Various Reinforcement Learning Algorithms to Solve Grid World

Matteo Esposito[a], William Ngo[a], Spyros Orfanos[a]

[a] *Concordia University, Montreal, Quebec*

Abstract

Reinforcement Learning (RL) is widely used in different research areas to tackle problems such as resource management in computers[1], traffic light control[1], and playing games like chess[2] at a super-human level. In this project, we tackle the problem of solving a two-dimensional maze given a specific starting point. To accomplish this, three different policies were devel- oped using RL algorithms known as SARSA, Q-Learning, and n-step SARSA. Each learning method has its theoretical tradeoffs from one another and thus we are comparing their per- formances by evaluating their average training time per single run, average number of episodes until they reach optimality, and scalability by testing the algorithms on 2 different sized mazes. We conclude that Q-Learning is the best suited approach for Grid World since in both of our mazes, we observed consistently faster convergence rates to the optimal return and quicker per single run execution times.

1. Introduction

Our project, commonly referred to as "Grid World", tackles the problem of exiting a two-dimensional maze. This problem consists of a starting point, an exit point, and a board in which the agent (player) can take actions to move up, down, left or right. This is quite a simple task for a human as one can simply inspect the entirety of the maze and, at a glance, deduce an exit strategy. However, a machine cannot readily "see" the maze or "think" of actions to take, so how can it learn to solve a maze?

To allow a machine to "understand" the problem at hand, we must first reformulate the problem into something it can understand: a sequential decision problem. A sequential decision problem consists of a set of states, a set of actions, and a reward for taking some action given the current state.

By formulating the problem in such a way, we may use Reinforcement Learning (RL) techniques. RL is learning what to do: how to map situations to actions so as to maximize some notion of cumulative reward. In other words, RL is all about discovering the best action given your current state. To do this, the objective is designed as a numerical reward by learning from an environment. Most reinforcement learning problems can be expressed

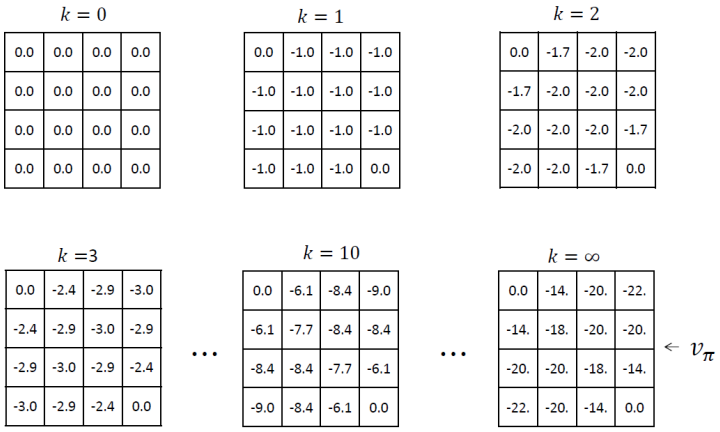
Email addresses: ds.mesposito@gmail.com, ID:40024121 (Matteo Esposito), ngowilliam96@gmail.com, ID:40031586 (William Ngo), spyros.orfanos1@gmail.com, ID:40032280 (Spyros Orfanos)

Preprint submitted to Dr.Frederic Godin, Concordia University *April 13, 2019*

as sequential decision problems, as they must include a set of the possible states of the environment, a set of possible actions and an objective. [3]

The goal of this project is to compare the rate of convergence of the estimated value function to the optimal policy's value function of three RL methods: SARSA, Q-Learning, and n-step SARSA.

Figure 1: Gridworld problem iterative policy evaluation. Here, v_π is the state-value function, k is the number of iterations. The figure depicts the convergence of the policy evaluation as $k \to \infty$.



1.1. Review of Literature

Given that the issue of optimal path finding/maze solving is a widely applicable problem, various machine learning, notably, reinforcement learning approaches have been taken in an effort to improve and/or complement current methods.

In Mnih et al. (2016), researchers from Google and MILA compared asnychronous variants of four standard reinforcement learning approaches. It was concluded that the asnychronous advantage actorcritic (A3C) method yields the most promising results. In an advantage actor-critic implementation, the actor is the policy, and the critic the value function. In practice, the agent will use the critic, or value estimate, to update the policy, or actor more efficiently than policy gradient methods. This method also introduces the concept of an advantage function, which is defined as $A(s,a) = Q(s,a) - V(s)$ where $Q(s,a)$ is the state value of s given action a and $V(s)$, the average value of state s. In practice, the discounted returns are used as an estimate of $Q(s,a)$.

