Introduction

Reinforcement learning is a type of machine learning technique that enables an agent to learn in an interactive environment by trial and error using feedback from its own actions and experiences. The best way to introduce the concepts of reinforcement learning is through games, in this case, an exploration game is introduced which basically introduces an agent to an unknown world. This agent has to discover the best path to a reward position without knowing the location and has to avoid a pit of doom and obstacles. All of this is performed in a rectangle bounded environment.
In order to explore the unknown world, a learning procedure is introduced that enables the agent to understand which the best steps are to take. Our approach introduces a Q learning algorithm that considers values for every decision.

Q-learning

The Q-learning algorithm is used because it helps us maximize the expected reward by selecting the best of all possible actions, considering initially an exploration of the environment and then an exploitation of the explored environment.

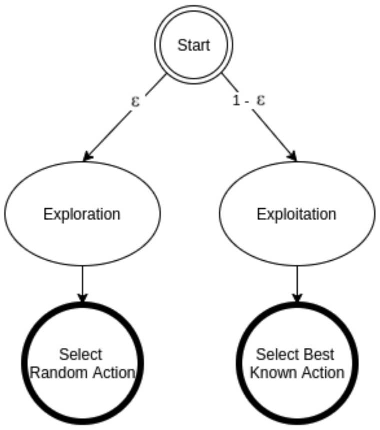
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

Bellman’s Equation

These values are updated using Bellman’s equation that takes two inputs: the state and the action of the agent. This equation is used to fill a Q-table that helps the agent update its exploration policy. In this case, S corresponds to state and A for action. R stands for the reward; t denotes the time step and t+1 denotes the following time step.

As said earlier Q-learning selects an action based on its reward value. For this approach, an epsilon-greedy policy is initiated that takes advantage of prior knowledge and exploration to look for new options.

The aim is to have a balance between exploration and exploitation. Exploration allows us to have some room for trying new things, sometimes risking our actual knowledge of the environment.



Epsilon-Greedy Procedure

Why Q-learning over SARSA :

Q-learning:

$$\boxed{\text{New } Q(s,a)} = \boxed{Q(s,a)} + \boxed{\alpha}\boxed{[R(s,a)} + \boxed{\gamma}\boxed{\max Q'(s',a')} - \boxed{Q(s,a)}]$$

- 🟥 New Q Value for that state and the action
- ⬛ Learning Rate
- 🟫 Reward for taking that action at that state
- 🟪 Current Q Values
- 🟩 Maximum expected future reward given the new state (s') and all possible actions at that new state.
- 🟧 Discount Rate

SARSA :

$$Q(s,a) \;=\; Q(s,a) + \propto [R(s,a) \;+\; \gamma * Q(s',a') \;-\; Q(s,a)]$$

SARSA is an algorithm where the agent doesn't explore all possible paths, instead it exploits the best options to the agent's knowledge now, which might not be the best path. To reach the best reward in the world, this method needs to be avoided.

Experimentation

There are different codes for different parts, namely *part1.py*, *part2.py*, *part3.py*. In the first part, a deterministic model with the greedy epsilon policy approach is used. For the second part, movement probabilities are added to form a transition model and convert the model into a non-deterministic one. I still used the greedy epsilon policy approach in this.

For the third part, I introduced a new exploration policy that balances the exploration and exploitation approaches in a better way. This method is called **decaying epsilon**, which allows the agent to have a good exploration of the environment at the beginning of the process with a high probability of exploration and reaches a low probability of exploration once the agent has become familiar with the environment. The epsilon does not decay at a particular rate, but it depends on the number of trials completed by the agent. As the trails keep increasing, epsilon decreases and the exploration decreases.

e = 1/(trails+1)

Considering that our approach will be tested with big environments and a short amount of time, I consider that the best exploration policy for the agent is a decaying epsilon. It gives a good symmetry in order to find the best reward in the world. Also, this approach allows instant feedback on the action taken previously, which in this case is an advantage given the time circumstances of this experiment.

