Various Reinforcement Learning Algorithms to Solve Grid World

Matteo Espositoa, William Ngoa, Spyros Orfanosa

*aConcordia University, Montreal, Quebec*

# Abstract

Reinforcement Learning (RL) is widely used in different research areas to tackle problems such as resource management in computers, traffic light control, and playing games like chess 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 episode, average number of episodes until they reach optimality, and scalability by testing the algorithms on different sized mazes. We conclude that Q-Learning is the best suited approach for Grid World since in both of our tests, we observed consistently faster convergence rates to the optimal return value per maze and quicker per episode runtimes.

# 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, an objective that is designed as a numerical reward by learning from an environment. Most reinforcement learning problems can be

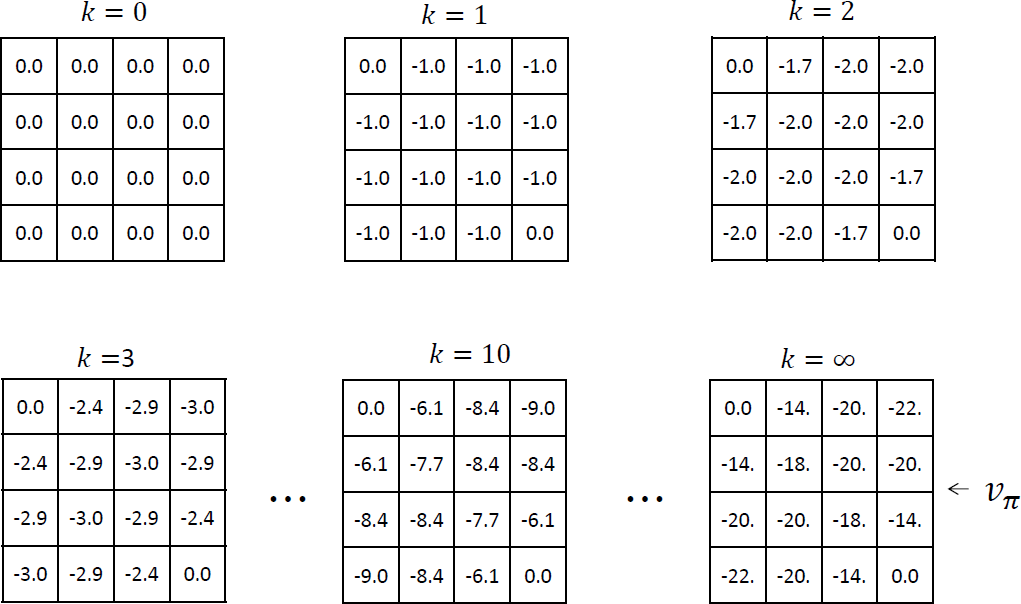
*Email addresses:* [ds.mesposito@gmail.com](mailto:ds.mesposito@gmail.com) (Matteo Esposito), [ngowilliam96@gmail.com](mailto:ngowilliam96@gmail.com) (William Ngo), [spyros.orfanos1@gmail.com](mailto:spyros.orfanos1@gmail.com) (Spyros Orfanos)

*Preprint submitted to Dr.Frederic Godin, Concordia University April 12, 2019*

expressed 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.

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, *vπ* is the state-value function, *k* is the number of iterations. The figure depicts the convergence of the policy evaluation as *k* → ∞.



* 1. *Review of Literature Could condense/remove.*

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 vari- ants of four standard reinforcement learning approaches. It was concluded that the asny- chronous 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. The critic is typically a single linear output connected to a convolutional neural network that has one softmax output for the actor/policy. 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. According to the article, it was its ability to train feedforward and recurrent agents as well as its success in discrete and continuous

action spaces that makes it the most general and successful reinforcement learning agent to date. [1]

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. Such an estimate is somehow implicit in the connection weights. [2]. Whereas in Lei et al. (2018), path planning for mobile robots is the motivation behind the study. Here, the application of a double Q- network (DDQN) created by DeepMind is used for a robot’s path planning in an unknown environment. To make due for the fact that conventional Q-Learning cannot effectively plan a path in random dynamic environment because of the lack of generalization ability and a large Q table and to tackle the issue of dimensionality in a high dimensional state space, the optimal action value function Q in Q-Learning can be parameterized by an approximate

value function *Q*(*s, a*; *θ*) *Q*∗(*s, a*) where *θ* is the Q-network parameter [3]. The introduction

≈

of this weight allows for the fair evalution of the value of the policy at hand.