Using this function versus discounted returns allows the agent to determine how much better its selected actions turned out to be than expected. This grants the algorithm the ability to focus on where the network's predictions were lacking in precision. By bypassing the use of a GPU, or any other specialized hardware, this method was able to outperform the state-of-the-art algorithms in Atari 2600 applications making use of a single multi-core CPU, and mastered a variety of 2D and 3D games, including the exploration of 3D mazes purely from visual inputs. It is considered the most general and successful reinforcement learning agent to date.[4]

In other articles such as Halici (2001), the use of random neural networks is proposed which differs from Q-Learning and temporal difference learning since it does not explicitly maintain estimates of reinforcements for each (state, action) pair. [5].

The motivation behind the study in Lei et al. (2018) was path planning for mobile robots. Here, the application of a double Q- network (DDQN) created by DeepMind is used for a robot's path planning in an unknown environment. Here, the optimal action value function Q in Q-Learning can be parameterized by an approximate value function $Q(s, a; \theta) \approx Q^*(s, a)$ where θ is the Q-network parameter. This approach is introduced to remedy the shortcomings of Q-Learning, namely lack of generalization ability and the fact that it requires the storing of a large amount of state-values among others. at hand.[6]

2. Problem Description

As described above, our problem, "Grid World", is a sequential decision problem as the set of states is the possible positions in the maze, the set of actions is deciding which direction to go and the objective would be to find the exit of the maze in the shortest amount of steps given a starting point and the exit point. Maybe add more here.

3. Analyses

3.1. Implementation - Code

We would like to briefly comment on the structure of our code. The project can be run entirely from `main.R`. The project has 2 dependencies, packages `R6` which grant object-oriented functionality and `rstudioapi`, in order to easily set the work directory. Object-oriented programming was a core concept in the design of our environment and our code as we wanted each script to do achieve specific task. By doing this, the script that generates the environment can be used for future tasks at hand even. In main.R all of our algorithms are called (by means of the `source()` function) and with them, average runtimes, optimal action plots and average return plots are output.

3.2. Implementation - Environment Spyros can you talk about the state mapping logic? $(x, y) \rightarrow$ real

Here, the set of states consists of all valid board positions (x, y) where $x \in \{1, 2, \ldots, nrow\}$ and $y \in \{1, 2, \ldots, ncol\}$. The set of actions consists of moving {Right $(action = 1)$, Down $(a = 2)$, Left $(a = 3)$ or Up $(a = 4)$} where the objective would be to find the exit of the maze.

To implement our maze, we create an n-by-n matrix that stores the values 0,1 and 10; these three values each map to valid spaces, wall and exit tiless. The starting point of the agent is defined as a variable named: "ratStartingPosition" defined in the main script. This was designed in a manner in which we could easily modify the characteristics of the maze by simply modifying one or two variables. Through the use of object-oriented programming and the R6 package, we can access and modify many properties of our maze object including

but not limited to: the possible actions at a given point in time, or a vector of the visited states.

Figure 2: General form of maze 1

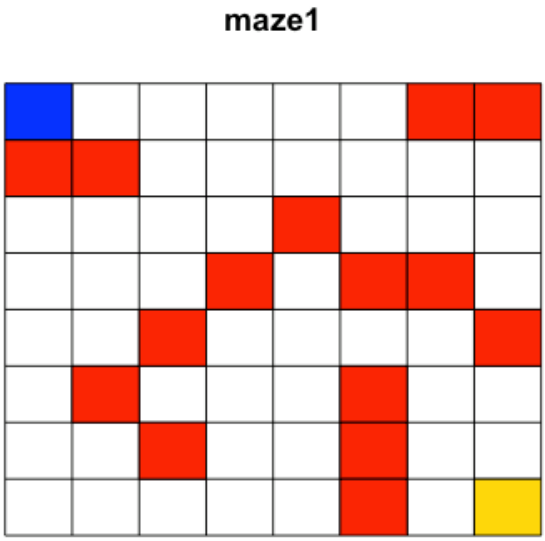
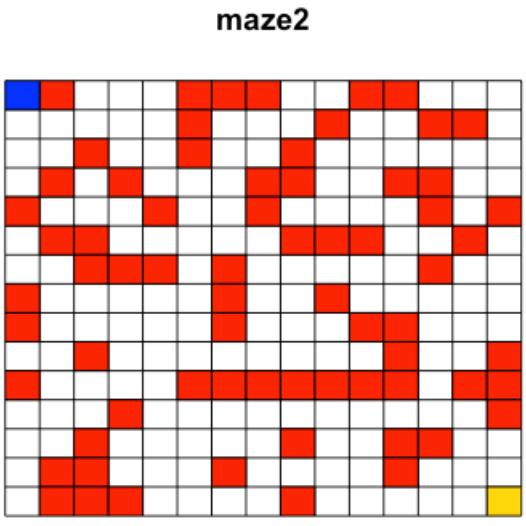


