Mahyar Mohammadi Matin – 610398166 – HW1

1.

a) What are the number of model states depends on? It depends on inputted table(environment) size. Each state corresponds to a unique position of the agent in the environment, and can be represented as a tuple of the agent's x and y coordinates in the grid. If table has height of n and length of m then we have n*m states.

b) Can we reduce number of states? Yes of course!

> - we can remove any sticked same states, since when we enter one of them it is not different how we exit them. For example a 3*3 white path can be reduce to just 1 state that has access to 12 different state for both entry and exit.

2.

States: The states in the problem correspond to the different positions of the agent in the environment. Each state can be represented by a tuple of the agent's x and y coordinates in the grid.

Actions: The actions in this problem are the movements that the agent can make in the environment. The actions can be defined as "up", "down", "left", and "right".

Rewards: The rewards in this problem can be defined based on the agent's progress towards the goal state. A positive reward can be given when the agent successfully passes a flag, and a large penalty can be given when the agent encounters a wall. The penalty can also be set to small negative point for other states that do not correspond to reaching a flag or encountering a wall since any extra move that not leading to any flag is not accepted.

Goal State: The goal state in this problem is the end point that the agent needs to reach, represented by the letter "T". The agent's goal is to reach the end point by passing all the flags in the way and avoiding the walls. The agent's final state will be the end point when the goal is reached, and the episode will end.

3,4.

In general, the learning rate (α) determines the extent to which the new information is incorporated into the Q-value for a given state-action pair. It controls the balance between exploration and exploitation in the algorithm.

A high α value means that the new information is given more weight and the Q-values are updated more rapidly, allowing the algorithm to learn more quickly. On the other hand, a low α value means that the Q-values are updated more slowly, allowing the algorithm to consider more historical information before making updates.

Here are some of the impacts of changing α in problem:

1. Speed of convergence: A high α value can speed up the convergence of the algorithm, but may result in overshooting the optimal solution and oscillating around it. A low α value can slow down convergence, but can lead to a more accurate solution.
2. Exploration vs. exploitation: A high α value encourages exploration, allowing the agent to try out new actions and potentially discover better solutions. A low α value promotes exploitation, allowing the agent to rely on the existing information and make more confident decisions.
3. Stabilization: A low α value can help stabilize the learning process and reduce the risk of overshooting the optimal solution.

Therefore, choosing the right value for α is a trade-off between the speed of convergence, the balance between exploration and exploitation, and the stability of the learning process. In general, a smaller α value is used at the beginning of the training process to allow for more exploration, and a larger α value is used later on to promote exploitation and speed up convergence.

The discount factor (γ) determines the importance of future rewards in the calculation of Q-values. It decides the amount of weight that should be given to future rewards versus immediate rewards.

Here are some of the impacts of changing γ in problem:

1. Long-term vs. short-term rewards: A high γ value means that future rewards are given more weight in the calculation of Q-values, encouraging the agent to make decisions that lead to long-term rewards. A low γ value gives less importance to future rewards and emphasizes immediate rewards, leading to a focus on short-term goals.
2. Optimal policy: The value of γ influences the shape of the optimal policy, as it determines the balance between exploration and exploitation. A high γ value can encourage the agent to take actions that lead to long-term rewards, even if they are not the most immediate rewards, while a low γ value will lead to a focus on immediate rewards.
3. Convergence: A high γ value can slow down the convergence of the algorithm, as the agent continues to explore and make updates to its Q-values even when it is close to the optimal solution. A low γ value can speed up the convergence of the algorithm, as the agent becomes less focused on exploration and more focused on exploitation.
4. Temporal consistency: γ determines the temporal consistency of the Q-values, as it governs the decay of the value of future rewards over time. A high γ value leads to more consistency, as the value of future rewards decreases slowly over time, while a low γ value leads to less consistency, as the value of future rewards decreases rapidly.

In general, choosing the right value for γ is a trade-off between emphasizing short-term rewards and emphasizing long-term rewards, as well as between exploration and

exploitation. A value of γ close to 1 emphasizes long-term rewards and exploration, while a value close to 0 emphasizes short-term rewards and exploitation.