# 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.

Here, the set of states consists of all valid board positions (*x, y*) where *x* 1*,* 2*, . . . , nrow*

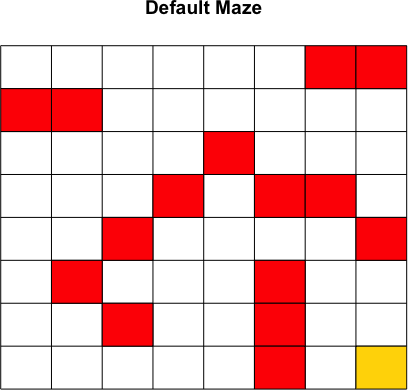
∈ { }

and *y* 1*,* 2*, . . . , 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 which is at point (*nrow, ncol*), starting at point (1*,* 1) in our test cases. To give our al- gorithms an easier time identifying the different elements of the maze environment, we’ve attributed valid spaces, walls and exit tile to integer values of 0,1 and 10 respectively in a 2D-array forming our Maze class. 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



For every action we assign a reward strictly dependent on the environment (set of states), with the following structure:

invalidMoveReward = -0.75 (Hitting a wall - denoted as a red tile, or going beyond the border)

•

* unseenStateReward = -0.05 (Visiting a new, unseen tile that isn’t the exit tile)
* seenStateReward = -0.30 (Revisiting a tile)
* winningGameReward = 5 (Objective: Visiting the exit tile - denoted as a yellow tile)

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. Given the structure of our test maze, our optimal return would be *G*0 = 3*.*85, which will be denoted as the green convergence asymptote curve in our return average plots presented in the following section of this report.

# Analyses

To complement our report, we would like to breifly comment on the structure of our code. The project can be run entirely from main.R. The project has 2 dependencies, packages R6 and rstudioapi, which grant object-oriented functionality and access to the RStudio API respectively Not sure why we need this, maybe write why over here.. In main.R all of our algorithms are called (by means of the source() function and with them, optimal action and average return plots as well as average runtimes are output. Add information about the structure if not previously included/wrong.

apple

* 1. *Considerations*

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, and found that an exponentially decaying epsilon trend was the rule that yielded the most rapidly converging return curve, as seen in the following figure. . . . Write more if necessary...

Figure 3: Q-Learning returns under various *ε*-update rules. (Default *ε* = 0*.*4)

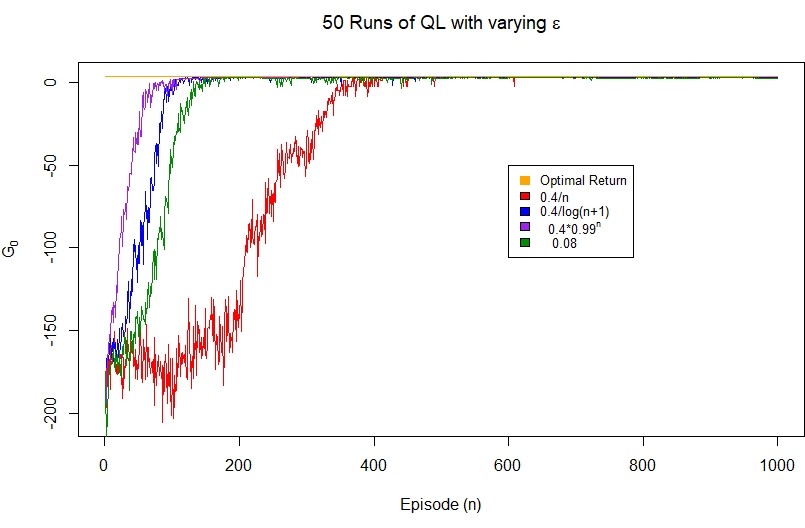


Table 1: Varying epsilon approaches runtime analysis

|  |  |  |
| --- | --- | --- |
| *ε***-function** | **Average # Episodes Before Convergence** | **Runtime (mins/ep)** |
| *ε*  *n* | 450 | 0.189 |
| *ε*  log(*n*+1) | 180 | 0.087 |
| *ε*(0*.*99)*n* | 120 | 0.073 |
| *ε* = 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 best/most appropriate actions everytime after the converging point.