Figure 3: General form of maze 2



Note: All plots and future references represent runs that have been performed in the maze 1 environment unless stated otherwise.

For every action we assign a reward strictly dependent on the environment (set of states), with the following structure: Maybe you guys are better equipped to answer why we need

negative returns/logic we used, best I could come up with is that we want to penalize the agent from taking a large amount of steps...

- Reward attributed to hitting a wall - denoted as a red tile, or going beyond the border = -0.75
- Reward attributed to visiting a new, unseen tile that isn't the exit tile = -0.05
- Reward attributed to revisiting a previously seen tile = -0.30 Might want to change to -0.05 as per Spyros' most recent analysis (12:44AM April 13)
- Reward attributed to visiting the exit tile - denoted as a yellow tile = 5

One thing to note is that if the agent attempts to hit a wall or the outer bound of the maze then it stays at its current position and gets attributed a reward of -0.30. The design of our rewards should effectively train the agent to effectively exit the maze in the least amount of steps. Given the structure of our test maze, our optimal return would be $G_0 = 3.85$, which will be denoted as the orange convergence asymptote curve in our return average plots presented in the following section of this report.

3.2.1. Random Policy

Before running any learning algorithm, we evaluated the performance of a policy that selects actions at random. As expected, the policy performs extremely poorly as out of 10000 episodes,only 11% of the episodes were able to find the exit within less than 400 moves and for the episodes that ended, the average return is extremely poor when compared to the optimal return.

Table 1: Results of random policy after 10000 episodes

Number of episodes that ended	Average Return of the policies that ended
1075	-115.569

Using the random policy algorithm, out of 10000 episodes only 10.75% of all episodes exit the maze successfully/terminate within 400 steps, yielding an average reward of -115.596.

The above results illustrate how weak random policies are at solving grid world.

3.3. Algorithms Can condense & Build table of algo-update function. - In the meantime I condensed the descriptions a fair amount.

Here we list, the theoretical backgrounds and state value update functions of the 3 approaches we considered.

3.3.1. SARSA

Our second algorithm is the SARSA (State-Action-Reward-State-Action) method. The SARSA (State-Action-Reward-State-Action, one-step SARSA or SARSA(0)) method is an on-policy temporal difference control method. It enables the agent to update the (state,value) or $Q_\pi(s, a)$ value after every single action under a given policy π. Given an epsilon value, the agent will either tend to explore alternate actions in an effort to find different trajectories to the end goal or exploit the current optimal action at a given state ($\epsilon - greedy$).

Its update rule is defined as follows:

$$Q_{t+1}(S_t, A_t) \leftarrow Q_t(S_t, A_t) + \alpha[R_{t+1} + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t)]$$

3.3.2. Q-Learning

Q-Learning is an off-policy temporal difference control policy. It is exactly like SARSA, the only difference being that it doesn't follow a policy to find the next action A' but rather chooses the action in a greedy fashion. Here is its update rule.

$$Q_{t+1}(S_t, A_t) \leftarrow Q_t(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t)]$$

3.3.3. n-Step SARSA

Finally, The main idea of the n-step SARSA is to simply switch states for actions (state–action pairs) and then use an ϵ-greedy policy. Its algorithm is as follows:Validate this formula...maybe get an expression for Q update if necessary...

$$G_{t:t+n} = Q_{t-1}(S_t, A_t) + \sum_{k=t}^{\min(t+n,T)-1} \gamma^{k-t}[R_{k+1} + \gamma Q_k(S_{k+1}, A_{k+1}) - Q_{k-1}(S_k, A_k)]$$

3.4. Consideration: Bias vs. Variance Tradeoff

Another consideration of any machine leaning project is the bias-variance tradeoff. Since we are only dealing with TD (temporal difference) methods, we will only speak about the bias-variance present in these types of reinforcement leaning algorithms. Generally speaking, TD methods have lower variance but have higher bias compared to Monte-Carlo method. The lower variance is due to the fact that in the update equation for the value function of TD methods, the term/target value $R_{t+1} + \gamma V_\pi(S_{t+1})$ shows that the state-value update utilizes only the immediate future reward while in contrast to the Monte Carlo method, which have target value $G_t = R_{t+1} + \gamma R_{t+2} + \ldots$, which clearly depend on a greater number of rewards.

