

# Reinforcement Learning

**Overview:** This paper will be a thorough analysis of Markov Decision Processes. We will be analyzing small, medium, and large grid world problems to see the differences in convergence for different sizes of problems. We will be comparing the performances of the three following reinforcement learning algorithms: value iteration, policy iteration, and Q-Learning. The metrics we will be using for comparing the three reinforcement learning algorithms are: runtime, number of iterations for convergence, and how optimal of a policy the result was. However, due to a lack of time I was not able to setup the same test cases for a non-grid world problem. Some non-grid worlds I had in mind but was not able to implement was cartpole, mountain climbing, and reversing symbols on input tape. They will be something I plan to implement over winter break as reinforcement learning is a very interesting subject with a high feeling of reward while watching the AI learn.

## Environment:

- The environment used in this paper will be OpenAI's frozen lake. I chose to use that environment because of the flexibility provided to us by the Gym library. We can easily change the environment's size and the provided environment has stochasticity already provided for us.

## Overview of Markov Decision Processes (MDPs):

Markov Decision Processes are a set of processes that provide a mathematical framework for modeling decision making in scenarios where outcomes are a mixture of deterministic and stochastic results.

MDPs use states, action, probability of a new state given the current state, and the reward for moving to the new state from the current state. The following variables are commonly used in MDP papers:

- $S$ , the set of possible states for which actions can be performed.
- $A$ , the set of actions representing the possible actions that can be performed over the state  $s$ .
- $T(s, a, s') == P(s' | s, a)$ , a transition/probability model representing the probability of moving from state  $s$  to state  $s'$ . This helps to model the stochasticity of certain events by providing randomness to actions, where we are not guaranteed to move where we want.
- $R(s, s')$ , the reward for moving from state  $s$  to state  $s'$ .
- $\pi(s)$ , the policy function of the MDP, specifying a sequence of actions to take given state  $s$ .

One important aspect of MDPs is that solving decision making problems depend on the Markovian assumption which states that the probability of landing in future states should depend only on the present state and not the previous states. It is because of this Markovian property that the transition/probability model uses only the current and next state for computing their respective values.

Examining the symbols above, we can see that MDPs use many different variables but they're all intuitive. MDPs are composed of states, actions, rewards, probability models, and policy functions. We use the current state to find the list of possibly actions to take, and then compute the rewards associated with each possible action. The computation of the possible rewards must factor in the probability model since some MDPs may have stochasticity in them. And finally, we repeat these steps until we reach our destination, in which this will be one of our policy functions. Policy functions are the sequence of actions we take given a certain state for reaching our goal/destination.

### **Overview of Value Iteration:**

Value iteration is a basic algorithm for finding the optimal path in reinforcement learning. Value iterations is a greedy approach and works as follows: when given a list of possible actions for a given state, take the action that yields the highest reward. This may work for some cases, but not all the cases because certain environments may provide low rewards for each action but the final outcome yields the highest reward. For example, the path to becoming a doctor; each passing year, we earn little to no income but once we obtain the license for becoming a doctor, our income jumps significantly higher than the average person.

To circumvent this, we look into the future for possible rewards and provide an expected discount factor. The discount factor works by telling the AI how much we value the rewards in the future; do we value immediate rewards or can we wait for future rewards? How far in the future can we wait for our rewards? The discount factor hyperparameter will determine whether it is best to wait indefinitely for the best reward or wait minimally for a decent reward. This transforms our value iteration algorithm from a greedy approach to a more optimal approach.

But how do we compute the values for each state? To compute the possible rewards for each state, we must look into the future and compute the possible reward for every state up until we reach our destination. To look into the future and compute the possible reward, we will be using bellman's

equation to recursively compute the next state until the destination is reached. The algorithm is defined below:

1. Assign a random reward(or zero) utility to every state  $S$ .
2. For ever state in  $S$ , calculate the new utility for state  $s$  based off the utilities of its neighbors(possible actions).
3. Update the utility for each state using the Bellman equation defined below:

$$U(s)_t = R(s) + \gamma \max_a \sum (T(s, a, s') U_{t+1}(s'))$$

4. Repeat steps 2 and 3 until convergence is reached(convergence is defined as very minimal to no change during the update of the utility for each state).