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

Another consideration of any machine leaning project is the bias(underfitting) - vari- ance(overfitting) tradeoff. Since we are only dealing with temporal difference methods, we will only speak about the bias anc variance present in these types of reinforcement leaning algorithms. Generally speaking, TD methods have low variance and are somewhat biased. This low variance is due to the fact that in the update equation for the value function of TD methods, the term/target value *Rt*+1 + *γVπ*(*St*+1) shows that the state-value update only depends on the immediate reward *Rt*+1. We can contrast this to high variance Monte Carlo methods, which have target value *Gt* = *Rt*+1 +*γRt*+2 +*. . .*, which clearly depend on a greater number of rewards.

The presence of bias in the TD methods is related to the fact that the value function estimate is what is being used in the update function and it relies on an initial value we set for it, and therefore is not representative of the actual value of the value function. not sure of this. This can explain why the the beginning of our return curves are not monotonically increasing. With time, the estimate of the value function becomes a decreasingly biased estimator of the actual value function. We should mention why were are discussing bias v. variance, if its not here just for a general comment.

* 1. *Algorithms Could remove, as its mostly repeat of the notes and textbook . . .*

Here we list, very generally, the theoretical backgrounds of the approaches we considered.

* + 1. *Random Policy*

To better visualize the effect of the aforementioned RL methods, we chose to include the use of a random policy. Therefore, we sample randomly from the list of possible moves our agent can perform and observe the state values that are generated. One thing to note is that this approach does not guarantee convergence/successful exiting of the maze provided.

* + 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 *Qπ*(*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 (*s greedy*).

−

Its update rule is defined as follows:

*Qt*+1(*St, At*) ← *Qt*(*St, At*) + *α*[*Rt*+1 + *γQt*(*St*+1*, At*+1) − *Qt*(*St, At*)]

* + 1. *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. Similar to SARSA its aim is to evaluate the Q val- ues. In this case, the learned action-value function, Q, directly approximates q\*, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state–action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. As we ob- served in Chapter 5, this is a minimal requirement in the sense that any method guaranteed to find optimal behavior in the general case must require it. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size param- eters, Q has been shown to converge with probability 1 to q\*. The Q-Learning algorithm is shown below in procedural form.

Contrary to SARSA which transitioned to an action-state *At*+1*, St*+1 to make the update,

this methods considers transitions from an action-state pair *St, At* to a state *St*+1;.

*Qt*+1(*St, At*) ← *Qt*(*St, At*) + *α*[*Rt*+1 + *γ* max *Qt*(*St*+1*, At*+1) − *Qt*(*St, At*)]

*a*

* + 1. *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 *s*-greedy policy. Its algorithm is as follows:

*Gt*:*t*+*n* = *Qt*−1(*St, At*) +

min(*t*+*n,T* )−1

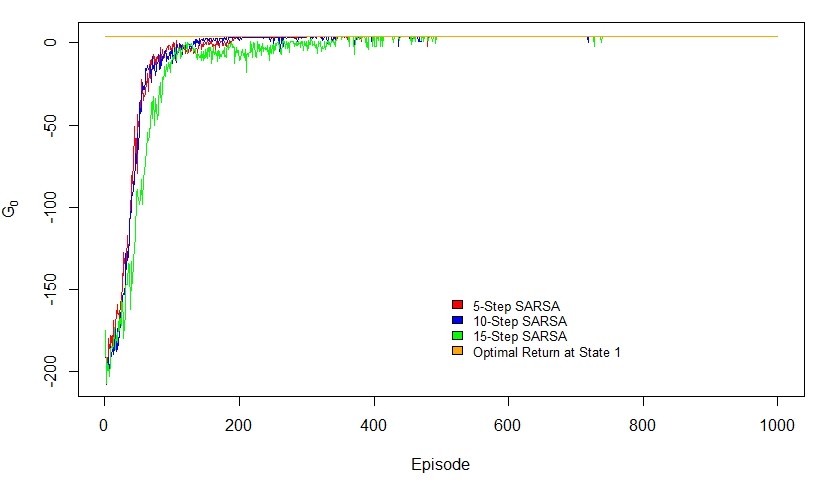
*k*=*t*

Σ

*γk*−*t*[*Rk*+1 + *γQk*(*Sk*+1*, Ak*+1) − *Qk*−1(*Sk, Ak*)]

Note: Prior to implementing the n-step SARSA algorithm we wanted to run a test to determine what the ideal step (n) value for the Grid World problem would be. We ran and compared 5,10 and 15 step SARSA implementations on our first 8*x*8 maze and observed the following results.

