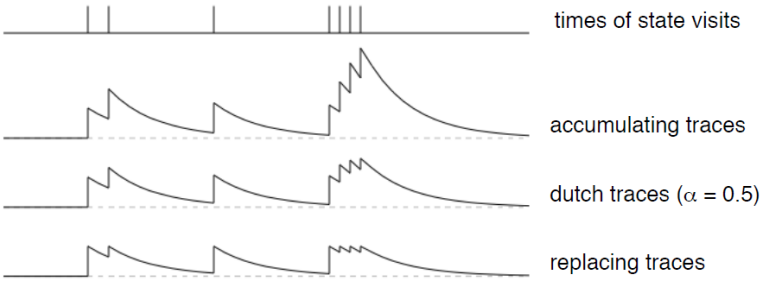


Figure 6: Types of eligibility traces

$$\begin{aligned} E_t(s,a) &= \gamma\lambda E_{t-1}(s,a) + I_{sS_t} I_{aA_t} \qquad (accumulating) \\ E_t(s,a) &= (1-\beta)\gamma\lambda E_{t-1}(s,a) + I_{sS_t} I_{aA_t} \qquad (dutch) \\ E_t(s,a) &= (1 - I_{sS_t} I_{aA_t})\gamma\lambda E_{t-1}(s,a) + I_{sS_t} I_{aA_t} \qquad (replacing) \end{aligned} \tag{11}$$

The accumulating traces adds up each time a state-action pair is visited while replacing traces are reset to one. Dutch traces functions as an in between depending on the value of β. Nevertheless, all the traces decay at the rate of $\gamma\lambda$, which defines the notion of "recently". Considering the one step TD error (δ_t) in equation 12 below, the Q value is updated for every state-action pair in proportion to δ_t and the $E_t(s,a)$ shown in equation 13 below.

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \tag{12}$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\delta_t E_t(s,a) \tag{13}$$

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
    $E(S, A) \leftarrow E(S, A) + 1$ (accumulating traces)
    or $E(S, A) \leftarrow (1 - \alpha)E(S, A) + 1$ (dutch traces)
    or $E(S, A) \leftarrow 1$ (replacing traces)
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
      $E(s,a) \leftarrow \gamma\lambda E(s,a)$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

Figure 7: Pseudo code implementation of Sarsa(λ) using eligibility traces

Figure 7 highlights the Sarsa(λ) algorithm. The class constructor initialises all necessary instance attributes: Q values array, ϵ, γ and λ. In an iteration, an eligibility trace array (E array) is first initialised to zero for every state-action pair. Till termination, A_t is chosen using the ϵ-greedy policy S_t for a given time step. Using the move() class function, the robot takes A_t and ends up in S_{t+1} and receives reward R_{t+1} at the next time step. A_{t+1} is then chosen using the ϵ-greedy policy from S_{t+1}. δ_t is then updated according to equation 12. E array for state-action pair (S_t, A_t) is also updated using accumulating traces for this report's implementation. Lastly, Q values for every state-action pair in the Q values array is updated using equation 5 and traces for every state-action pair in the E array is decayed by $\gamma\lambda$. Upon termination, the grid resets to its initial state for the next iteration for a chosen number of iterations.

2.4.2 Watkin's Q(λ)

Unlike forward view Sarsa(λ), the forward view Q(λ) does not look ahead to the end of the episode in its backup. Instead, it looks one action past the first exploratory action like Q Learning, unless the episode terminates before any exploratory actions. Given that A_{t+n} is the first exploratory action, δ_t is given in equation 14 below, where A^* is the greedy action.

$$\delta_t = R_{t+1} + \gamma G_{t+1} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A^*) \tag{14}$$

The backward view of Q(λ) utilises eligibility traces as well, as shown in Figure 8. The class constructor initialises all necessary instance attributes: Q values array, ϵ, γ and λ. In an iteration, the E array is first initialised to zero for every state-action pair. Till termination, A_t is chosen using the ϵ-greedy policy S_t for a given time step. Using the move() class function, the robot takes A_t and ends up in S_{t+1} and receives reward R_{t+1} at the next time step. A_{t+1} is then chosen using the ϵ-greedy policy and A^*_{t+1} is chosen using the greedy policy from S_{t+1}. δ_t is then updated according to equation 15.