Convergence is a hyper parameter (the tolerance) defined by the user, where convergence is defined as the change in utility is less than the tolerance and so we should stop running the algorithm.

### **Overview of Policy Iteration:**

Policy iteration is the successor to value iteration in that, policy iterations aims to only compute what is necessary to save time and converge faster than value iteration.

Value iteration has some weaknesses, namely it can be incredibly slow for reaching convergence in some scenarios and that value iteration computes the values for each state and not the policy. The policy is only an indirect finding for value iteration. To save time, we introduce policy iteration as a means to save time by only computing what is necessary; finding the optimal policy directly.

Policy iteration is a lot quicker than value iteration because the set of possible policies for reaching the goal is a lot smaller than the list of possible states. However, there is no free lunch in the world and there is definitely a trade-off for looking only at the possible policies which will be talked about further down in our analysis. The algorithm is as follows:

1. Initialize a random policy. The randomness used here can be a permutation of possible actions for all states in the MDP.
2. Compute the utility for every state  $S$  using the current policy.
3. Update the utilities for each state.
4. Select the new optimal actions for every state and change the current policy if there exists a new optimal policy.

5. Repeat steps 2, 3, and 4 until we reach convergence (no action is changed within our policy).

It's evident that policy iteration is faster because we are looking at policies and not values for each state. The set of possible policies is a lot smaller than the number of possible states in an environment. However, computing policies is a lot more expensive than computing the value for each state. But, in the long run it is expected that policy iteration is faster than value iteration because we are only computing the values for the states which matter for the finite set of possible policies for the environment.

### Overview of Q-Learning:

Both value iteration and policy iterations are incredibly powerful and useful for determining the optimal policy for a given MDP, but the two algorithms shine mostly for offline training where we have knowledge about the transition model and the list of possible rewards for all possible states. However, in many situations we may not have access/knowledge to all this information about the environment.

In fact, not much learning is really done for value and policy iteration because most of the information needed is already provided for the algorithms beforehand. With the information already being provided for us, we can essentially use shortest paths algorithms like Dijkstra's for finding the shortest path from point A to point B. However, it is not that simple since MDPs have an element of randomness involved since some environments have stochasticity in them. And it is because of this stochasticity that value and policy iteration use MDPs bellman equation for computing the most optimal policy.

Ring in Q-learning. Q-learning is a family of model-free reinforcement learning algorithms that attempts to learn the optimal policy when given minimal information; with the given knowledge of the possible states and actions. This process is true learning which is why Q-learning is one of the most important algorithms for reinforcement learning. It learns by first creating a Q table with randomly initialized values, known as Q-values. Then, when a state is visited, we update the Q value for that state using another bellman equation mentioned below:

$$Q(s, a) = R(s) + \gamma \sum_{s'} (T(s, a, s') \max_{a'} Q(s', a'))$$

This equation is defined as follows: the q-value is the value for arriving in state  $s$ , leaving with action  $a$ , and then proceeding optimally to state  $s'$ . Q-learning then proceeds to update the estimated Q-values for each state until we reach convergence. And convergence here is similar to convergence mentioned previously, where the change in the Q-values are less than the tolerance.

The learning in Q-learning was never brought up. If we only select the value in the Q-table to determine our next action, we are guaranteed to reach the local optimum and not the global minimum. To find the best path to take, we would have to implement the exploration vs. exploitation dilemma. The exploration-exploitation dilemma is defined as the trade-off between how much exploring we need to do and how much exploitation of what we currently know. If we explore too much, then we never exploit what we discovered. But if we exploit too much, then we will not have a chance to explore and find new discoveries (in our case, we will be stuck at a local optima). To circumvent this, we must find an equal balance for exploring and exploiting. One possible solution to this is by implementing an algorithm to prioritize exploration in the beginning and slowly decrease our prioritization of exploration and increase our prioritization of exploitation. By doing this, we are exploring more in the beginning since we are still unfamiliar with environment but once we have explored enough, we can start exploiting the knowledge we obtained to fine-tune and find the optimal.