Figure 4: Comparison of n-Step SARSA algorithms by return *tt*0 at different n-values with exponentially decaying *ε* values.



From the following figure we can conclude that there isn’t a significant discrepency 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.

* 1. *Maze 1 Results*

Figure 5: Maze 1 returns

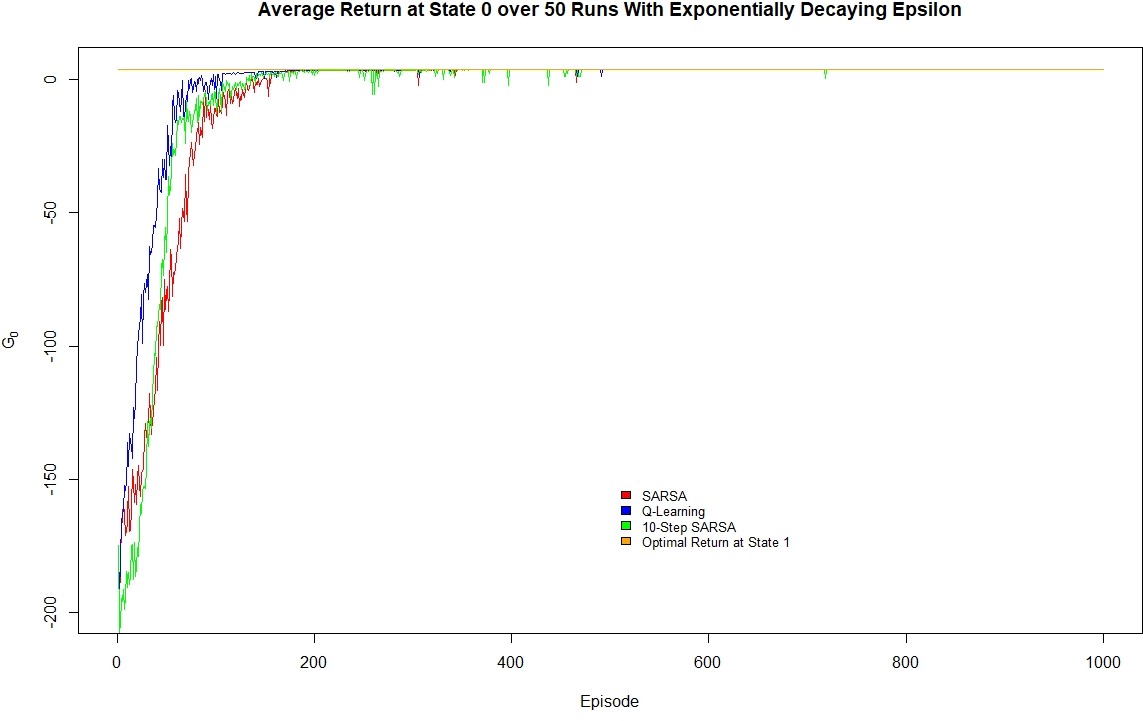


Table 2: Maze 1 runtime analysis

|  |  |  |
| --- | --- | --- |
| **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 much 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. The results are somewhat similar to each other since the maze that the agent’s had to solve was relatively simplistic in that it was only an 8*x*8 grid. In the maze 2 example (section 3.4) we are dealing with a 15*x*15 grid and therefore expect there to be a larger discrepency between the best and remaining algorithm runtimes and episodes until convergence/optimality.

Table 3: Results of random policy after 10000 episodes Do we have this output for maze 1 and maze 2? At the moment I put this table for both . . .

|  |  |
| --- | --- |
| 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 success- fully/terminate, yielding an average reward of -115.596.