The presence of bias in the TD methods is due to the fact that the initial value function estimate is being used to update our value function estimate. It is worth noting that even while TD methods are bias, the bias is exponentially decaying thus asymptotically it is not an issue.

The concept of bias-variance tradeoff is a theoretical view of these learning methods but to know which learning method is best for a given problem we have to implement them and see which performs the best. Before comparing the different learning algorithms between each other, we first vary the "N" in N-Step SARSA to determine which one seems to be performs the best. We ran and compared 5,10 and 15 step SARSA implementations on our first 8x8 maze and observed the following results:

Figure 4: Comparison of n-Step SARSA algorithms by return G_0 at different n-values with exponentially decaying ε values.

From the following figure we can conclude that there isn't a significant discrepancy between 5-step and 10-step SARSA whereas 15-step SARSA is less suited to our problem considering all 3 convergence rates. 5-step or 10-step is clearly better for this problem. We decided to use 10-step SARSA in all future analyses/plots.

3.5. Considerations: Epsilon Selection

In approaching this problem, as in almost all reinforcement problems, choosing a suitable value of ε was a challenge, as we wanted to encourage the agent to use the highest return policy at any point in time as well as take time to explore alternate solutions to uncover a potentially better policy in the long run.

To address this challenge, we tested multiple epsilon update functions on one of our learning algorithms, and found that using an exponentially decaying epsilon yielded the most rapidly converging return curve, as seen in the following figure.

Figure 5: Q-Learning returns under various ε-update rules.

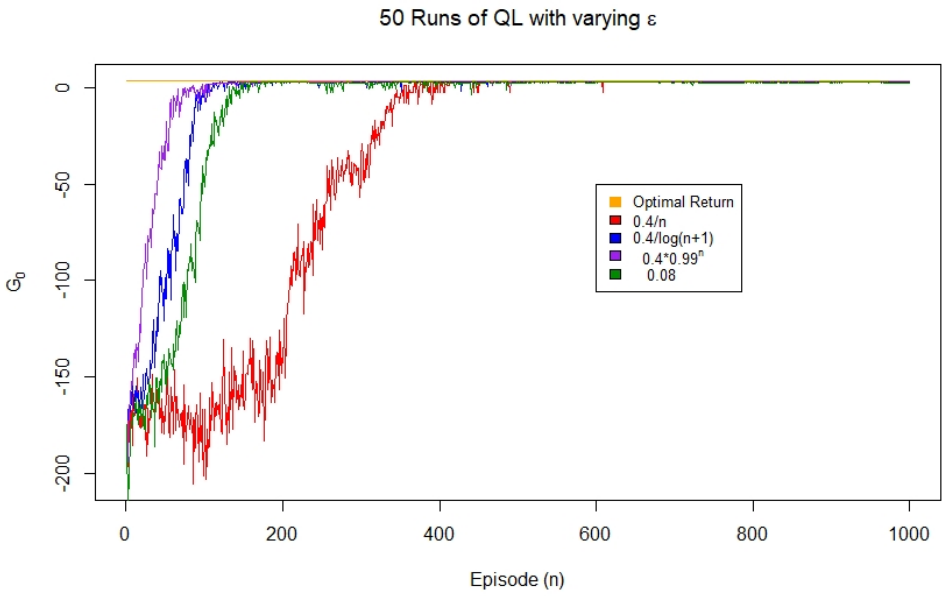


Table 2: Varying epsilon approaches runtime analysis

ε-function	Average # Episodes Before Convergence	Runtime (mins/ep)
$\frac{0.4}{n}$	~450	~0.189
$\frac{0.4}{\log(n+1)}$	~180	~0.087
$0.4(0.99)^n$	~120	~0.073
$0.4 = 0.08$	~200	~0.109

We observe that the average runtime per episode is proportional to the average number of the episodes before convergence which is expected since policy that was that was at- tributed to the update rule more quickly found the optimal policy and chose the most appropriate actions in the greedy case after converging.

Note: The reported runtime averages are volatile as they are hardware dependent.