To accomplish this, we introduce a new hyperparameter,  $\epsilon$ , to control the rate at which we decay the exploration and increase the exploitation. The algorithm will work as follows:

1. Initialize random values for the Q-table.
2. Choose the best Q-value for a given state with probability  $1 - \epsilon$ . If we do not choose the best Q-value, then we choose a non-optimal action instead. (exploration vs. exploitation)
3. Update the Q-values for the given state/action.
4. Repeat steps 2 and 3 until convergence is reached.

### **Grid World and Non-Grid World Problems:**

Gridworlds are simple rectangular/square spaces divided into uniform sized squares that are used to represent the real world. Each square in the grid world corresponds to an agent, a decision maker, wall or obstacle, or a goal/terminal state. Gridworlds are frequently used in reinforcement learning for simulations and learning of the real world by using a subset/smaller world.

For this paper, we will be using the FrozenLake gridworld provided to us by OpenAI's gym library. The gridworld is a simulation of a frozen lake with slippery ice, holes for the agent to fall into, and the goal which is where the frisbee is located. The goal of the agent is to navigate the slippery terrain without falling into the hole and obtaining the frisbee. The introduction of slippery terrain is to simulate

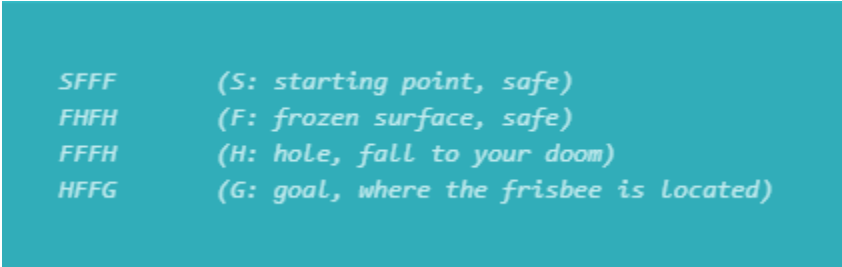
stochasticity. The agent may want to move from point A to point B, but because the floor is slippery the agent may move past point B into point C.

Another example of a gridworld problem is one where the agent is a taxi driver and the agent must navigate around the map to pickup passengers and drop them off to their desired location. In this world, there isn't stochasticity like the previous gridworld, where slippage is involved. However, there is the introduction of walls to block the agent from moving from point A to point B so the agent must know to navigate around it.

And finally, a non-grid world problem is one where the agent is tasked to reverse a string. This problem can be solved using computer science algorithms, but making this basic task a problem for an agent is definitely interesting to say the least. To make this a reinforcement learning problem, we can reward the agent for each successive character the agent reverses correctly and penalizing the agent whenever it incorrectly reverses a character.

