ECE:517
Project #2:Report:
Authors: Ronald Randolph, John Geissberger Jr.

1 Introduction

1.1 Background

A bomb and a robot are placed onto a grid of a specified size. To avoid any possibility of harm to itself or others that may be around it, the robot must learn how to push a bomb off the grid and into the water surrounding it. Additionally, this task must be completed with a sense of urgency, because, at the end of the day, well - it is a bomb... and who wants to have a bomb go off on their grid? The answer is no one. On the surface, this task may appear to be rather trivial, but it is not. The key phrase is, “the robot must learn”, which implies one cannot simply relay the bomb’s position to the robot and then program it to push it into the water. Instead, one must design and construct reinforcement learning algorithms such that the robot will teach itself how to find the bomb, how to approach it, and how to push it into the water - all without any outside guidance.

1.2 Problem Specifications

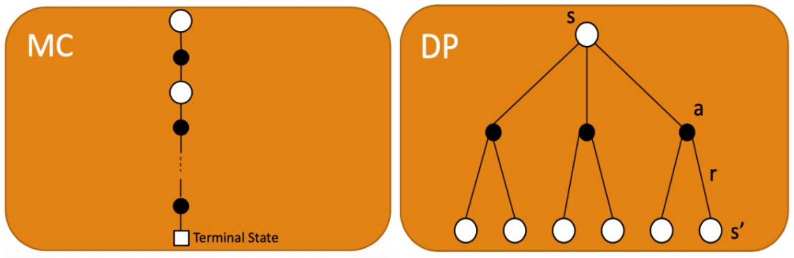
Here are the dynamics of the problem, the grid is 8x8, which is surrounded by water, and the robot and bomb’s starting position is completely random. The robot can only move in the four cardinal directions: North, South, East, and West. In addition, the robot can only push the bomb when it resides in the same location as the bomb and the bomb can only be pushed forward in the direction from which the robot came in. For example, if the bomb is located on the coordinates (4,4) and the robot moves to (4,4) by moving north then the bomb will be pushed North from (4,4) to (3,4). With these dynamics in place, one must design an algorithm such that the robot learns how to move towards the bomb’s location in a precise and optimal manner that will enable it to push the bomb into the water as fast as possible. This project will illustrate different reinforcement learning algorithms, such as Monte Carlo Methods, Temporal Difference Learning, and Q-learning, all of which have the capability of completing the task described above by utilizing different strategies..

2 Algorithms

2.1 Monte Carlo Methods (MC)

In order to solve the problem described above, one could choose to utilize a Monte Carlo method in order to produce a solution. Monte Carlo methods are drastically different from the dynamic programming (DP) methods used in Project 1. Dynamic programming methods require a complete model to be constructed and full knowledge of the environment in order to produce a solution, which is not always possible or ideal. On the other hand, Monte Carlo methods “do not require knowledge of the model’s environment” in order to learn. They only require experience. In other words, Monte Carlo methods require just enough information about the environment such that an episode can be generated. It is truly stunning that Monte Carlo Methods begin with significantly less information about the dynamics of the model when compared to Dynamic Programing, but can still reach an optimal solution regardless.

To illustrate the difference, imagine utilizing both dynamic programming and Monte Carlo methods as a means to teach a learning agent how to play chess. To effectively use dynamic programming, one would require a distribution of all possible future events as well as an evaluation of each of those possible future events in order to determine the optimal move to make. Whereas, utilizing a Monte Carlo method only requires enough information about the dynamics of the environment in order to produce episodes which it can then learn from in order to produce an optimal solution. The Monte Carlo method only needs the ability to play and complete a game in order to learn the optimal moves to make - and none of that requires any rigorous planning, computation, or a completely mapped out environment. To further understand their differences, one can compare the backup diagrams of both methods in the figures below. The juxtaposition of these backup diagrams serve as an illustration of the differences mentioned above.



The crux of Monte Carlo methods is the manner in which they learn from experience and its explanation may seem rather simplistic. After an episode has been generated, the rewards from each of the state action pairs can be averaged. As the algorithm experience more episodes, those averages should begin to converge toward their expected value. Some Monte Carlo algorithms, such as the first-visit MC, possess desirable statistical properties that result in the returns from each episode being identically and independently distributed. Under these circumstances, the law of large numbers applies. Thus, convergence to the expected values can be guaranteed when a sufficiently large number of episodes have been generated.