Code Explanation:

Class Q_table

This class written for handling all things about a Q_table.

```python
class Q_table:
    def __init__(self,environment,actions=['L','R','U','D']):
        self.Q = {}
        self.actions = actions
        self.temp_env = environment
        for state in environment.states:
            for action in self.actions:
                self.Q[(state, action)] = 0

    def get(self,state,action):
        return self.Q[(state, action)]

    def set(self,state,action,data):
        self.Q[(state, action)] = data

    def best_move(self,environment,state):
        max_q = -1000000
        action_q = None
        for a in environment.valid_actions(state):
            q = self.get(state, a)
            if q > max_q:
                max_q = q
                action_q = a
        return action_q,max_q

    def display(self):
        # Print the final Q-table
        print("Q-table:")
        for state in self.temp_env.states:
            for action in self.actions:
                print("Q[{}, {}] = {}".format(state, action, Q[(state, action)]))
```

Constructor: First it's initialize with environment object and valid moves. It build a dictionary from (state,action) to it's reward.

Get/set: for get and set data to Q_table.

Best_move: choose best move in some inputted state considering the maximum reward that can agent get.

Display: just for printing Q table.

Draw: Draw corresponding directed-weighted graph of q-table(That was a hard one because of the edge conflict)

Class Environment:

All functions regarding our environment handled here.

```
class Environment:
    def __init__(self,table,reward_map):
        self.table = table
        self.size = np.array(table).shape
        self.reward_map = reward_map

        self.states = []
        self.fill_states()

    def fill_states(self):
        for i in range(self.size[0]):
            for j in range(self.size[1]):
                self.states.append((i, j))

    def get_state(self,state):
        return self.table[state[0]][state[1]]

    def set_state(self,state,data):
        self.table[state[0]][state[1]] = data

    def valid_actions(self,state):
        valid_actions = []
        if state[1] > 0 and self.table[state[0]][state[1]-1] != 'B':
            valid_actions.append('L')
        if state[1] < len(self.table[0])-1 and self.table[state[0]][state[1]+1] != 'B':
            valid_actions.append('R')
        if state[0] > 0 and self.table[state[0]-1][state[1]] != 'B':
            valid_actions.append('U')
        if state[0] < len(self.table)-1 and self.table[state[0]+1][state[1]] != 'B':
            valid_actions.append('D')
        return valid_actions

    def get_start_state(self):
        # Define the agent's starting position
        for state in self.states:
            if self.table[state[0]][state[1]] == 'A':
                return state
        print("Environment don't have any start state.")
        return None
```

Constructor: gets table and reward map. Also build all possible states.

get_state/set_state: for changing in environment.

valid_actions: return possible moves(we cant go outside of the board or blocked)

Class Q_learning

This class developed like other machine learning libraries with same functions.

```python
class Q_learning:
    def __init__(self, alpha, gamma, epsilon,episode_count):
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.episode_count = episode_count
        self.Q = None

    def fit(self, environment):
        # Start the Q-Learning algorithm
        Q = Q_table(environment)
        epsilon_decrease = self.epsilon/self.episode_count
        epsilon = self.epsilon
        for episode in range(self.episode_count):
            #if episode%100==0:
            #    print(f'Episode: {episode}/{self.episode_count}')
            state = environment.get_start_state()
            cur_env = environment.copy()
            target_seen = False
            episode_len = 0
            env_size = cur_env.size[0]*cur_env.size[1]
            epsilon = epsilon - epsilon_decrease
            while (episode_len<env_size or not target_seen) and episode_len<1.5*env_size:
                episode_len+=1
                # Choose an action
                valid_actions = cur_env.valid_actions(state)
                if np.random.uniform(0, 1) < epsilon:
                    action = np.random.choice(valid_actions)
                else:
                    action,_ = Q.best_move(cur_env,state)

                # Take the action and get the new state and reward
                new_state = cur_env.move(state,action)
                reward = cur_env.get_reward(new_state)
```

Constructor: create instance by inputting Q_learning basic parameters and episode_count as iteration limit for number of episodes.

Fit: get an environment object and fill new Q_table object for that environment with given parameters in previous section.