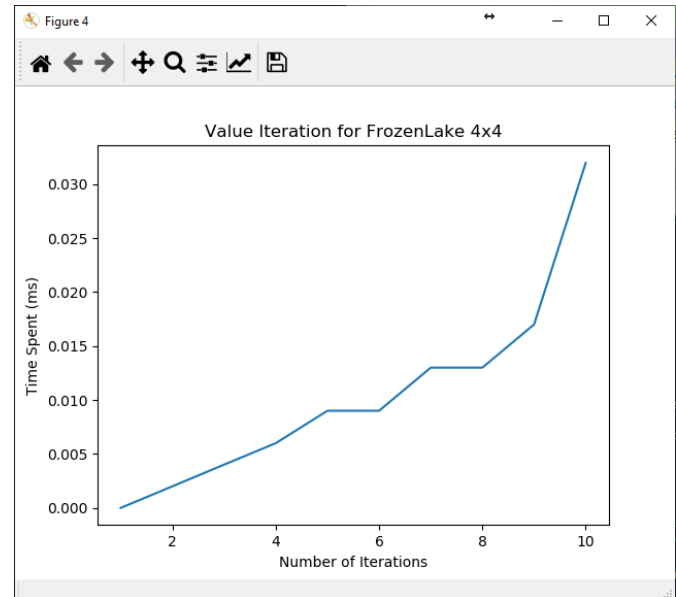
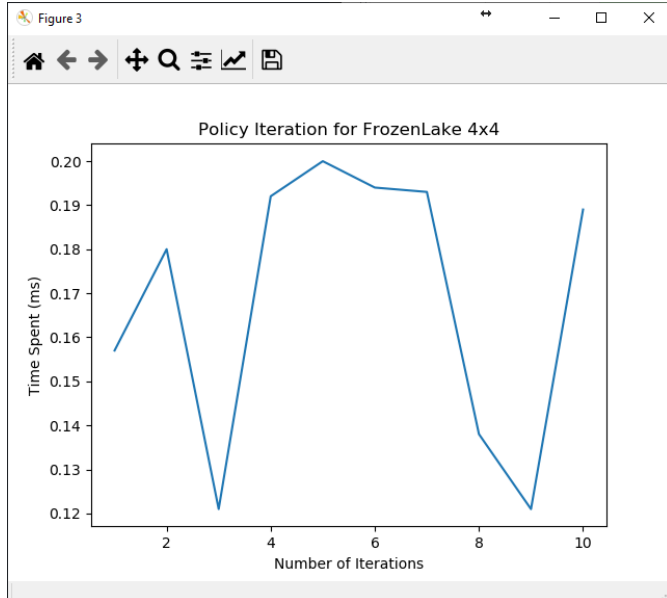
The usage of grid world and non-grid world problems for reinforcement learning is to provide a training grounds for the agent. The goal is to extend what it learns from the grid world into the real world environment.

### **Grid World (small):**

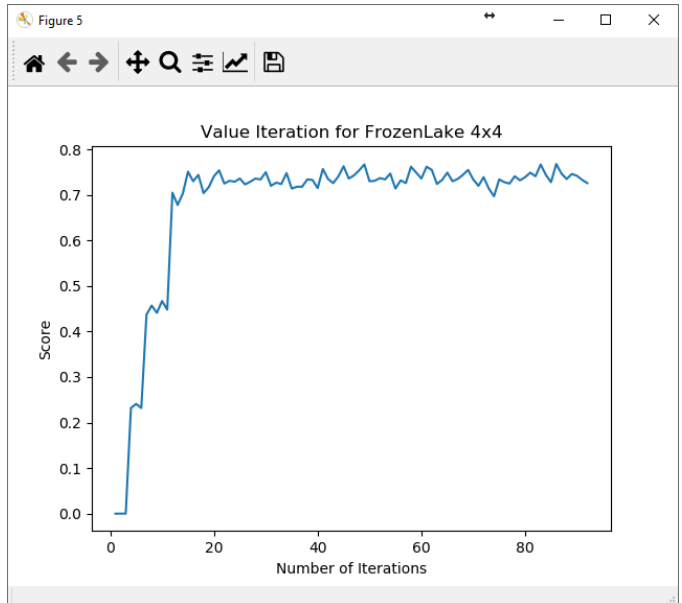
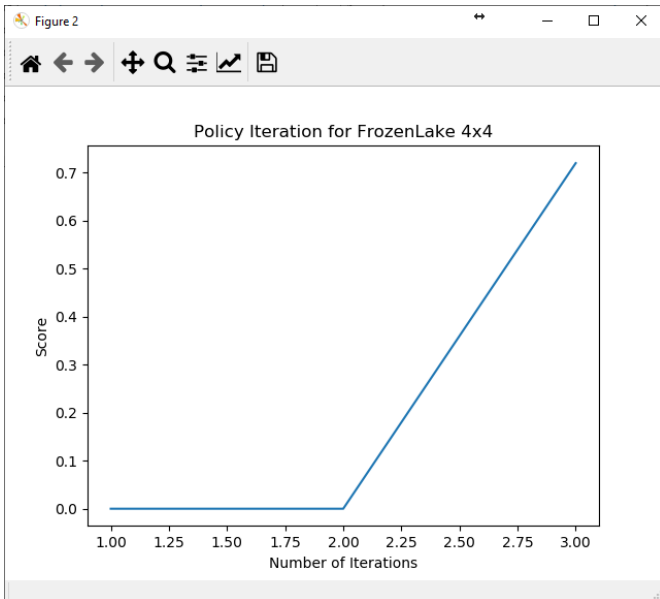