Lastly, for this project, a first-visit MC control algorithm with a constant update rule was utilized in order to estimate the optimal action-value function. The equation for the constant update is shown in the figure below. However, it should be noted that the figure below contains an update for the state value function, not the action-value function.

$$V(S_t) \leftarrow V(S_t) + \alpha \Big[G_t - V(S_t) \Big]$$

2.2 Temporal Differencing (TD)

Temporal differencing is another effective reinforcement learning method which can be utilized to provide a method in which the robot can learn how to complete the task of pushing the bomb off the grid and into the water. Additionally, because TD methods also learn from experience and do not require complete knowledge of the full environment in order to learn, these methods are found to be very similar to the MC methods, described in the section above. Even though TD and MC methods are very similar, there exists many differentiating aspects that set them apart.

One of the main differences between MC and TD is the differences related to the timing of when learning takes place and when estimates are updated. In order to describe this difference, consider a common scenario that almost everyone, at some point or another, has been involved in - predicting how long it will take to get home after leaving a destination. Specifically, one should focus on how predicted travel typically changes. For instance, suppose an individual predicts it will take 15 minutes to get home given that there is limited traffic. What happens to that prediction once that individual proceeds to get onto the interstate only to see a wreck has occurred and there is bumper to bumper traffic? *Does the individual immediately increase their prior travel time prediction or do they wait until they get home in-order to adjust their prediction?* Typically, it is the former. Generally speaking, individuals would immediately adjust their prior travel prediction once they have encountered a miscalculation like the fact that there was heavy traffic when they predicted there would not be. Using this example, one can draw a parallel from this immediate adjustment in travel time to temporal differencing. That is, temporal differencing immediately adjusts its estimates with the completion of each state transition. Contrarily, Monte Carlo methods update their estimates only after a full episode has been generated. Going back to the example, that would mean an individual would only update their travel time prediction once they have experienced the whole trip and have arrived at their home. This characteristic is the main reason why TD methods tend to converge to a solution faster than Monte Carlo methods.

One of the effects of this difference in update timing is that TD methods require bootstrapping, i.e. updating estimates by utilizing other estimates. This results in a dependent relationship between the estimates generated by TD. Conversely, MC methods produce estimates which are independent from other estimates. Hence, they cannot be, and are not, used as a means to bootstrap. While TD and MC methods may differ in this regard, this same characteristic is shared by dynamic programming methods. As such, TD methods could be described as a combination of both dynamic programming *and* Monte Carlo methods. The one-step TD (TD(0) update equation can be found in the figure below to better understand how the TD algorithms work differently compared to MC. Specifically, the state value function is updated as a result of a one-step transition.

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]$$

2.3 Q-learning

According to the text, Q-learning was an early breakthrough in the field of reinforcement learning due to the fact that this off-policy TD control method was able to approximate the optimal action-value function directly independent of the policy being followed. As a result, the analysis of the algorithm was simplified and early convergence proofs were produced. Specifically, when sufficient training data, generated under any ε-soft policy, is produced, Q-learning converges to the action-value function, $Q(S_t,A_t)$, with a probability of 1. For the task provided within this project, it can be shown that Q-learning converges to the optimal action-value function much faster when compared to MC First-Visit. One can observe the update rule of the Q-learning algorithm below

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big].$$

3 The Markov Decision Process (MDP)

3.1 General Definition

Many reinforcement learning problems can be framed as Markov Decision Processes - or MDP's - which have many benefits. This MDP framework can provide an elegant abstraction for the reinforcement learning problem. To obtain these benefits, this problem was framed as an MDP. First, one must be familiar with what makes up an MDP. Generally speaking, an MDP can be defined as a 5-tuple, as seen in the figure below.

$$\mathcal{M} = \langle S, A, R, P \rangle$$

The elements of this 5-tuple are, States, Actions, Rewards, and a Transition Probability Model. For the purposes of the problem at hand, these elements are defined as follows. The state space is comprised of all possible locations on the grid for the robot and bomb at any given step. There are four possible actions regardless of state. These actions are for the robot to either move North, South, East, or West. There are two separate reward structures for this reinforcement learning problem. The first reward structure simply rewards -1 at each step, regardless of action and state. The second structure's rewards are associated with robot's movements, the direction the robot has pushed the bomb in, and whether the bomb has been pushed into the river. Lastly, a transition probability model was not necessary for the implementation of this problem. These definitions are illustrated below.