The second part is ran with multiple epsilon values (0.01, 0.1 and 0.3). The algorithm is run 10 times for each of the epsilon values to get an average of the mean rewards obtained. Also, for one of the runs, a graph of time elapsed vs the mean reward obtained at that point in time is plotted. Then I also run the third part and plot a graph for it as well.

Results

The results generated below are all for the following grid world of size 4x5. The treasure reward is 10 and the pit penalty is -5. The parameters taken are: reward per action = -1, gamma = 0.9, probability to move correctly = 0.7 and ran it for 1 sec.

GRID WORLD:

10	0	0	0	0
-5	0	0	X	0

```
0   0   0   0   0
S   0   0   0   0
```

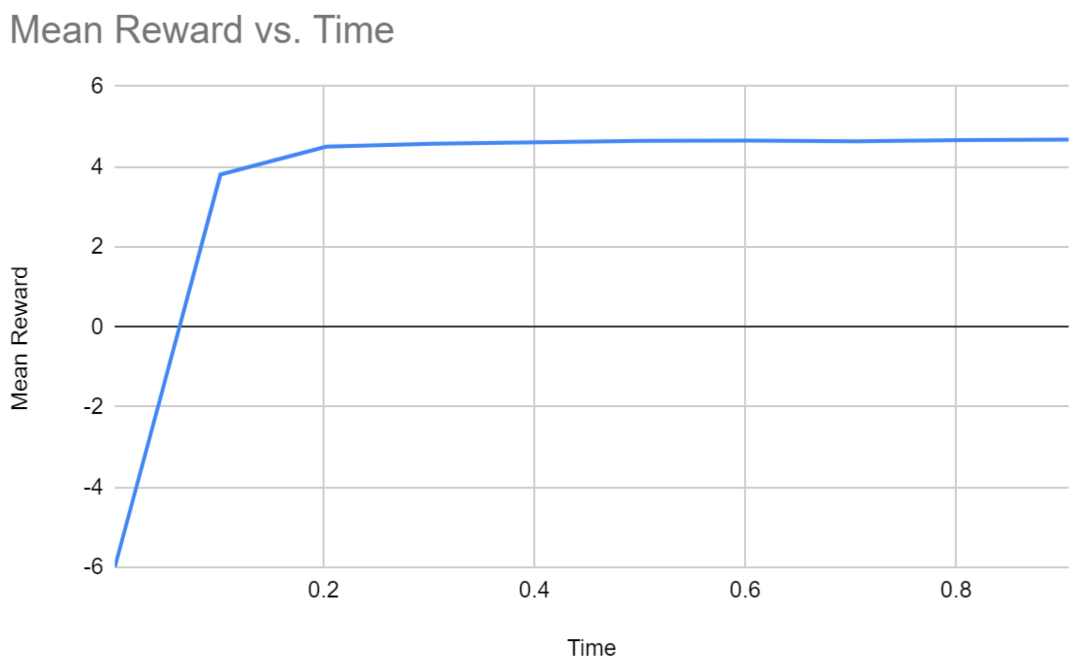
From the results, it can be said that as the epsilon value increases, the mean reward value decreases. This is understandable because I'm limiting the exploration and thus reducing the per action penalty of -1. Also, the decaying epsilon approach gave me a higher mean reward compared to the epsilon greedy approach even with a value of 0.01.