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A^*_{t+1}) - Q(S_t, A_t) \tag{15}$$

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $A^* \leftarrow \arg\max_a Q(S',a)$ (if A' ties for the max, then $A^* \leftarrow A'$)
    $\delta \leftarrow R + \gamma Q(S', A^*) - Q(S,A)$
    $E(S,A) \leftarrow E(S,A) + 1$ (accumulating traces)
    or $E(S,A) \leftarrow (1-\alpha)E(S,A) + 1$ (dutch traces)
    or $E(S,A) \leftarrow 1$ (replacing traces)
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
      If $A' = A^*$, then $E(s,a) \leftarrow \gamma\lambda E(s,a)$
        else $E(s,a) \leftarrow 0$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

Figure 8: Pseudo code implementation of Q(λ) using eligibility traces

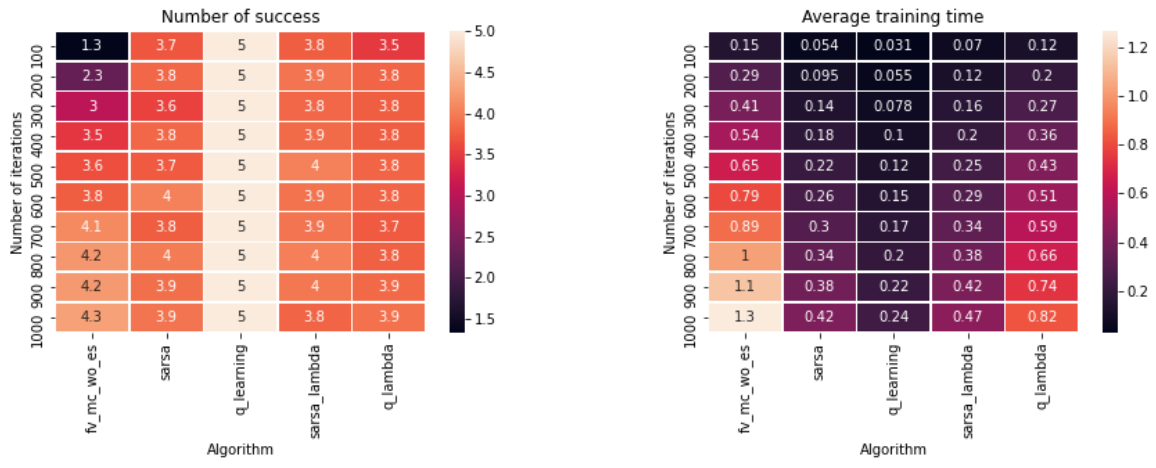
E array for state-action pair (S_t, A_t) is also updated using accumulating traces for this report's implementation. Q values for every state-action pair in the Q values array is updated using equation 5. However, traces in the E array is decayed by $\gamma\lambda$ only if the action in the state-action pair is A^*_{t+1}. Else, the trace is set to zero Upon termination, the grid resets to its initial state for the next iteration for a chosen number of iterations.

3 Results and Discussion

3.1 Basic Implementation

For the basic implementation, the following parameters are studied: ϵ, λ, discount rate, number of iterations and algorithm used. The range of values used are as follows: 1) ϵ: ranging from 0.1 to 0.3 (inclusive) with step size of 0.05, 2) λ: ranging from 0.8 to 0.95 (inclusive) with step size of 0.05, 3) discount rate: ranging from 0.5 to 0.9 (inclusive) with step size of 0.1, 4) number of iterations: ranging from 100 to 1000 (inclusive) with step size of 100. For each combination of the parameters, an agent is trained from scratch and implemented using a greedy policy (greedification) where the success (i.e. reaching the frisbee) and training time is logged. The episode is considered a fail if the robot doesnt not reach the frisbee within 1000 time steps as well, for computational time purposes. This process is repeated five times, where the number of success and average training time is recorded in two five dimensional arrays, success and time array respectively. Each axis of the array represents a parameter, where the axis length is the length of range of the parameter, hence dimensioned for every combination of parameters.