States - S - two10x10 Grids

Both Grids are identical. One represents the Robot's possible positions and the other represents the Bomb's possible positions.									
(0,0)	water	water	water	water	water	water	water	water	(0,9)
water	(1,1)	(1,2)	(1,3)	...	...	...	….	(1,8)	water
water	(2,1)	(2,2)							water
water	(3,1)	….	(3,3)						water
water	...			...					water

water	...				...				water
water	...					...			water
water	...								water
water	(8,1)	...	...	...	...	...	...	(8,8)	water
(9,0)	water	water	water	water	water	water	water	water	(9,9)

Action Space - A

A(0) - North	A(1) -South	A(2) - East	A(3) - West

Reward Schemes

r_bomb_not pushed_away_from_center	-1
r_bomb_pushed_away_from_center	+1
r_bomb_pushed_into_water	+10

r_every_step_taken	-1

For the MC, TD, and Q-learning algorithm the state action value function, Q(s,a), was estimated.

4 Code Design and Data Structures

The included file, robot_bomb*.py*, contains the implementation designed to solve this problem through the structure of a MDP and utilization of MC, TD, and Q-learning. For the MC method, the on-policy first-visit MC control algorithm was utilized.

In order to generate episodes such that data can be generated and fed into the various algorithms object orientated programing was utilized. As such, this program's structure is comprised of two main classes - *robot* and *environment*. Both of these classes contain multiple data structures and methods which manipulate them. The robot class is designed to implement the tasks carried out by the robot during the episodes in addition to storing relevant information associated with the robot. The environment class is designed to perform three main functions, the first is to maintain the bomb's position and other relevant information associated with the bomb throughout an episode. Secondly, it possesses numerous class attributes associated with the state value function as well as the policy. Third, it maintains helper functions which perform necessary operations required throughout the completion of a given episode.

The robot class possesses the following class attributes: a time step variable, a list to maintain the sequence of robot positions, actions, and time step, a list to hold the rewards received, and a position variable in the form of an np array with one row and two columns to represent the robot's current position on the grid. In addition to the class attributes listed above, class functions were constructed that implemented various steps within an episode. For instance, a next_position(self,action) function was constructed which would take an action as an input and determine the robot's next position on the grid while considering the case when the robot would fall into the water. Lastly, a helper function was created which allowed for the sequence of steps within an episode to be printed out.

The environment class contains attributes similar to the robot class such as, a time step variable, a list to maintain the sequence of bomb positions, and a position variable in the form of an np array with one row and two columns in order to represent the bomb's current position on the grid. The environment class included additional class attributes such as, a qa attribute of the form of an np array with the following dimensions np.array(9,9,9,9,4), a qa_check attribute of the form of an np array with the following dimensions np.array(9,9,9,9,4), and a policy attribute of the form of an np array with the following dimensions np.array(9,9,9,9,4). The qa attribute represents state-action value function, there exists a total of 162, (9*9)*2, different states representing the different possible combinations of the robot's position and the bomb's position on the grid. The qa_check attribute was utilized as a logical check within the first visit algorithm. The first visit algorithm only includes first visits to state action paris. Hence, the qa_check was a means to track whether a particular state action pair had already been visited or not. Lastly, the policy class attribute was a strictly a data structure to maintain the policy while it was being updated throughout the duration of an algorithm.