<u>For Epsilon-greedy approach</u>

Command to run: ***python part2.py 'grid_sample.txt' -1 0.9 1 0.7***

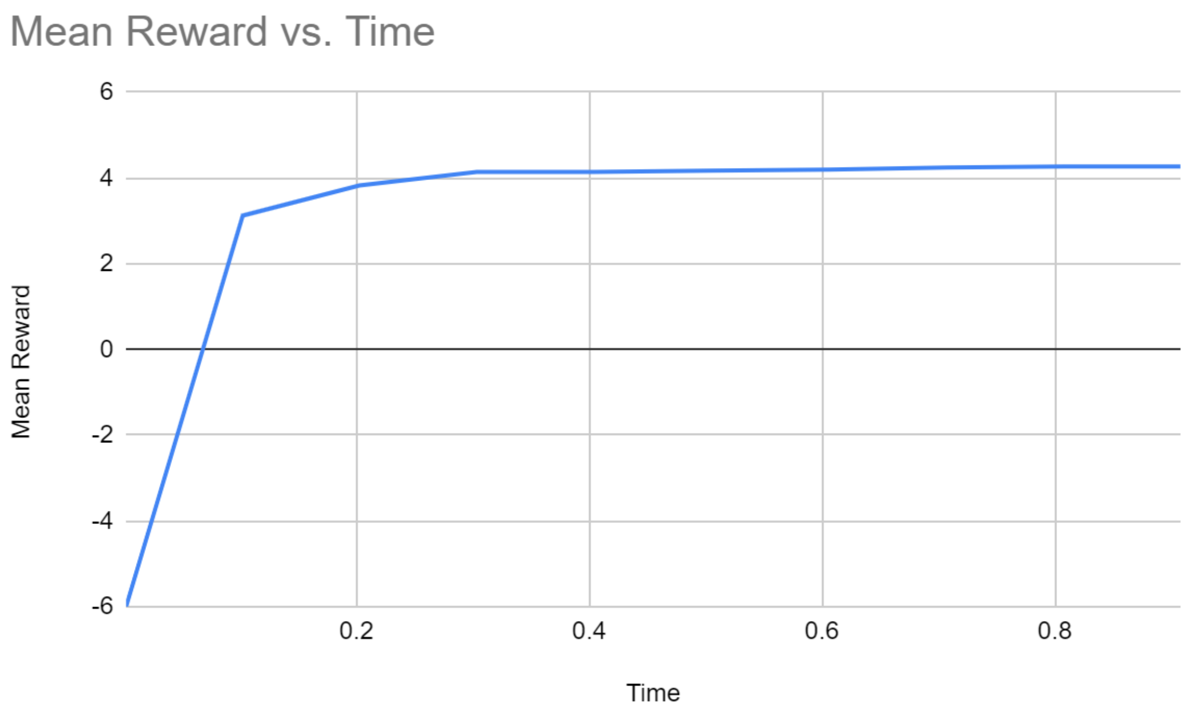
$\epsilon = 0.01$

Run	Mean Reward
1	4.595
2	4.551
3	4.539
4	4.612
5	4.469
6	4.588
7	4.602
8	4.526
9	4.658
10	4.591
Average	**4.5731**