3.1.1 Number of Iterations vs Type of Algorithm



(a) Number of Success

(b) Average training time

Figure 9: Results for Number of Iterations vs Type of Algorithm

Figure 9 shows the heat maps of the number of success and average training time for the number of iterations against the algorithms used. The values for the heat map are obtained by averaging the values for the success and time array across the other parameter axis. From Figure 9a, FVMCES is a the poorest performing algorithm in terms of number of success on average for a low number of iterations between 100 and 300, with an average of 2.3 success out of 5 when the number of iterations is 200. This is due to the fact that FVMCES is an offline algorithm that learns from a terminated episode of experience generated based on the policy from the previous iteration. As a result, it cannot learn while the episode is generated, hence having the least optimal policy compared to other algorithms when the number of iterations is low. However, beyond a certain threshold of approximately 400 for the number of iterations , FVMCES obtains a similar number of success compared to most of other algorithms, such as Sarsa, Q-Learning and Q(λ), given that it has converged to an approximately optimal policy like the other algorithms. In fact, it can be observed that for larger number of iterations beyond the threshold between 700 to 1000, FVMCES actually has a slight edge over Sarsa, Sarsa(λ) and Q(λ) on average. A possible reason for this observation could be attributed to the fact that the target for MC learning is an unbiased estimate of the true optimal Q value as compared to the TD target, which is a biased estimate of the true optimal Q value. As a result, the MC control methods would have an edge over TD control methods in reaching the true optimal Q values, which would guarantee that it would reach a policy equally optimal to TD control methods if not more optimal, assuming that number of iterations are not a constraint. However, such an assumption is usually invalid in reality given that MC control methods in generally are highly inefficient in terms of training time as compared to other online algorithms as shown in Figure 9b. This is again due to the fact that a terminated episode has to be generated for each iteration before an update, exacerbated by a necessary search through the episode to check if a given state-action pair is a first visit for FVMCES.

In comparison to FVMCES, TD control methods are perform consistently well on average for all number of iterations. For example, Sarsa, Sarsa(λ) and Q(λ) have average number of success ranging between 3.5 to 4, even at low number of iterations. In particular, Q Learning excelled in this problem with an average number of success of 5 for all number of iterations, indicating it has performed optimally each repeated trial of 5 for all combinations of parameters. This highlights the benefits of online learning using TD control methods as it is able to converge to an approximately optimal, if not optimal, policy with less iterations compared to MC control methods.

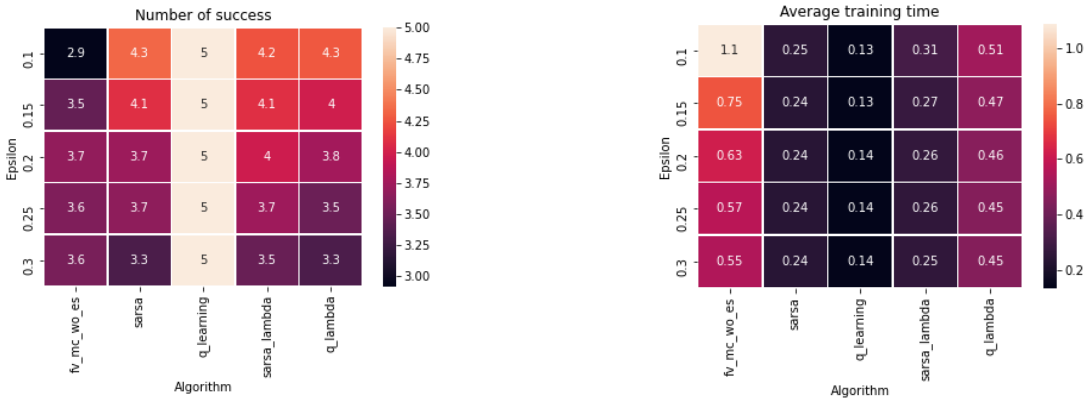
Comparing the number of success performance of on and off policy algorithms without eligibility traces, the stellar performance of Q Learning highlights the differences in performance between an optimal greedy policy (Q Learning) vs an approximately near optimal ϵ-greedy policy after greedification (Sarsa) on average. Considering the well-known Cliff Walking example and Figure 1, it can be observed that there are no 'safer' routes that the robot can take to minimize the risk of falling into a hole should it take an exploratory action. Hence, a possible explanation for Sarsa's relatively poorer performance could be due to the fact it is struggling to learn how to mitigate exploration risk given that there is no near optimal policy as a solution for the basic implementation of the FLP.