<i>SFFF</i>	<i>(S: starting point, safe)</i>
<i>FHHF</i>	<i>(F: frozen surface, safe)</i>
<i>FFFH</i>	<i>(H: hole, fall to your doom)</i>
<i>HFFG</i>	<i>(G: goal, where the frisbee is located)</i>

The small grid world is the FrozenLake-v0 from OpenAI's gym library. S is the starting point, F is a frozen surface, H is a hole that we fall into, and G is the goal. We use a simple 4x4 grid here to demonstrate how the policy and value iteration algorithms will perform.



We can see that the time spent increases as the number of iterations goes up for both value and policy iteration (normal). However, there is perturbation in the timings for policy iteration and this is most likely due to another program running in the memory of my CPU. However, we can see there is an increasing trend from iteration 1 to iteration 10 of the policy iteration graph. We also notice that the time spent for value iteration is significantly lower than the time spent for policy iteration and this is due to what we talked about previously, where policy iteration spends more time looking through the policies, whereas value iteration spends less time looking through each value but has more values to look at.



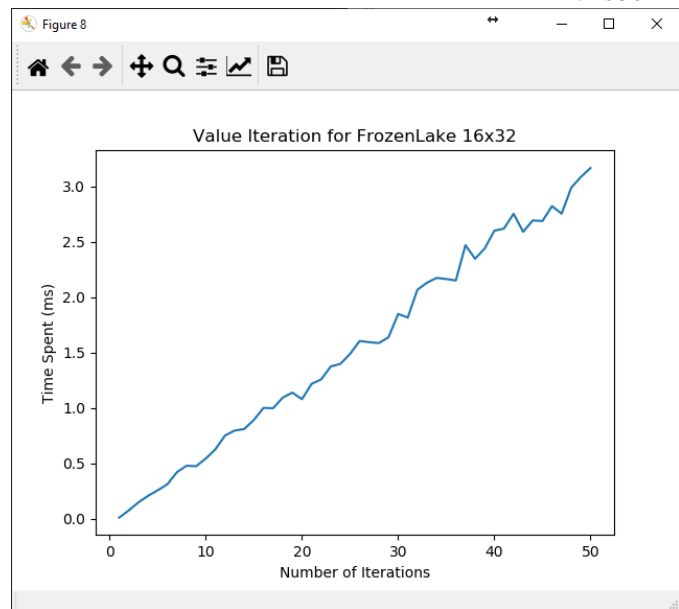
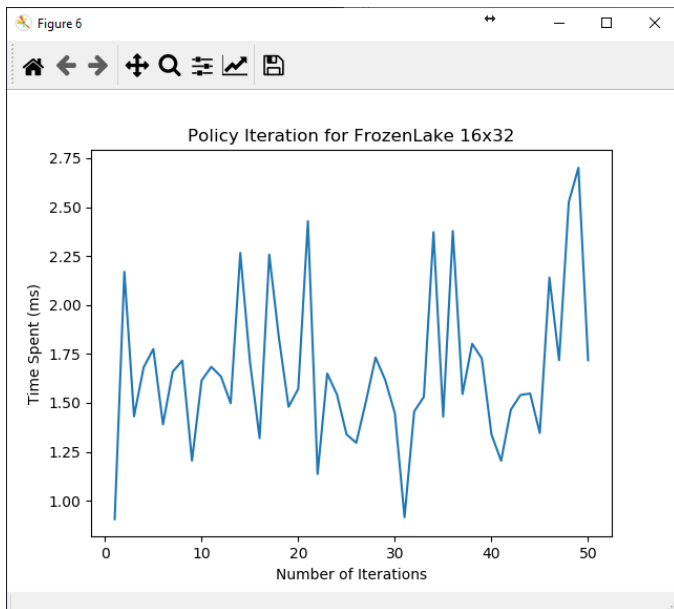
This is a graph showing the rewards/scores for policy and value iteration. As expected, we can see that policy iteration converges after 3 iterations, but value iteration takes approximately 90 iterations to converge. The findings are in line with what we discussed previously about policy iteration converging in fewer iterations than value iteration.