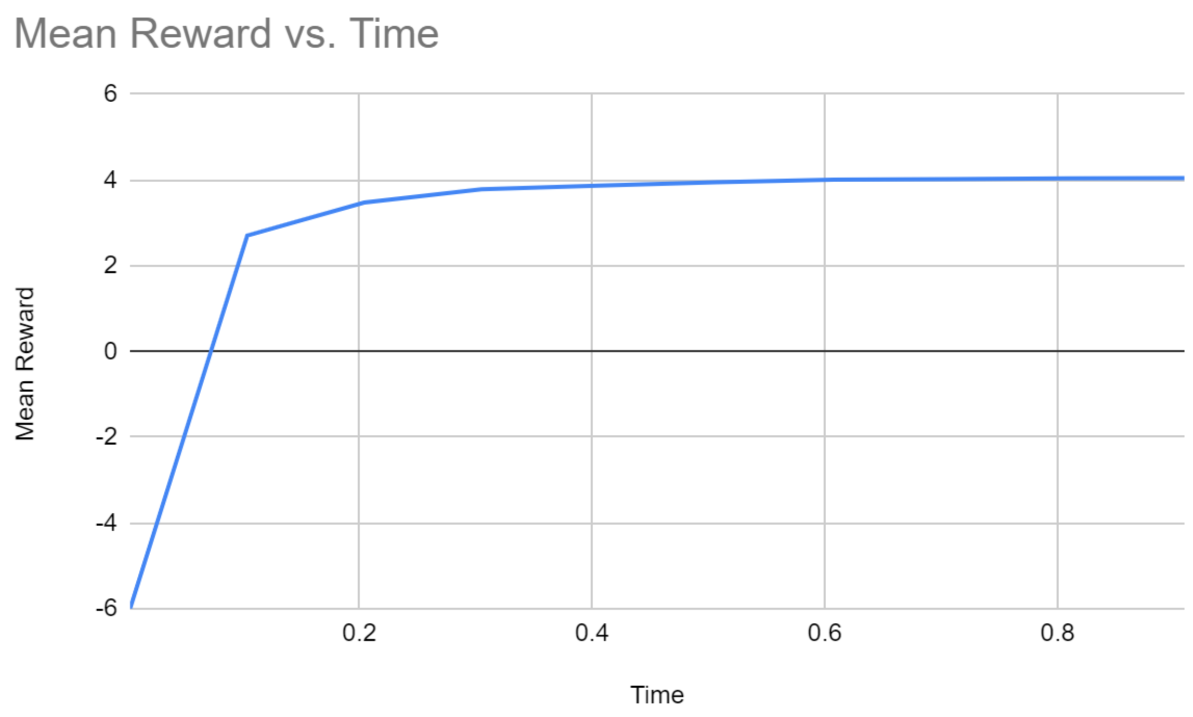
$\epsilon = 0.1$

Run	Mean Reward
1	4.505
2	4.453
3	4.469
4	4.525
5	4.497
6	4.323
7	4.415
8	4.438
9	4.393
10	4.405
Average	**4.4423**



$\epsilon = 0.3$

Run	Mean Reward
1	4.143
2	4.037
3	4.017
4	4.013
5	4.239
6	4.095
7	4.209
8	4.165
9	4.104
10	4.142
Average	**4.1164**

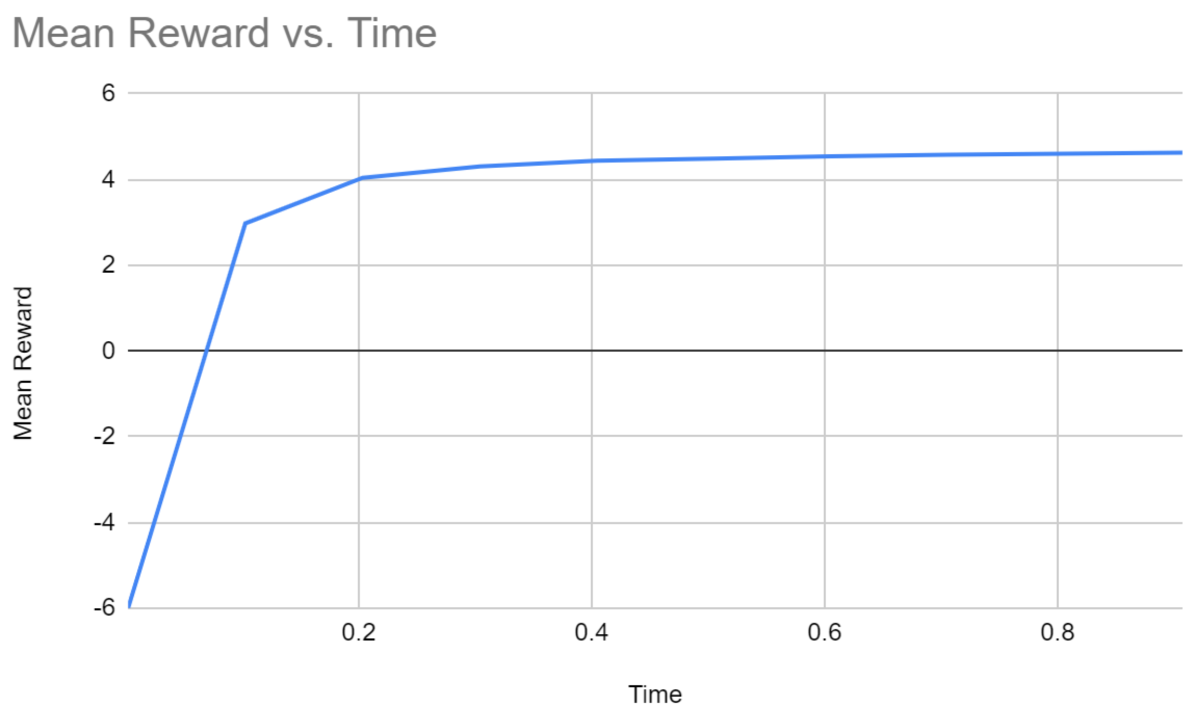


<u>For decaying epsilon approach</u>

Command to run: ***python part3.py 'grid_sample.txt' -1 0.9 1 0.7***

Run	Mean Reward
1	4.619
2	4.691
3	4.755
4	4.609
5	4.712
6	4.699
7	4.685
8	4.704
9	4.71
10	4.685