Comparing the number of success performance of on and off policy algorithms with eligibility traces, it can be observed that the performance of Sarsa(λ) and Q(λ) are approximately similar as opposed to the difference between their TD(0) counterparts. A possible explanation for such a discrepancy could be attributed to the implementation of the eligibility traces in Q(λ), where the eligibility traces are decayed only if the action follows the target policy (greedy policy) and reduced to zero for an exploratory action. However, given that the behaviour policy (ϵ-greedy policy) only takes the an exploratory action for $1 - \epsilon$ probability, the 'expected' deviation from the greedy policy is not of a degree to reduce the eligibility trace to zero. As a result, by crudely reducing eligibility traces to zero for an exploratory action, Q(λ) could have lost some of its efficacy. A larger eligibility trace decay (e.g. $\epsilon\gamma\lambda$), as compared to $\gamma\lambda$ for the greedy action, could perhaps be implemented for the exploratory action instead.

Moreover, the the number of success performance of Sarsa(λ) and Q(λ) is comparable to Sarsa, highlighting that the eligibility trace methods did not reap any significant learning benefit beyond its one-step counterpart on average for the basic implementation of the FLP. This is further corroborated by the fact that the average training time for Sarsa and Sarsa(λ) are approximately the same as shown in Figure 9b. Furthermore, while Q Learning is the most training time efficient algorithm for all number of iterations, the training time for Q(λ) is the highest amongst TD control methods and appears to increase at a higher rate compared to its counterparts. This is due to the fact that while eligibility traces offer significantly faster learning, especially when rewards are delayed by many time steps, it requires more computation than TD(0) control methods. Sarsa and Sarsa(λ) only achieved similar time performance in code implementation due to the fact that constant decay of $\gamma\lambda$ for all eligibility traces can be vectorised using arrays and made efficient as compared to a for loop iteration through all state-action pairs as stated in the pseudocode from Figure 7. On the other hand, the reduction of eligibility traces to zero for exploratory actions in Q(λ) required at least a masking array for a vectorised implementation, resulting in larger computation time. Hence, for the case of the basic implementation of the FLP, where the simulation is inexpensive, the computational cost for eligibility traces does not pay given that the objective is

for the algorithm to process through as much data as possible instead of getting more out of a limited dataset.