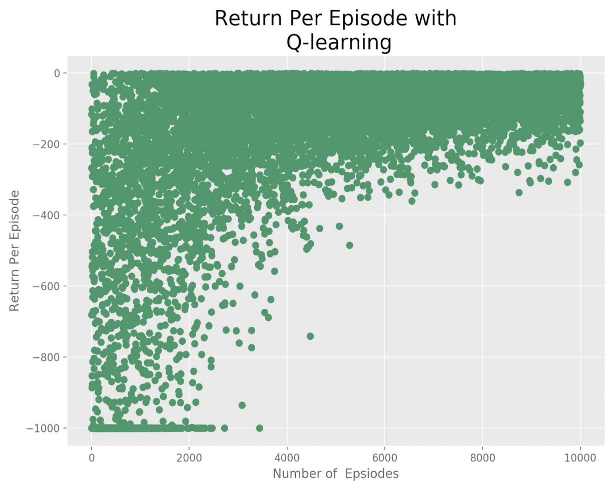
Like the robot class, the environment class contains functions that assist in the generation of episodes. For instance, a get_rewards_bomb_push(self,last_pos) function was created in order to calculate the reward associated with a particulate bomb push. The get_rewards_bomb_push(self,last_pos) function takes the last position of the bomb as input, it has access to the bomb's current position through the environment instance, and returns a reward after determining whether the robot had either pushed the bomb away from the center, closer to the center, or into the water. In addition, the environment class contains a first_visit_MC(self,robot,epsilon,constant) which is designed to implement the On Policy First Visit MC control algorithm. Furthermore, the q_learning(self,robot,epsilon,constant) function is designed to generate episodes and implement Q-learning. Lastly, the environment class contains a function, printb(self,robot) which is designed to plot the movements of the robot throughout a given episode.

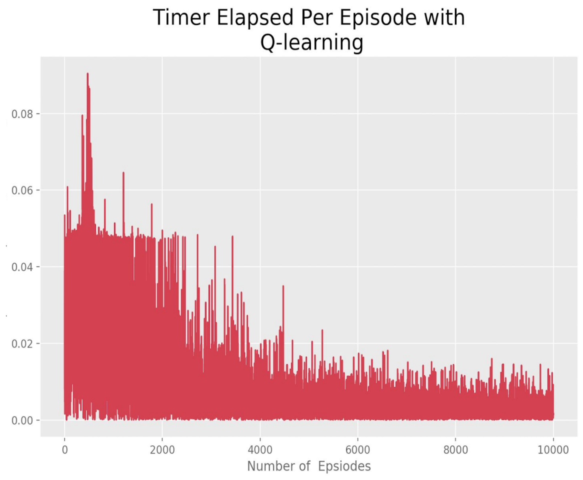
In addition to all of the functions mentioned above there was an additional function constructed that exists outside of both of the classes, episode_generator(). Episode_generator is, as the name implies, a function designed to generate episodes for the Monte Carlo Fist Visit algorithm. The function interacts with instances of the robot and environment class in order to walk through a particular number of episodes while printing out relevant information and visuals that depict a given episodes sequence of states, actions, and rewards.

5 Correctness Results using Q-learning

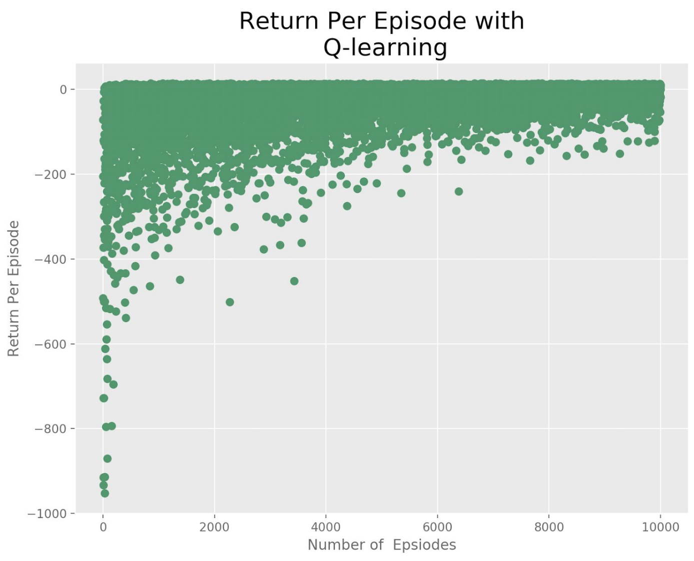
Section four of the project includes a specific setup related to the problem at hand and it dictates the following, epsilon and alpha are set equal to 0.1, the grid is 8x8, and there is no discount. In addition, two scenarios with different reward structures were provided. The first scenario describes a reward structure where every step gives a -1 reward. The second scenario describes a reward structure where every step gives a -1 reward, moving the bomb from the center gives a reward of +1, and pushing the bomb into the water gives a reward of +10. Under these conditions the Q-learning algorithm was run utilizing a total of ten-thousand episodes.