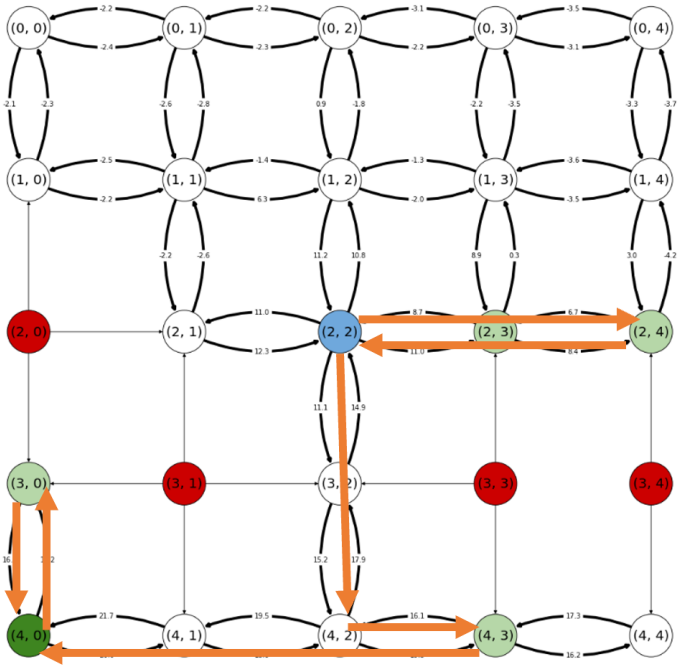
Get_path: with respect to Q_table that fitted to our environment, this function run many agents to find the best way to solve the problem. Also in some points agent maybe go in a loop that is not an efficient way. So helper function delete_loop, remove any loop in the best agents way.

Test model for simple example:

```
table = [
    ['W', 'W', 'W', 'W', 'W'],
    ['W', 'W', 'W', 'W', 'W'],
    ['B', 'W', 'A', 'F', 'F'],
    ['F', 'B', 'W', 'B', 'B'],
    ['T', 'W', 'W', 'F', 'W']]
reward = {'B':-100,'F':30,'T':100,'W':-3,'A':-3}

environment = Environment(table,reward)
model = Q_learning(alpha=0.1, gamma=0.9, epsilon=0.3,episode_count=1000)
Q = model.fit(environment)
Q.draw()
```

```
Episode: 0/1000
Episode: 100/1000
Episode: 200/1000
Episode: 300/1000
Episode: 400/1000
Episode: 500/1000
Episode: 600/1000
Episode: 700/1000
Episode: 800/1000
Episode: 900/1000
```