3.1.2 ϵ vs Type of Algorithm



(a) Number of Success

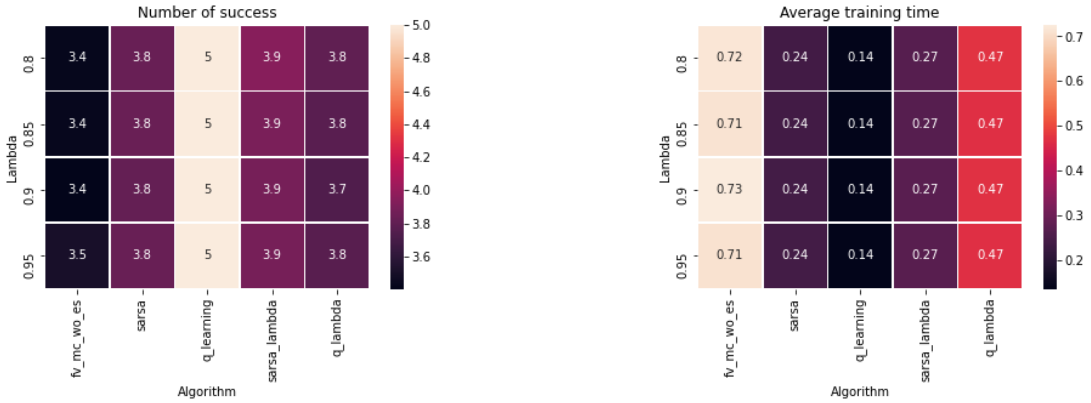
(b) Average training time

Figure 10: Results for ϵ vs Type of Algorithm

Figure 10 shows the heat maps of the number of success and average training time for ϵ against the algorithms used. The values for the heat map are obtained by averaging the values for the success and time array across the other parameter axis. From Figure 10a, it can observed that for TD control methods such as Sarsa, Sarsa(λ) and Q(λ), performance generally decreases on average with increasing ϵ with the exception of Q-Learning. As increasing ϵ enhances exploration risk, in which Sarsa and Sarsa(λ) recognises given their ϵ-greedy policy target policy, they struggle to mitigate it due to the fact that there is no near optimal policy as a solution for the basic implementation of the FLP, hence deteriorating learning and consequently performance. For Q(λ), increasing exploration would further undermine eligibility traces given that they are reduced to zero for any exploratory action, reducing the average length and consequently, the efficacy of the back ups. The robustness of Q Learning in converging efficiently to an optimal greedy policy as the target policy is apparent given its insensitivity to the values of ϵ.

On the other hand, increasing exploration actually improves the performance of FVMCES, a MC control method that is offline. This is due to the fact that increasing exploration broadens the range of experiences from terminated episodes and the episode length that are logged by FVMCES for learning. This is especially valuable for offline learning given that any specific episode follows a fixed policy till termination before any updates, hence making learning more efficient. This is corroborated with the decreasing average training time with increasing number of iterations for FVMCES as shown in Figure 10b, while the average training time for the rest of the algorithms remains largely constant.