3.6. Maze 1 Results

Figure 6: Maze 1 returns

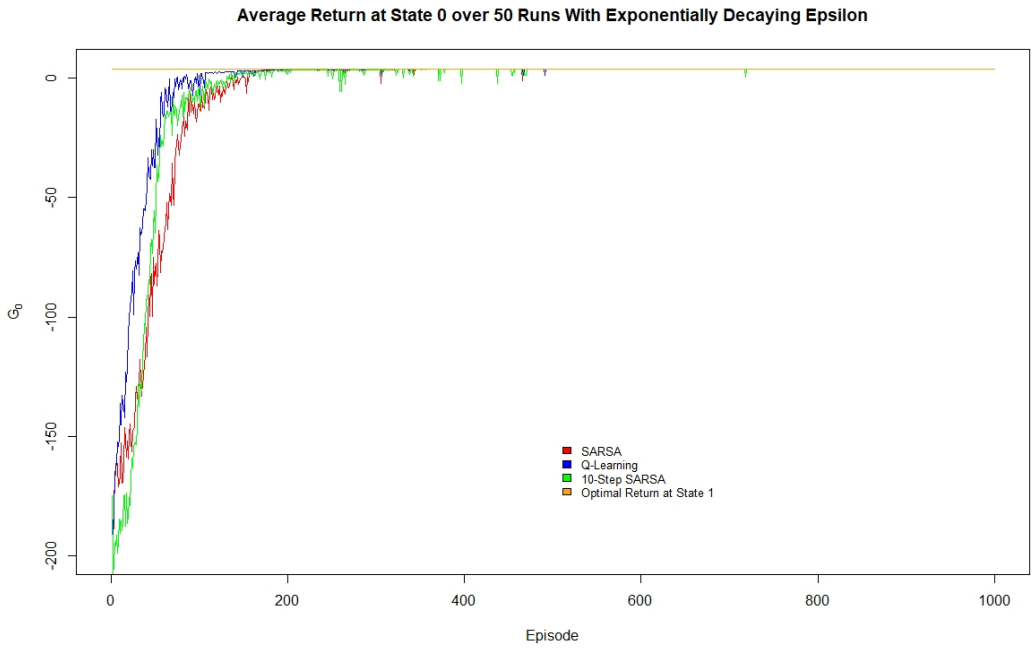


Table 3: Maze 1 runtime analysis

Algorithm	**Average # Episodes Before Convergence**	**Runtime (mins/run)**
SARSA	∼200	∼0.085
Q-Learning	∼150	∼0.075
n-Step SARSA	∼200	∼0.094

We observe that Q-Learning converges at a greater rate than SARSA and 10-step SARSA. It converges in roughly 3/4 the number of episodes and has a per episode runtime which is quicker than that of the SARSA algorithms. We can observe that the average runtime is proportional to the average number of episodes before convergence because past a certain point, when the agent selects the greedy action, it will select the optimal action and thus an episode would terminate quickly.