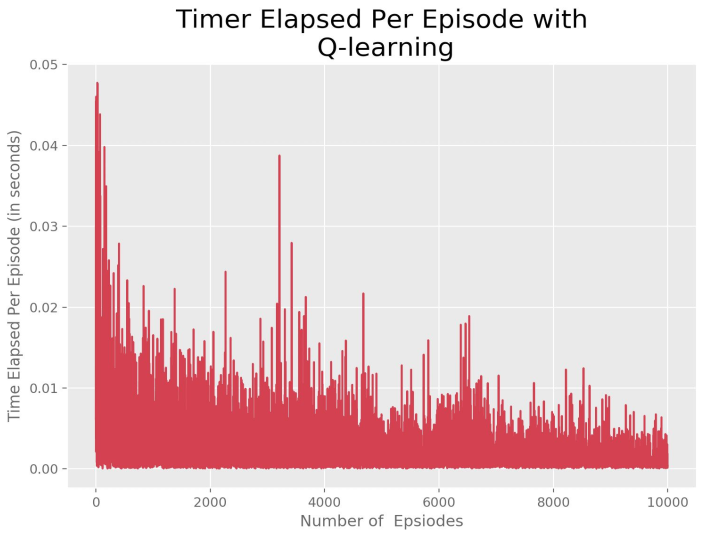
First Scenario Results:



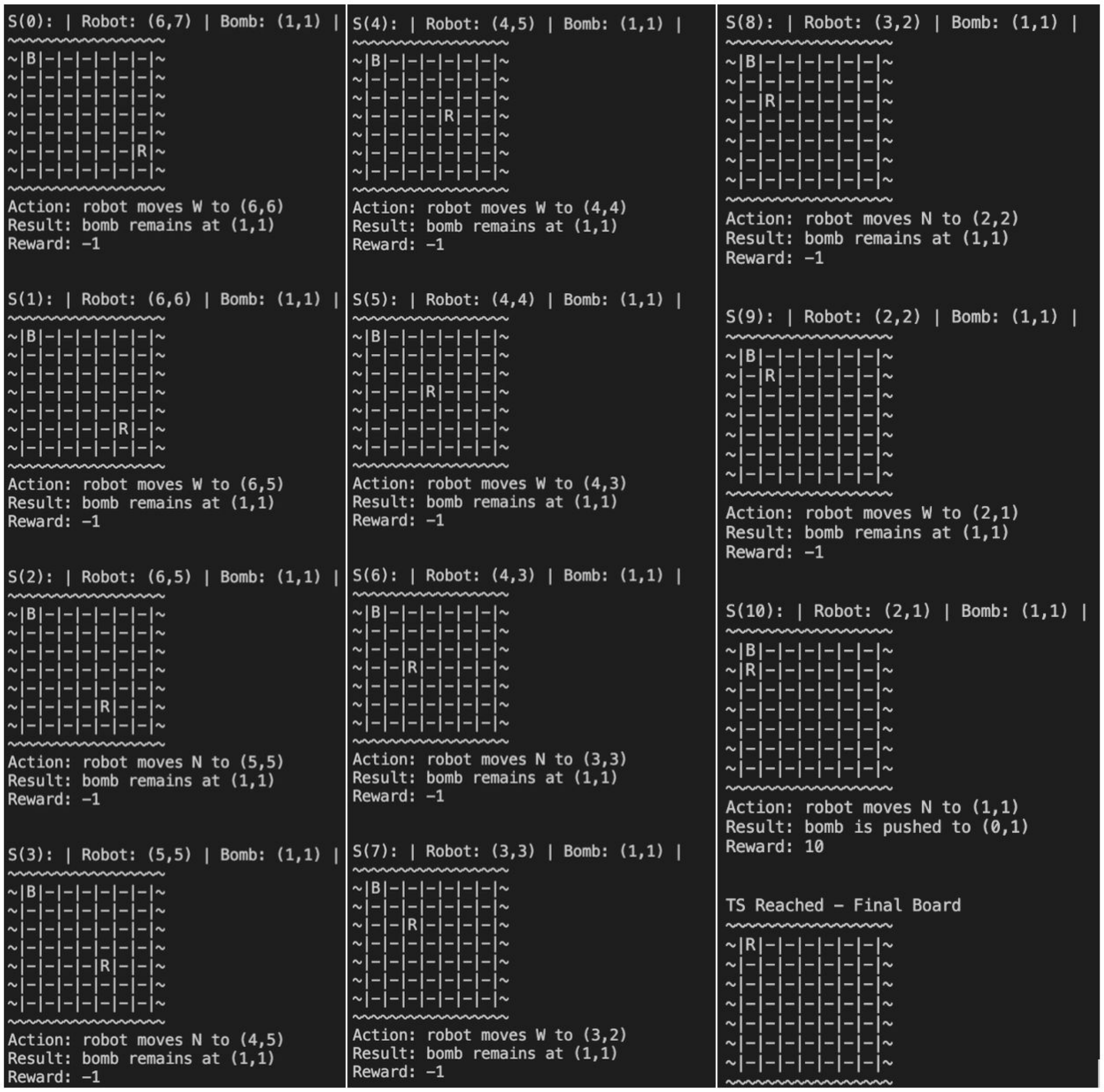


Second Scenario Results:





The results from the second scenario are expected, it appears that the second scenario produces much higher returns per episode and requires much less computation time when compared to the first scenario. However, the results from the first scenario are surprising, to say the least. Both group members concluded that the reward structure provided in scenario one would result in the robot just randomly moving around the board. However, that was not the case due to the fact that the robot learned how to push the bomb into the water. After observing the results, one can conclude that the robot will learn how to push the bomb into the water because that action will terminate the episode quicker, resulting in a higher reward. The reward structure in scenario 1 is not desirable since the bomb requires much more time to learn and computation time is significantly increased.



```
S(0): | Robot: (6,7) | Bomb: (1,1) |
~~~~~~~~~~~~~~~~~
~|B|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|R|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
Action: robot moves W to (6,6)
Result: bomb remains at (1,1)
Reward: -1

S(1): | Robot: (6,6) | Bomb: (1,1) |
~~~~~~~~~~~~~~~~~
~|B|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|R|-|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
Action: robot moves W to (6,5)
Result: bomb remains at (1,1)
Reward: -1

S(2): | Robot: (6,5) | Bomb: (1,1) |
~~~~~~~~~~~~~~~~~
~|B|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|R|-|-|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
Action: robot moves N to (5,5)
Result: bomb remains at (1,1)
Reward: -1

S(3): | Robot: (5,5) | Bomb: (1,1) |
~~~~~~~~~~~~~~~~~
~|B|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|R|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
Action: robot moves N to (4,5)
Result: bomb remains at (1,1)
Reward: -1
```

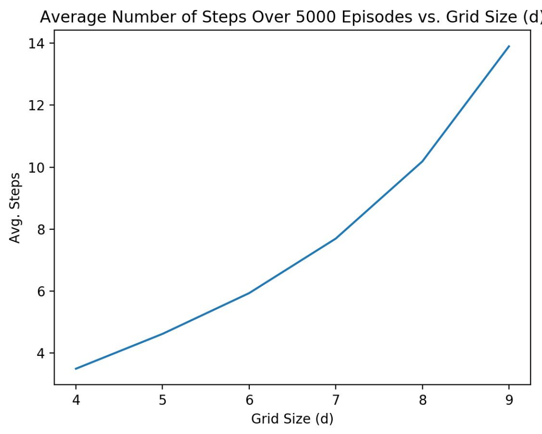
```
S(4): | Robot: (4,5) | Bomb: (1,1) |
~~~~~~~~~~~~~~~~~
~|B|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|R|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
Action: robot moves W to (4,4)
Result: bomb remains at (1,1)
Reward: -1

S(5): | Robot: (4,4) | Bomb: (1,1) |
~~~~~~~~~~~~~~~~~
~|B|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|R|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
Action: robot moves W to (4,3)
Result: bomb remains at (1,1)
Reward: -1

S(6): | Robot: (4,3) | Bomb: (1,1) |
~~~~~~~~~~~~~~~~~
~|B|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|R|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
Action: robot moves N to (3,3)
Result: bomb remains at (1,1)
Reward: -1

S(7): | Robot: (3,3) | Bomb: (1,1) |
~~~~~~~~~~~~~~~~~
~|B|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|R|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
Action: robot moves W to (3,2)
Result: bomb remains at (1,1)
Reward: -1
```