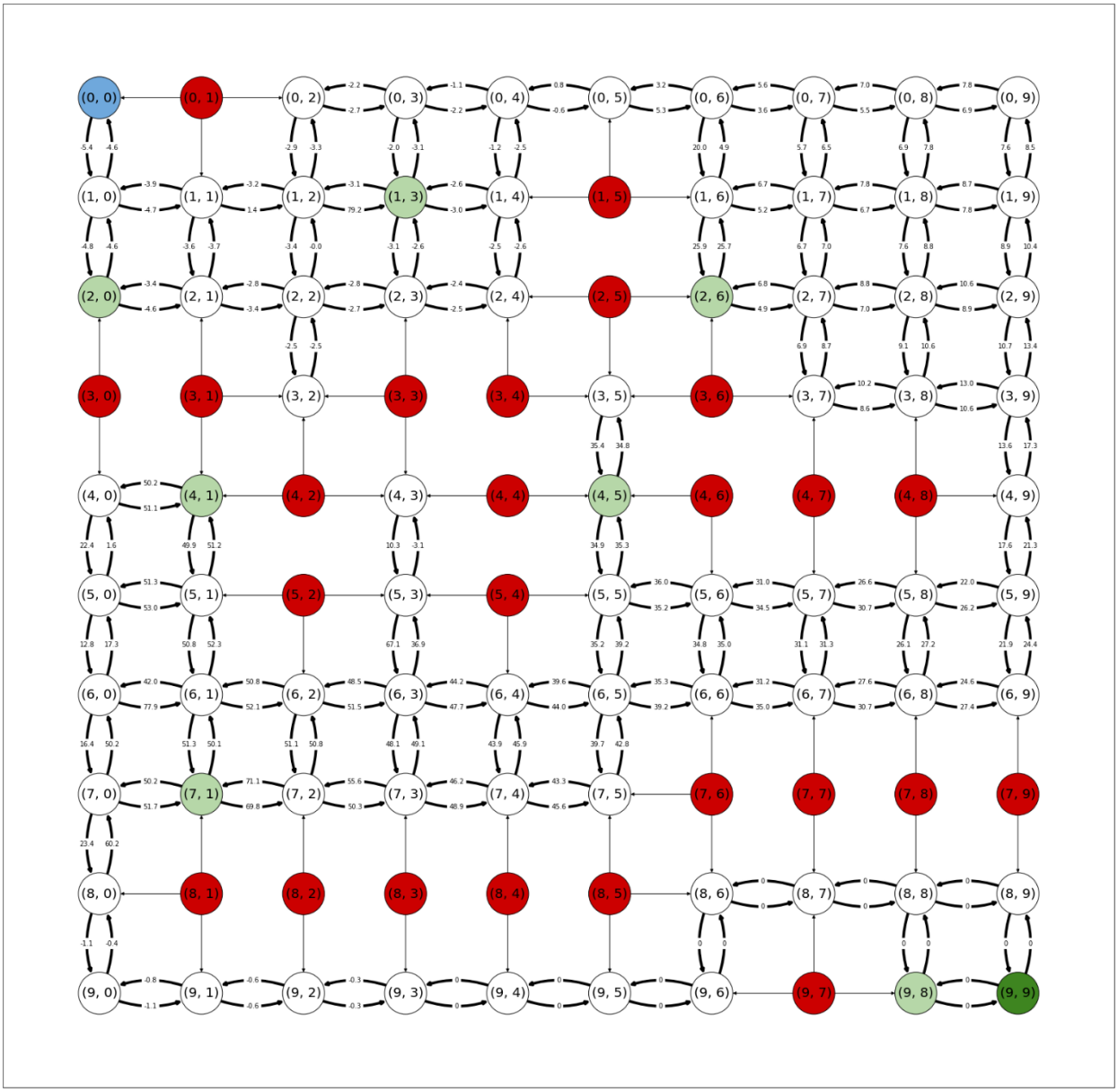


Best way:

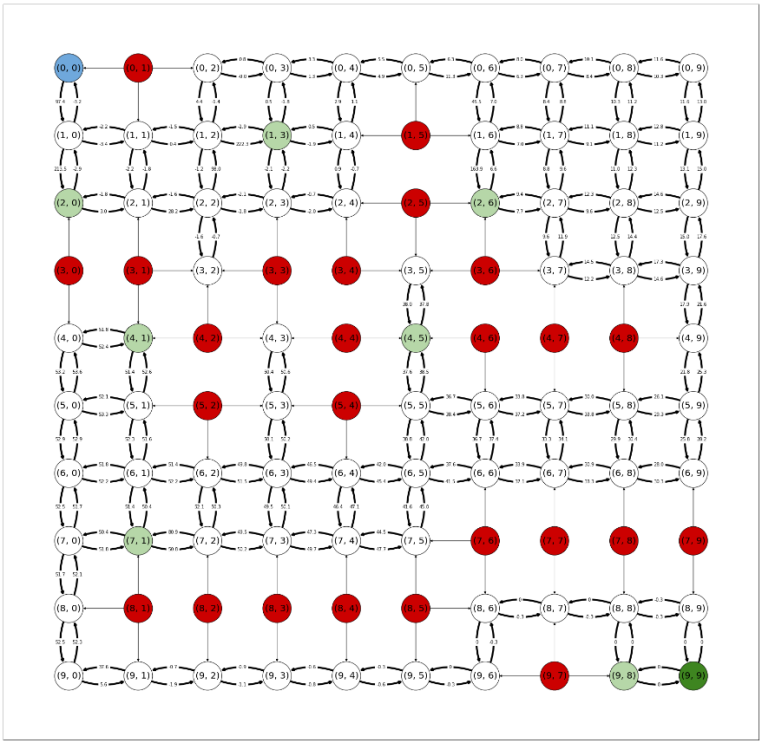
```
[((2, 2), 'R'),((2, 3), 'R'),((2, 4), 'L'),((2, 3), 'L'),((2, 2), 'D'),((3, 2), 'D'),
((4, 2), 'R'),((4, 3), 'L'),((4, 2), 'L'),((4, 1), 'L'),((4, 0), 'U'),((3, 0), 'D')]
```