Figure 7: Optimal actions on maze 1 using SARSA

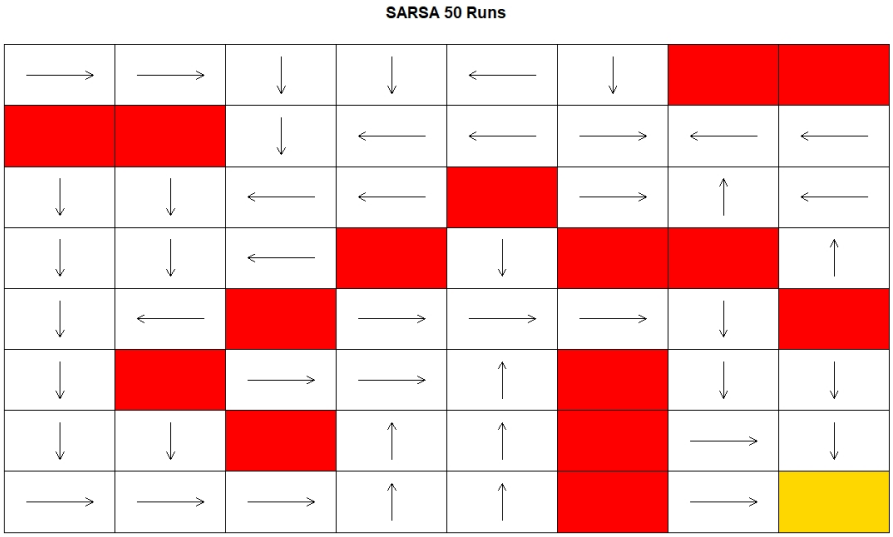


Figure 8: Optimal actions on maze 1 using Q-Learning

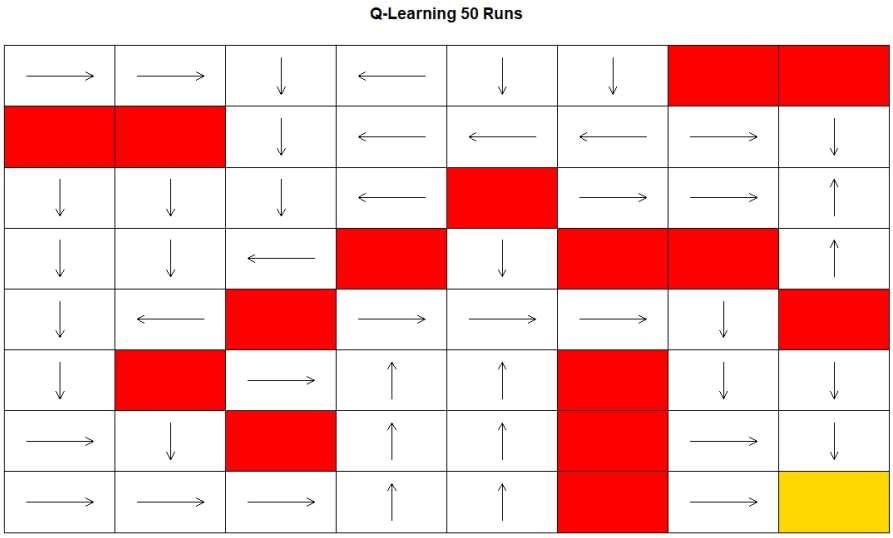
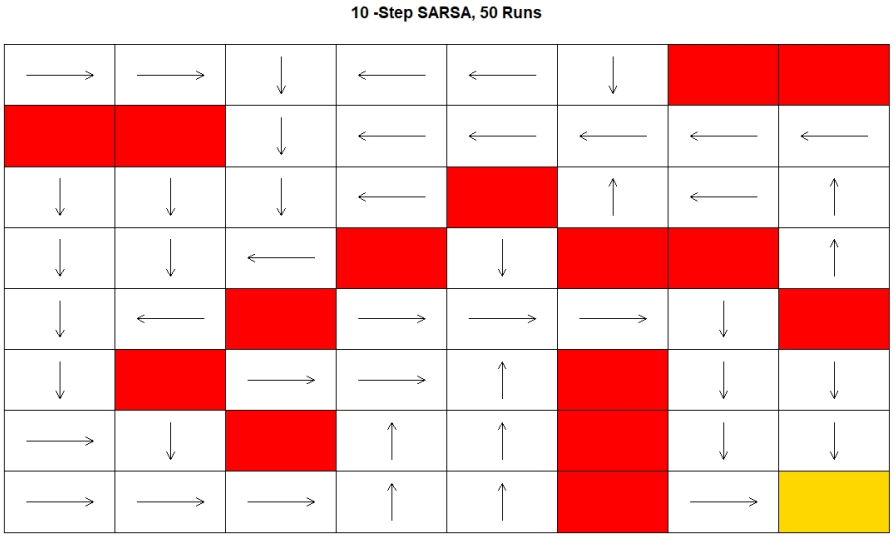