3.1.3 λ vs Type of Algorithm



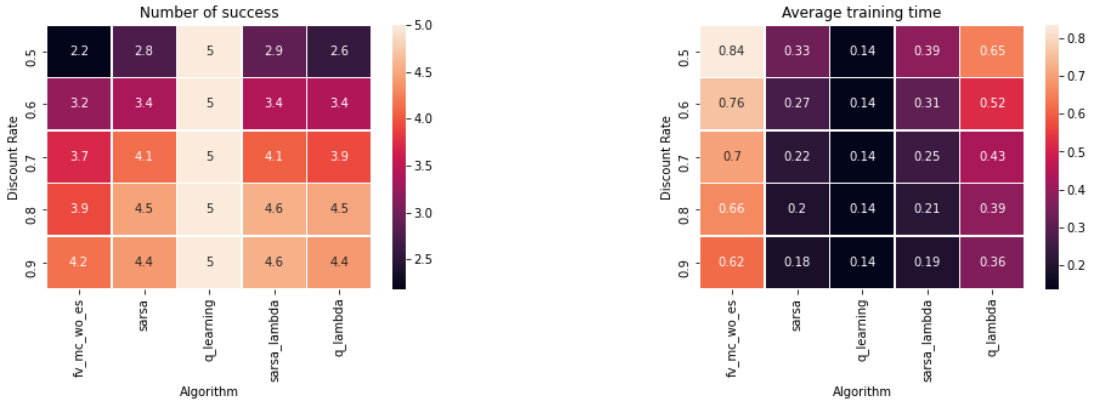
(a) Number of Success

(b) Average training time

Figure 11: Results for λ vs Type of Algorithm

Figure 11 shows the heat maps of the number of success and average training time for ϵ against the algorithms used. The values for the heat map are obtained by averaging the values for the success and time array across the other parameter axis. Only applicable to Sarsa(λ) and Q(λ), it can be observed that variations of the λ parameter does not have a significant effect on performance and training time on average. Given that the performance and training time are not sensitive to the relatively narrow scope of the λ parameters utilised, a broader range can be implemented in future studies to probe the effects of the λ parameter. In addition, it is also worth noting that while the λ parameter serves a control for the extent of bootstrapping for Sarsa(λ), the same cannot be said for Q(λ) given that bootstrapping occurs regardless of the value of λ, even when $\lambda = 1$.

3.2 Discount Rate vs Type of Algorithm



(a) Number of Success

(b) Average training time

Figure 12: Results for Discount Rate vs Type of Algorithm

Figure 12 shows the heat maps of the number of success and average training time for ϵ against the algorithms used. The values for the heat map are obtained by averaging the values for the success and time array across the other parameter axis. From Figure 12a and 12b, it can be observed that performance and training time on average increases with increasing discount rate, with the exception of Q Learning. A possible explanation would be the fact if discount rate is too small, given a reward structure that rewards with 1 or -1 upon termination and 0 otherwise, the Q value update could be too small for convergence to an optimal policy. Nevertheless, the robustness of Q Learning is apparent given its insensitivity to the discount rates for the basic FLP.

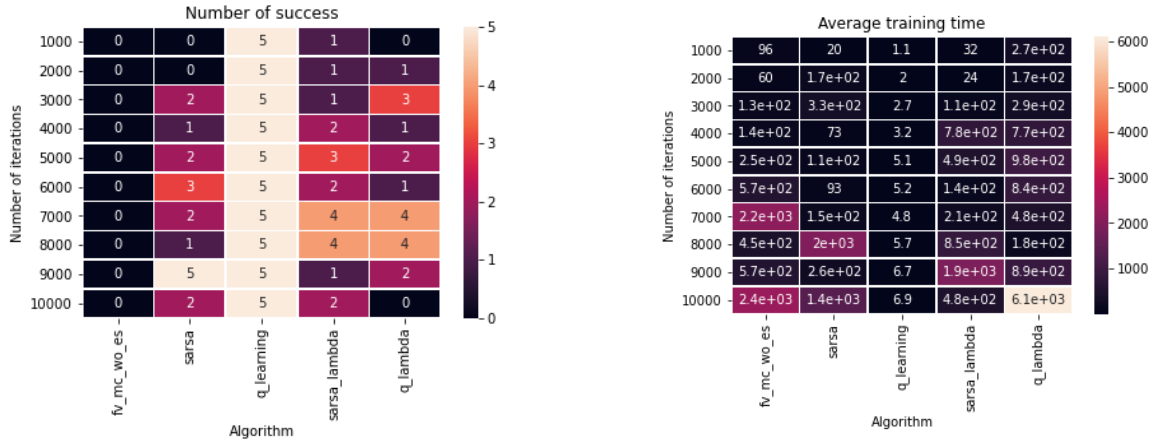
4 Extended Implementation

For the extended implementation, the following parameters are studied: number of iterations and algorithm used. ϵ, λ, discount rate are not studied given that significant computational cost expected for the extended implementation. The values for the number of iterations ranges from 1000 to 10000 (inclusive) with step size of 1000. The values used for ϵ, λ, discount rate are 0.1, 0.9 and 0.9 respectively, which are determined to be the optimal parameter values from the studies in the basic FLP. Most importantly, a training timeout of 10000 time steps for all algorithms in each iteration was implemented for the extended implementation. This is due to the observation that some of the algorithms (e.g. Q(λ)) have a propensity to reach a policy where the robot wanders the grid for an unacceptably long time for purposes of training (e.g. half a day) before termination for a reasonable number of iterations. The remaining implementation is identical to that of the basic implementation, other than the alterations of the success and time array to three dimensional arrays.