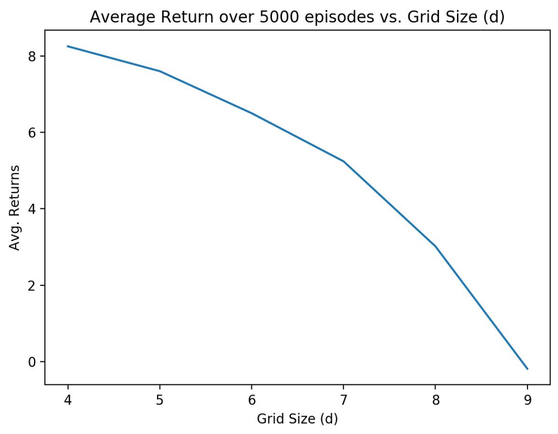
```
S(8): | Robot: (3,2) | Bomb: (1,1) |
~~~~~~~~~~~~~~~~~
~|B|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|R|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
Action: robot moves N to (2,2)
Result: bomb remains at (1,1)
Reward: -1

S(9): | Robot: (2,2) | Bomb: (1,1) |
~~~~~~~~~~~~~~~~~
~|B|-|-|-|-|-|-|~
~|-|R|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
Action: robot moves W to (2,1)
Result: bomb remains at (1,1)
Reward: -1

S(10): | Robot: (2,1) | Bomb: (1,1) |
~~~~~~~~~~~~~~~~~
~|B|-|-|-|-|-|-|~
~|R|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
Action: robot moves N to (1,1)
Result: bomb is pushed to (0,1)
Reward: 10