Figure 9: Optimal actions on maze 1 using 10-step SARSA



3.7. Maze 2 Results

Figure 10: Maze 2 returns

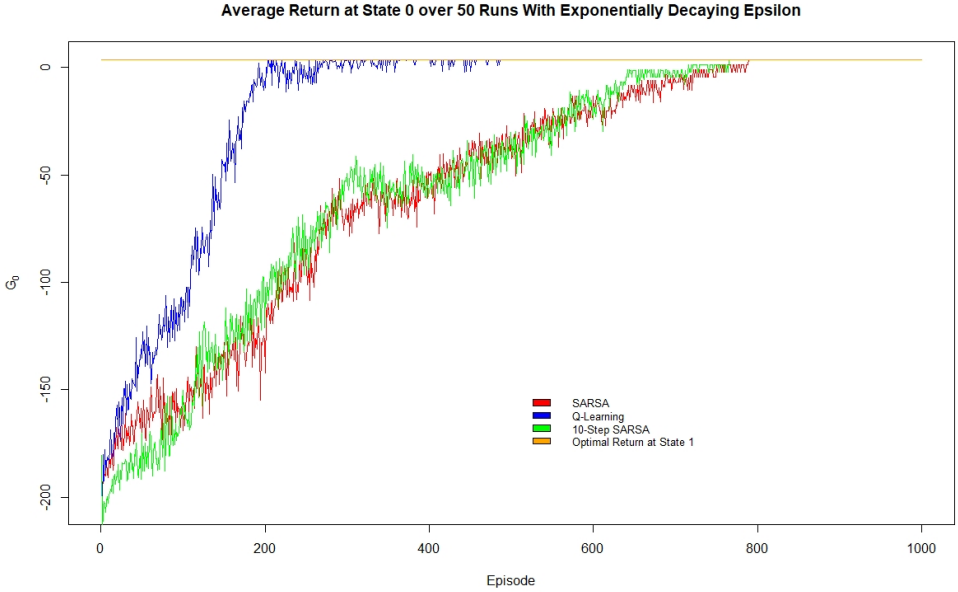


Table 4: Maze 2 runtime analysis

Algorithm	**Average # Episodes Before Convergence**	**Runtime (mins/run)**
SARSA	∼800	∼0.375
Q-Learning	∼400	∼0.275
n-Step SARSA	∼800	∼0.354

Similar to the plots in maze 1, we observe that Q-Learning converges at a much greater rate than SARSA and 10-step SARSA. It converges in roughly 1/2 the number of episodes and has a per episode runtime approximately 3/4 that of the SARSA algorithms. As expected, since we are dealing with a much larger grid relative to maze 1, there is a much larger difference in results between the optimal and sub-optimal algorithms.

In the maze 2 example we are dealing with a $15x15$ grid and therefore a larger discrepancy is to be expected as since there are more states and episodes last longer, the different learning algorithms have a larger room to distinguish themselves from one another.