Figure 6: Optimal actions on maze 1 using SARSA

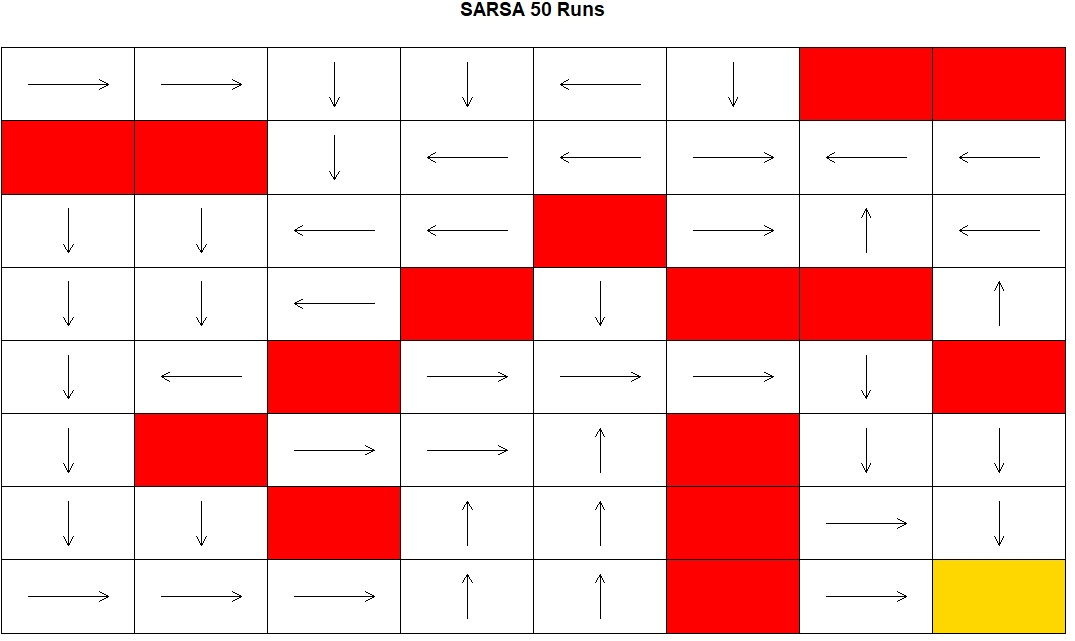


Figure 7: Optimal actions on maze 1 using Q-Learning

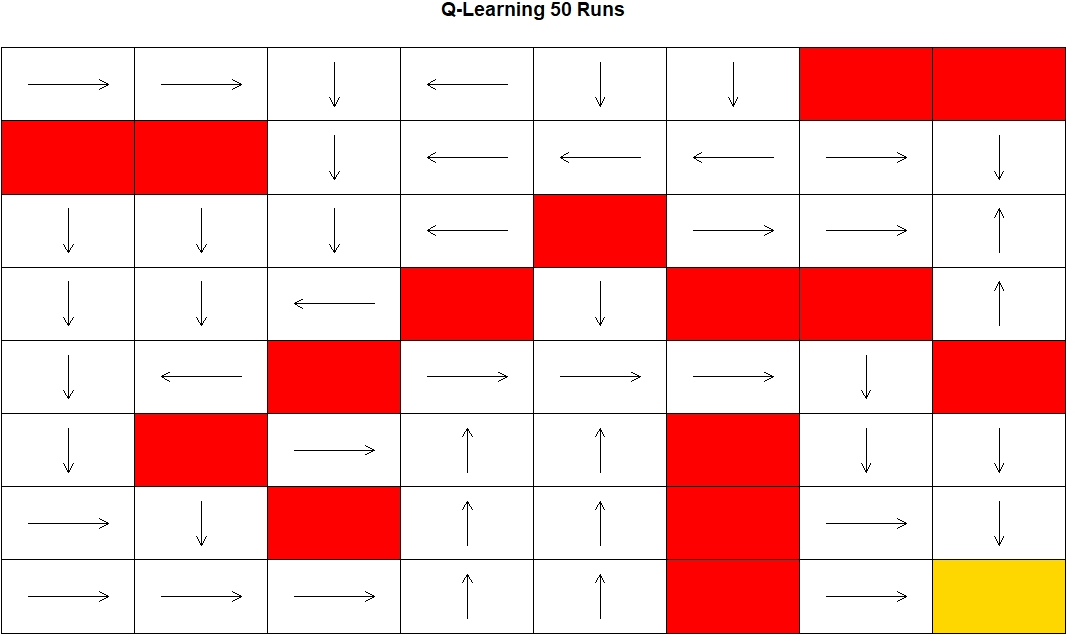
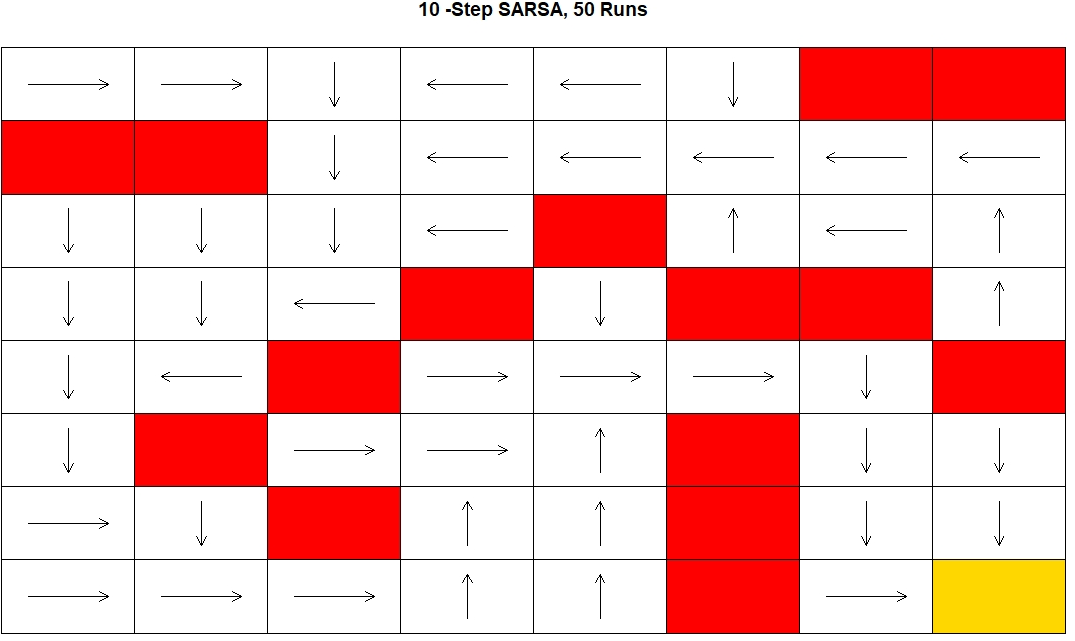