Average	**4.6869**



Parameter Optimization

Epsilon

In the action selection step, a specific action is selected based on the Q-values that are already there. The epsilon parameter introduces randomness into the algorithm, forcing us to try different actions. This helps not get stuck in a local optimum. If the epsilon is set to 0, we never explore but always exploit the knowledge we already have. On the contrary, having the epsilon set to 1, it forces the algorithm to always take random actions and never use past knowledge.

Since our algorithm does not use any decay rate and it only depends on the number of trials, there is no optimization required for this parameter.

Alpha

Alpha defines the learning rate or step size; this means that the Q-value for a given state is calculated by incrementing the old Q-value times alpha. Alpha is a number between 0 and 1 that allows the agent to learn from a previous action, if we set alpha close to zero the agent is not learning from previous experience, however if we set alpha close to 1, the agent ignores prior knowledge and only takes into account recent information.

Tuning the value for alpha: We set the other parameters as below and run the file *part3.py* to test the best value range for alpha-

reward_per_action = -1

gamma = 0.9

time_to_learn = 1

prob_of_moving = 0.7

Alpha	Mean Reward
0.1	4.58
0.2	4.65
0.3	4.73
0.4	4.68
0.5	4.68
0.6	4.55
0.7	4.43

For the 4x5 grid world that I generated all our results on, it is observed that the range of alpha is 0.2-0.5 for getting the best mean reward value for the grid.

Gamma

In Q-learning, gamma is multiplied by the estimation of the optimal future value. The next reward's importance is defined by the gamma parameter. If gamma is set to zero, the agent completely ignores the future rewards. Such agents only consider current rewards. On the other hand, if gamma is set to 1, the algorithm would look for high rewards in the long term.

Tuning the value for gamma: Setting up other parameters as follows-

reward_per_action = -1

alpha = 0.5

time_to_learn = 1

prob_of_moving = 0.7

Gamma	Mean Reward
0.1	4.61
0.2	4.63

0.3	4.62
0.4	4.64
0.5	4.62

Keeping the other values constant and changing the gamma values, I got pretty similar results as shown in the table. The value of gamma might come into picture when we are dealing with bigger boards with multiple treasures and pits. But for the board I tested, it can be concluded that the gamma value does not affect the results much.

Now using these values to solve a larger board of size 10x10 with a time of 20 secs and the reward per action as -0.01 (since the number of steps will increase drastically), the following results are acheived:

python part3.py 'grid_sample2.txt' -0.01 0.9 20 0.7

OUTPUT:

```
This program will read in grid_sample2.txt
It will run for 20 seconds.
Its decay rate is 0.5 and the reward per action is -0.01
Its transition model will move the agent properly with p = 0.7


GRID WORLD:
10   0   0   0   0   0   0   0   0   0
0    0   0   0   0   0   0   0   0   0
0    0   0   0   0   0   0   0   0   0
-5   0   0   0   0   0   0   0   0   0
0    0   0   X   0   0   0   0   0   0
0    0   0   0   0   0   0   0   0   0
0    0   0   0   0   0   0   0   0   0
0    0   0   X   0   0   0   0   0   0
0    0   0   0   0   0   0   0   0   0
S    0   0   0   0   0   0   0   0   0

GRID POLICY:
10   <   <   <   v   v   <   <   <   <
^    <   <   <   <   <   ^   ^   v   <
>    ^   ^   <   <   <   <   ^   <   ^
-5   ^   ^   ^   ^   <   <   <   ^   <
>    ^   ^   X   >   v   ^   ^   ^   <
```

```
v     ^     v     <     ^     v     <     <     v     >
>     ^     <     <     <     <     <     ^     v     ^
^     <     <     X     ^     >     ^     v     <     v
^     v     ^     >     v     ^     v     v     >     ^
^     <     v     >     ^     v     <     <     >     <

HEAT MAP:
10    1     0     0     0     0     0     0     0     0
7     8     2     0     0     0     0     0     0     0
1     7     0     0     0     0     0     0     0     0
-5    8     0     0     0     0     0     0     0     0
0     8     0     X     0     0     0     0     0     0
1     8     0     0     0     0     0     0     0     0
10    9     2     0     0     0     0     0     0     0
8     2     0     X     0     0     0     0     0     0
8     0     0     0     0     0     0     0     0     0
9     0     0     0     0     0     0     0     0     0

Mean reward per trail = 9.859
```