Figure 11: Optimal actions on maze 2 using SARSA

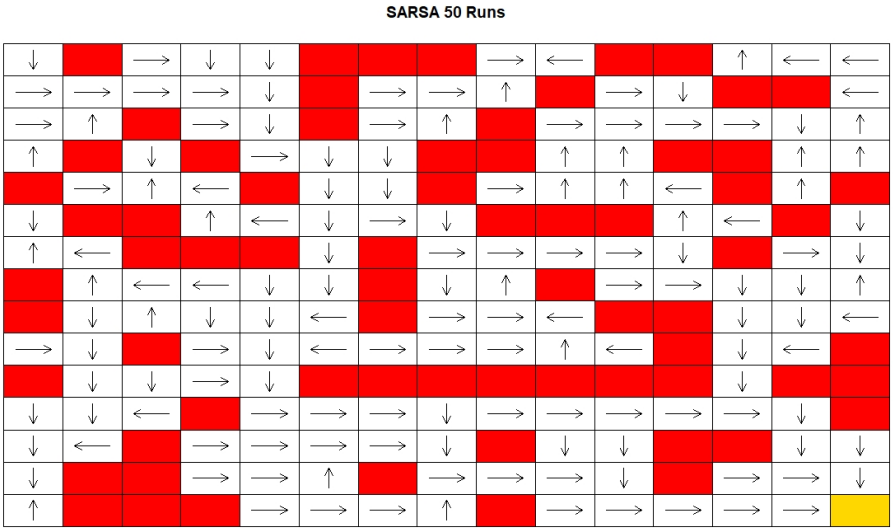


Figure 12: Optimal actions on maze 2 using Q-Learning

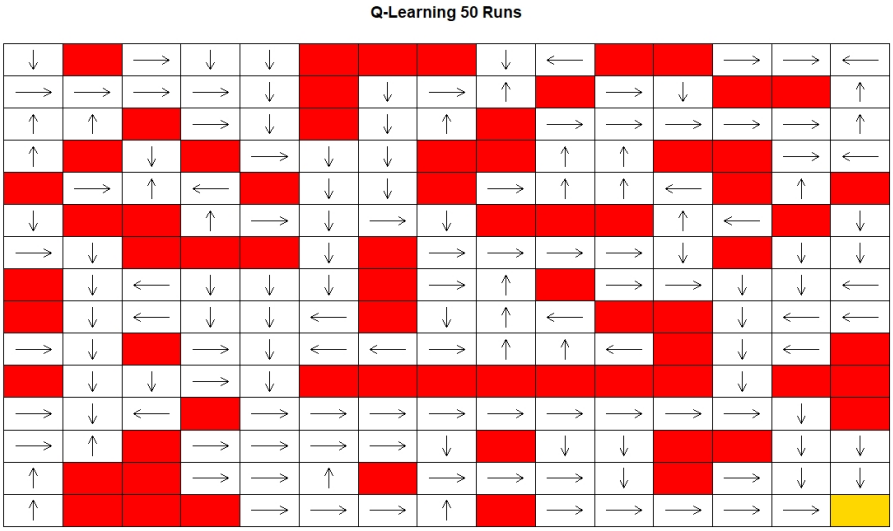
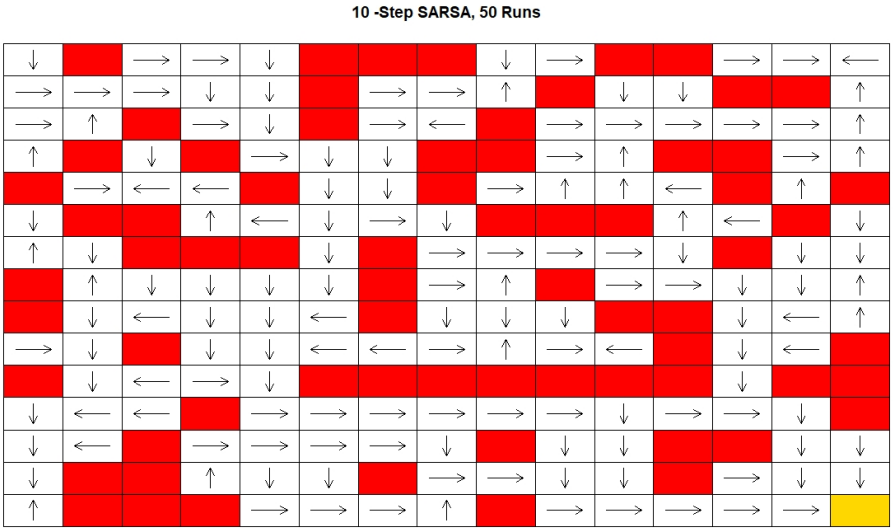


Figure 13: Optimal actions on maze 2 using 10-step SARSA



4. Conclusion

In this report, we have presented 3 approaches to solving Grid World, where an agent needs to navigate through a maze composed of individual tiles by means of 4 actions, moving

up, down, left or right, to ultimately reach a previously specified exit tile. The algorithms used were, a random policy as a control mechanism, SARSA, Q-Learning and n-step SARSA. To do this, we first compared different epsilons and we observed that exponentially decaying epsilons yielded the best results for one learning algorithm. Prior to implementing n-step SARSA to determine which n (in n-step SARSA) would be work well, we compared 5,10 and 15 step values and found that 5 and 10-step had better performance given their rates of convergence and runtimes and decided to implement 10-step SARSA.

We implemented the 3 aforementioned algorithms on an $8x8$ and $15x15$ maze with randomly chosen walls. Q-Learning (QL) was the most effective policy based on 3 factors, scalability, average runtime and average return per policy (or speed to convergence to optimal return per maze). We found that in maze 1, QL converged in $\sim$150 episodes compared to the SARSAs which converged in $\sim$200 episodes with runtimes with an average runtime 0.015min/run slower than QL. on maze 2, roughly twice the size of maze 1, the difference became much clearer. QL converged in $\sim$400 episodes compared to the SARSAs which converged in $\sim$800 episodes with runtimes with an average runtime 0.09min/run slower than QL.

Say something about "future avenues of research"

References

[1] Applications of reinforcement learning in real world, Towards Data Science (Aug 2018).
URL https://towardsdatascience.com/applications-of-reinforcement-learning-in-real-world-1a94955bcd12

[2] Alphago zero: Learning from scratch, DeepMind.
URL https://deepmind.com/blog/alphago-zero-learning-scratch/

[3] F. Godin, Stat497 reinforcement learning slides chapter 1, Concordia University (2019).

[4] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, CoRR abs/1602.01783 (2016). arXiv:1602.01783.
URL http://arxiv.org/abs/1602.01783

[5] U. Halici, Reinforcement learning with internal expectation in the random neural networks for cascaded decisions, Biosystems 63 (1-3) (2001) 21–34 (2001). doi:10.1016/s0303-2647(01)00144-7.

[6] X. Lei, Z. Zhang, P. Dong, Dynamic path planning of unknown environment based on deep reinforcement learning, Journal of Robotics 2018 (2018) 1–10 (2018). doi:10.1155/2018/5781591.