Figure 8: Optimal actions on maze 1 using 10-step SARSA



* 1. *Maze 2 Results*

Figure 9: Maze 2 returns

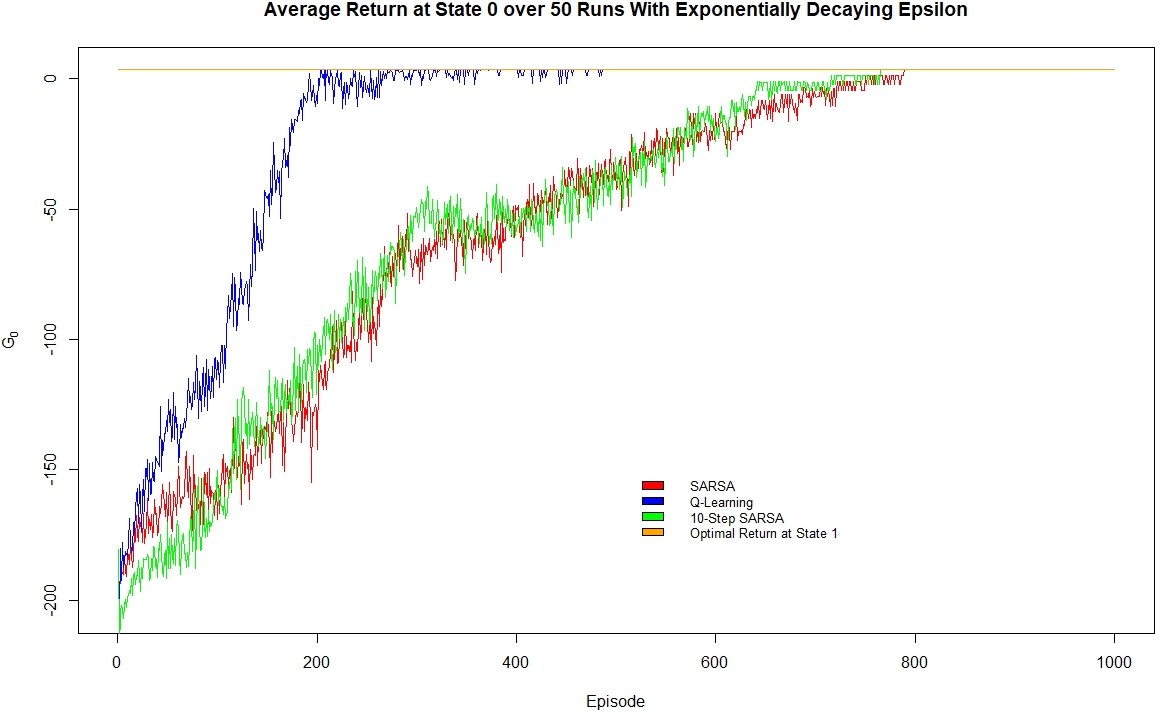


Table 4: Maze 2 runtime analysis

|  |  |  |
| --- | --- | --- |
| **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 ex- pected, 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.