### Grid World (Large):

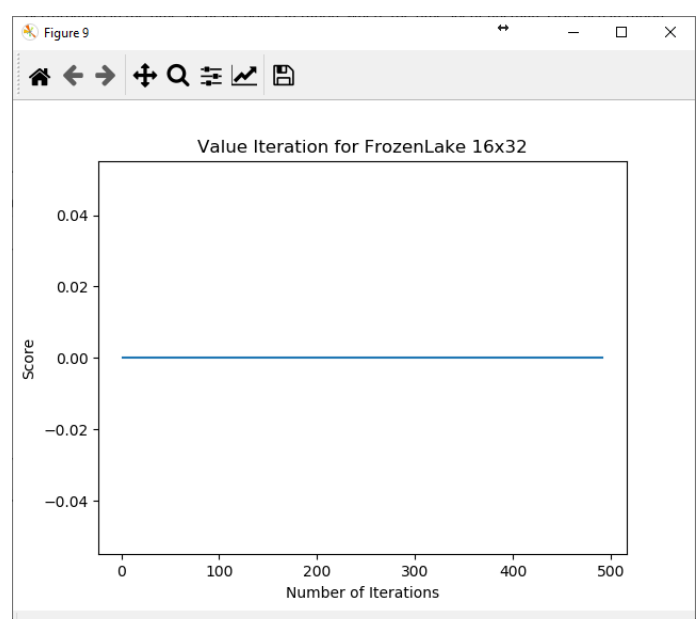
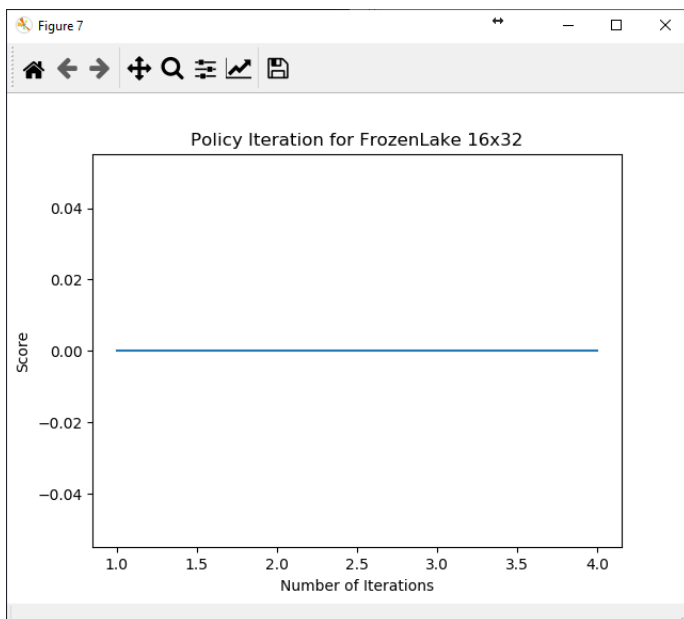
```
[ "SFFFFFFFFFFFFFFHFFFFFFHHHFFFFFFF",
  "FFFFFFFFFHHFHHFFFFFFFHHFHHHHHF",
  "FFFHFFFHHFHHFFFFFFFHHHFFFFFFH",
  "FFFFFFHFFFHFHHFFFFFFHHFHHFHFHF",
  "FFFHFFFHFHFHHFFFFFFHHFHHFHHFFF",
  "FHHFFFHFHFHFHHHHHHFHHFHHHFFFF",
  "FHFFFHFHHFHHHHHHFHFHHFFFFFFHFF",
  "FFFFFFFFFHHFHHFFFFFFFHHFHHHHHF",
  "FFFHFFFHHFHHFFFFFFFHHHFFFFFFFH",
  "FFFHFHHFFFHFHHFFFFFFHHFHHFHFHF",
  "FFFHFFFHFHFHHFFFFFFHHFHHFHHFFF",
  "FHHFFFHFHFHFHHHHHHFHHHFFFFFFF",
  "FHFFFHFHHFHHHHHHFHFHHFFFFFFHFF",
  "FFFFFFFFFHHFHHFFFFFFFHHFHHHHHF",
  "FFFHFFFHHFHHFFFFFFFHHHFFFFFFFH",
  "FFFHFFFHHFHHFHFHHFFFFFFHHHFFFF"]
```