Extra Credit - Wormholes

I implemented the wormholes into the world taking into account the statements given in the assignment in which if the agent encountered a wormhole, it would tele transport into another position. This makes the learning process harder for the agent given that the exploration becomes more chaotic and the teletransportation gives a sense of restart because initially the new position would be completely unknown.

I used the deterministic model and greedy epsilon policy to develop the code for wormholes in a file *extra2.py*. Run the program for 10 secs with the following command-

```
python extra2.py 'grid_wormholes.txt' -1 0.9 10 0.7
```

OUTPUT:

```
This program will read in grid_wormholes.txt
It will run for 10 seconds.
Its decay rate is 0.9 and the reward per action is -1
Its transition model will move the agent properly with p = 0.7
```

```
GRID WORLD:
10   0   C   0   B   0   0   0   D
-5   0   0   X   0   C   E   0   0
0    B   0   0   0   0   0   0   0
0    0   0   0   0   0   0   0   0
0    0   0   0   0   0   0   0   0
0    0   0   0   0   0   0   0   0
S    0   0   0   D   0   0   0   0
0    0   0   0   0   0   0   0   0
0    0   E   0   0   0   0   0   0


GRID POLICY:
10   <   <   <   <   v   <   <   <
-5   ^   ^   X   >   >   <   ^   >
v    ^   <   >   ^   ^   >   <   <
<    v   ^   v   ^   ^   ^   ^   <
>    <   ^   ^   v   v   >   ^   <
v    ^   <   ^   <   >   ^   ^   <
>    v   v   <   v   ^   ^   ^   <
v    v   v   >   >   ^   v   v   <
>    >   ^   <   >   ^   ^   ^   <
HEAT MAP:
10   14  14  0   0   0   0   0   0
-5   0   0   X   0   0   14  0   0
0    0   0   0   0   0   0   0   0
0    0   0   0   0   0   0   0   0
0    0   0   0   0   0   0   0   0
1    0   0   0   0   0   0   0   0
15   13  0   0   0   0   0   0   0
1    13  0   0   0   0   0   0   0
1    14  0   0   0   0   0   0   0

Mean reward per trail = 3.845
```

<u>Analysis:</u>

A grid with 4 pairs of wormholes is created as shown in the output above. The agent here learns to make use of the wormholes to reach the treasure state with the least number of moves(which in turn reduces the penalty per action). The agent was using wormhole E in the bottom row to directly go to the second of the board. It was also using wormhole C to get into the top row and finally to the treasure.

Discussion

After analyzing the method presented, having a decay epsilon search has a better overall reward, even if a very low epsilon is considered, it does not match the average result for a decaying epsilon. This corroborates that having a balanced approach between exploring and exploiting is the best policy for this reinforcement learning example. On the other hand, now of tuning our parameters, it can be seen that the changes do not affect the reward outcome.

The SARSA algorithm is also used to solve a particular board to decide whether Q-learning or SARSA is better for our problem statement. For the non-deterministic model with the decaying epsilon approach, the mean reward achieved is **4.68** for the Q-learning algorithm. With the same time and the same parameters, the SARSA algorithm is used, and the mean reward came out to be **4.1** approximately as the average for 10 trials. Even for the Epsilon greedy approach, a better mean reward value is achieved for each of the different values of Epsilon.

So, it can be concluded that Q-learning was a better approach for solving the given problem statement.