4.0.1 Number of Iterations vs Type of Algorithm

Figure 13 shows the heat maps of the number of success and average training time for the number of iterations against the algorithms used. From Figure 13a, it can be immediately observed that FVMCES is the poorest performing algorithm in terms of number of success, with zero success for all number of iterations. The drawbacks of FVMCES being an offline algorithm is blatantly apparent now, as stated previously with regards to the basic FLP. Given a considerably larger state-action space ($4 \times 4 \times 4 = 64$ vs $10 \times 10 \times 4 = 400$) in the extended FLP, FVMCES would require a lot more iterations than before for in the basic FLP, as well as relative to the other online TD control methods algorithms, to converge to an optimal policy.

Relative to FVMCES, the performance of Sarsa, Sarsa(λ) and Q(λ) is comparably better, with at least 1 or more success for most number of iterations. However, it is hard to find any distinguishing factors differentiating



(a) Number of Success

(b) Average training time

Figure 13: Results for Number of Iterations vs Type of Algorithm for grid size of 10

the the 3 algorithms from each other as their performance appears to be rather erratic. The number of success does not appear to generally increase with increasing number of iterations, for the range of number of iterations used. This is corroborated by an evident decrease in performance at higher number of iterations from all 3 algorithms from Figure 13a, albeit glimpses of good performance at smaller iterations that are likely be flukes. Hence, it can inferred that the performance of the algorithm is highly sensitive to the random initialisation of the grid (e.g. position of the holes) and the randomness in exploration when following the ϵ-greedy policy as the behaviour policy. This inference is supported by the seemingly large variance in training time from the each trial out of 5. For example, at 10000 number of iterations, the shortest training time for Sarsa for one trial is 13 seconds while the largest training time is 5375 seconds (approximately 1.5 hours), an approximately 400-fold difference. Hence, given the inherent randomness that significantly influences the convergence to an optimal policy, it can be reasoned that a greater number of iterations is required for all 3 algorithms in order to observe more consistent performance.

On the other hand, the performance of Q Learning remains impeccable, with 5 success out of 5 for all number of iterations. Hence, it can be inferred that the robustness of Q Learning applies to the extended implementation of the FLP, for reasons emphasised in the previous section. From Figure 13b, the time performance of the Q Learning is similarly stellar relative to other algorithms, requiring only a few seconds of training time on average to converge to the optimal policy thus far. Meanwhile, the other algorithms take significantly more training time compared to Q Learning, ranging from a few minutes to more than an hour on average with increasing number of iterations. Furthermore, the average training time for these remaining algorithms could very well be much longer, without the implementation of the timeout in training as stated previously. In particular, Q(λ), with eligibility traces implementations that involves cutting off eligibility traces for exploratory actions, takes significantly more training time with increasing number of iterations as compared to algorithms without eligibility traces (e.g. Sarsa) or eligibility traces methods that are easily vectorised (e.g. Sarsa(λ)). From debugging, it is observed that Q(λ) can take up to 10 hours of training time prior to the implementation of a timeout.

5 Conclusion

From the results, the best algorithm for the FLP, both basic and extended, is the Q Learning algorithm. The strength of an online, off policy algorithm with an optimal greedy policy as target policy is evident from the performance of Q Learning for the FLP. Furthermore, it can concluded that eligibility traces methods are not desirable for the FLP, given little performance gains from Sarsa(λ) and Q(λ), with increased computational costs, particularly from Q(λ).

Future studies could involve altering the reward structure of the agent to train an optimal policy that guides the robot to the frisbee with the shortest possible path. This could perhaps be achieved by altering the reward structure to give a negative reward (e.g. -0.1), rather than zero, every time the robot lands in an empty state in an episode, hence giving incentive for the robot to reach the frisbee as soon as possible. This alteration in the reward structure may also resolve the issue where the robot wanders aimlessly in the grid for an unacceptably long time given that aimless wandering is now penalised. Hence, it may be possible that a training timeout may not be required given such a change in the reward structure.