TS Reached - Final Board
~~~~~~~~~~~~~~~~~
~|R|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~|-|-|-|-|-|-|-|~
~~~~~~~~~~~~~~~~~
```

Example episode after 50,000 episodes using the Q-Learning algorithm.

6 Results Analysis

One, perhaps trivial, exploration conducted was the analysis of the effect of the grid size in the problem. The program takes a parameter, d, which specifies the grid to be of size dxd. Our team predicted that an increased grid size would lead to drastic increases the average steps per episode and overall time-to-convergence. The reasoning behind this position is that by simply doubling the dimensions of an 8x8 grid to a 16x16 one, increases the number of states from 10,000 to 104,976 states - including terminal states. The magnitude of this change can be view even clearer when one considers the action-value function. The number of action-values increases from 40,000 to 419,904. The effects of increasing the parameter, d, is illustrated in the figures below.

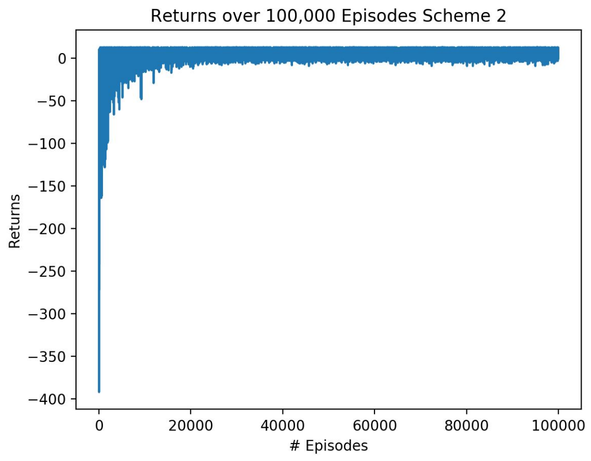


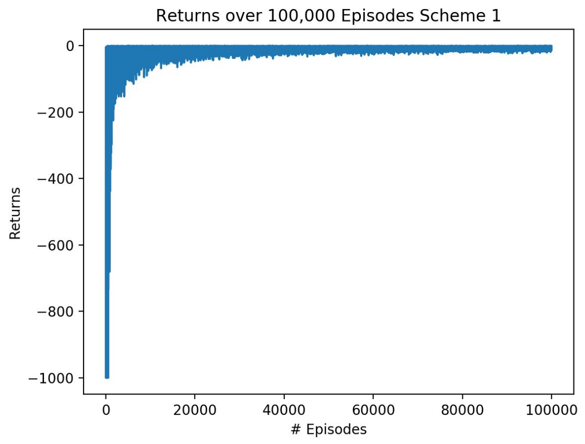


As a second consideration, we wanted to illustrate the differences between the two supplied reward structures. To review, the structures were defined as follows:

1. Every step gives a reward of -1
2. Every step gives a reward of -1. Moving the bomb away one cell from the center gives a reward of +1. Pushing the bomb into the water gives a reward of +10.

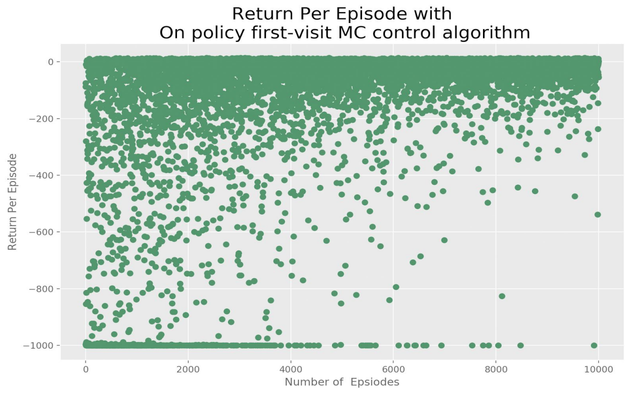
The comparisons of the effectiveness of these two schemes are illustrated in the figures below.

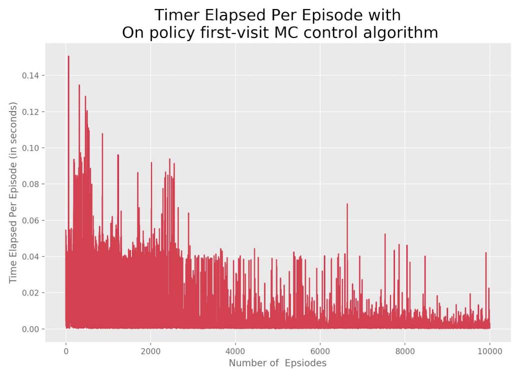


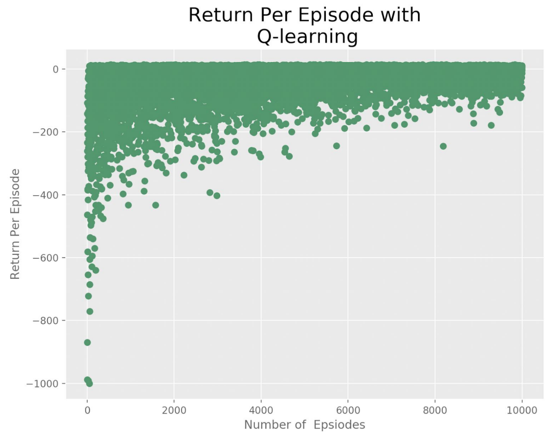


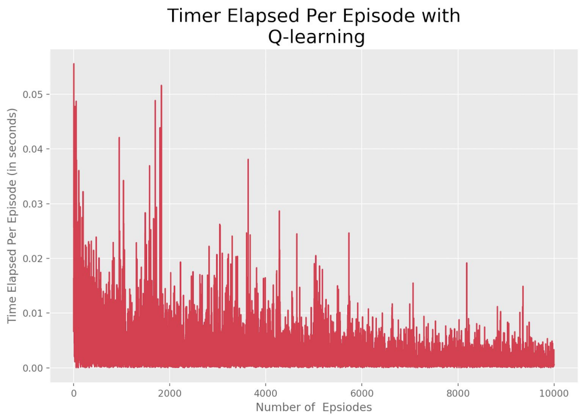
7 Conclusion

Both the Monte Carlo and Q-Learning approaches succeeded in enabling a robot to learn the most optimal way to dispose of the bomb. From the data above, it is apparent that this was a task best suited for TD learning as it was able to converge much faster than the first-visit Monte Carlo.









8 Appendix

Our group determined it would be best to split the work associated with the report and the code such that no individual was particularly responsible for any particular section. Therefore, no individual was necessarily “in charge” of any particular part. The report was created via a shared google doc and code was shared back and forth as well. Throughout the completion of this project both group members were actively engaged in producing solutions in regards to the code and writing different sections of the report.

References

R. Sutton and A. Barto, “Reinforcement Learning: An Introduction,” 2018, 2nd Ed.