Using the frozen lake grid world but with a size of 16x32 with 512 total states. S, F, H, and G represent the same meanings as the small grid world frozen lake environment.





Due to the size of this problem, I was only able to restrict the number of iterations to 500. But as we can see, the time it took for each iteration of frozenlake 16x32 increased significantly. Again, we see a perturbation in the time spent for policy iteration, where the time spent fluctuates for each iteration. Another error we see here is that the time spent in value iteration is a little larger than policy iteration and this is most likely due to an error in my CPU timing, where I may have had another program running in the background. But overall, we can see that the time increases as the number of iterations increases.



We can see that there is a score of 0 for both policy and value iteration and this is due to the number of iterations being very small. Due to the timing constraint, I was only able to train for 500 iterations and could not reach convergence. Even though the model never converged, we can still obtain some insight from this: value and policy iteration take a long time to converge for large grid world problems.

### Q-Learning for both Small and Large Grid World Problem

```
Time Spent : 19.99

##### TEST #####
episodes : 10000
total reward : 7348.0
average reward: 0.73
```

```
Time Spent : 74.46

##### TEST #####
episodes : 10000
total reward : 0.0
average reward: 0.00
```

On the left, we have the Q-Learner for 4x4 FrozenLake and on the right, we have 16x32 Frozenlake. We can see that Q-Learning has about the same reward as policy and value iteration for the small grid world problem. For the Q-learner in the large grid world problem, we see that it has not converged after 10,000 iterations and I was only able to run it for such a small number because of the time constraint. However, I firmly believe that Q-learning is more suitable for the larger grid world as the Q-table with exploration-exploitation will converge faster with better results than policy/value iteration.

### Conclusion:

We can see that policy and value iteration perform really well for small problems where the state space and action space are small in size. We also found that policy iteration and value iteration provide similar values for rewards. One notable difference between policy and value iteration is that policy iteration takes considerably less iteration to converge with the trade-off that each iteration takes more time to compute. Q-Learning is definitely the winner for computing the optimal policy when given the larger grid world problem because of how powerful the Q-table is for finding the optimal policy. However, Q-learning takes a significantly larger runtime than policy iteration. And it is because of this, that we will only be using Q-learning for large grid world problems where using value and policy iteration is not feasible. Due to my poor time management this month, I was really restricted in time and was not able to run all 3 algorithms on a non-grid world problem. However, this was definitely a very exciting assignment as reinforcement learning is one of my favorite topics and I was always curious as to how the AI, alphastar (starcraft 2 AI), used reinforcement learning to play the game. I plan to put in more effort to learn more about reinforcement learning because this class has given me the foundation.

## References

<https://medium.com/@m.alzantot/deep-reinforcement-learning-demystified-episode-2-policy-iteration-value-iteration-and-q-978f9e89ddaa>

<https://gist.github.com/jojonki/6291f8c3b19799bc2f6d5279232553d7#file-frozenlake-v0-qlearning-py-L36>