Figure 10: Optimal actions on Maze 2 using SARSA

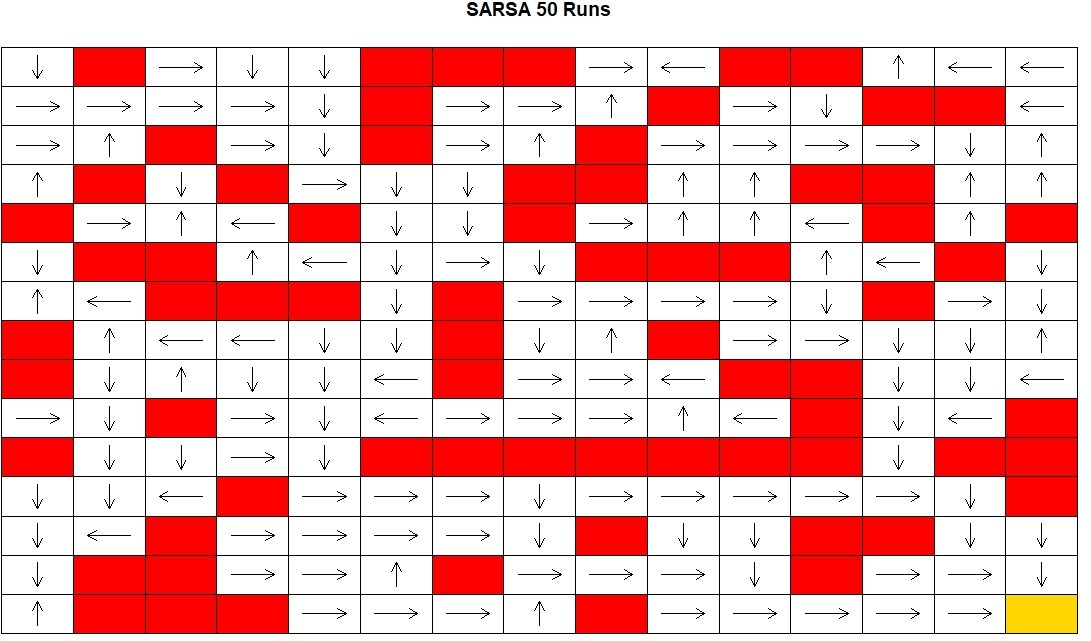


Figure 11: Optimal actions on Maze 2 using Q-Learning

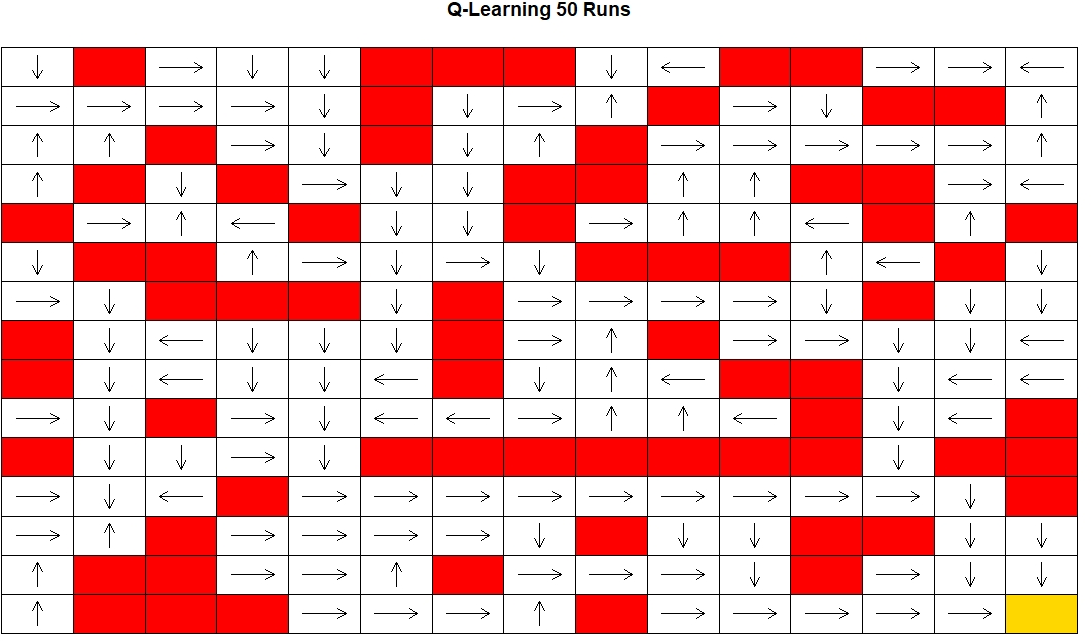
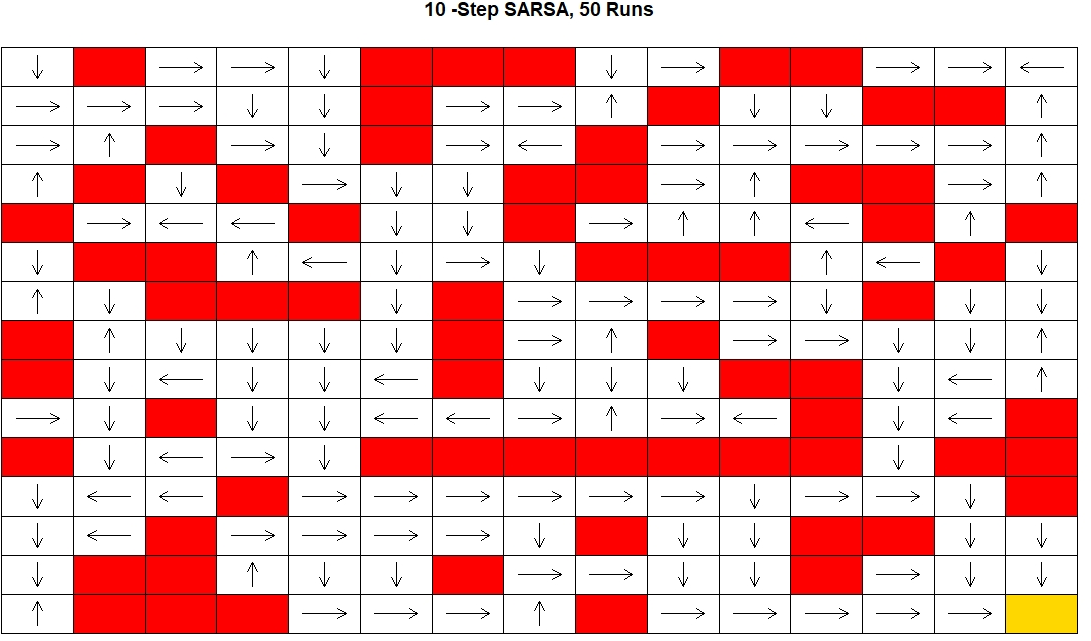


Figure 12: Optimal actions on Maze 2 using 10-step SARSA



# 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. Prior to implementing n-step SARSA, we compared 5,10 and 15 step values and found that 5 and 10-step were better suited to the problem given their rates of convergence and runtimes and decided to implement 10-step SARSA.

We implemented the 3 aforementioned algorithms on an 8*x*8 and 15*x*15 maze with ran-

domly chosen walls where the starting tile and exit tile were consistently at points (1*,* 1) and (*nrow, ncol*) respectively. 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 150 episodes compared to the SARSAs which converged in 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 400 episodes compared to the SARSAs which converged in 800 episodes with runtimes with an average runtime 0.09min/run slower than QL.

Add more here, and should we say something about ”future avenues of research”?

# References

1. 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>

1. 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.
2. 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.