Now we find out that our code work correctly. Let's look at some Q-tables with different γ and α for our main problem:

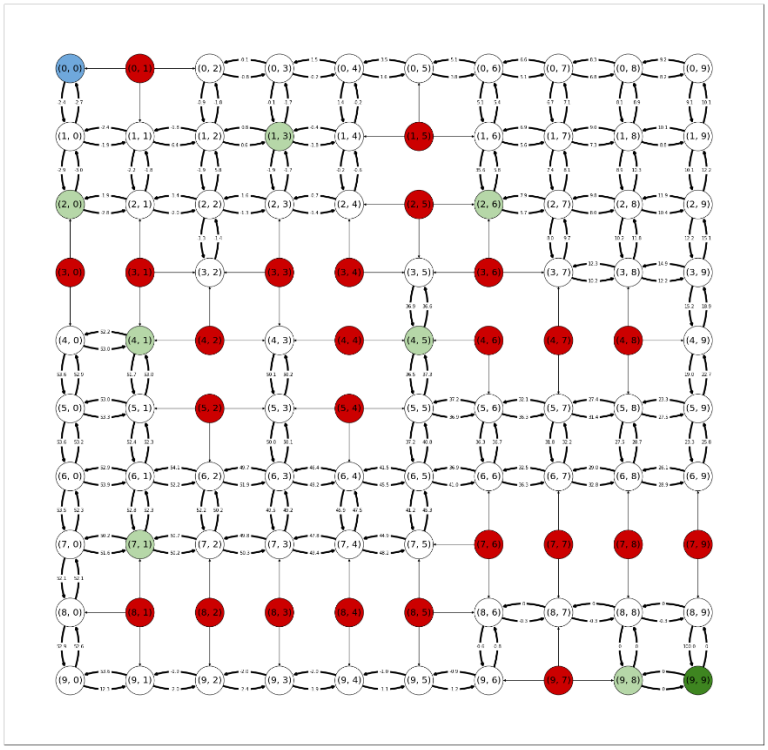
gamma:0.25,alpha:0.1



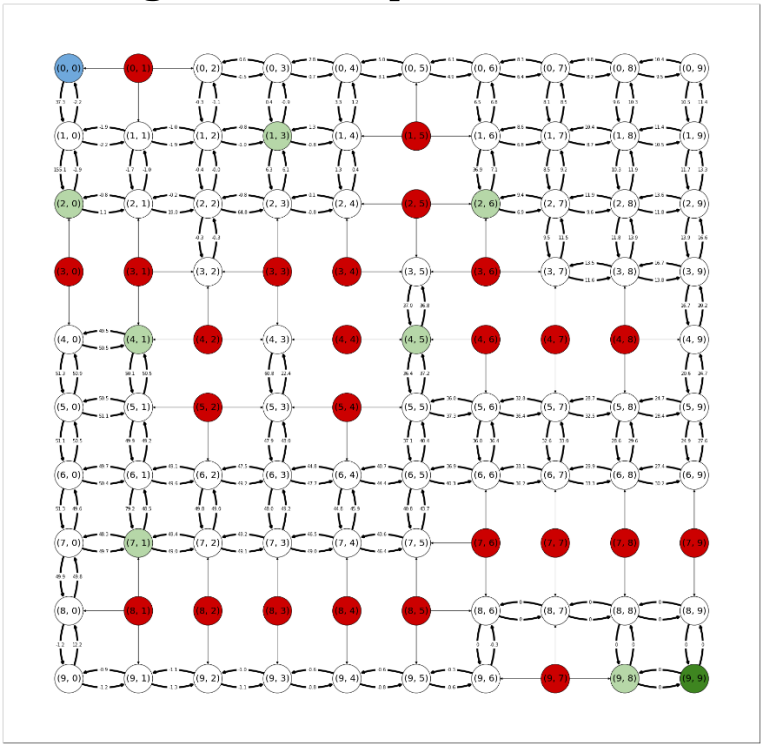
gamma:0.25,alpha:0.5



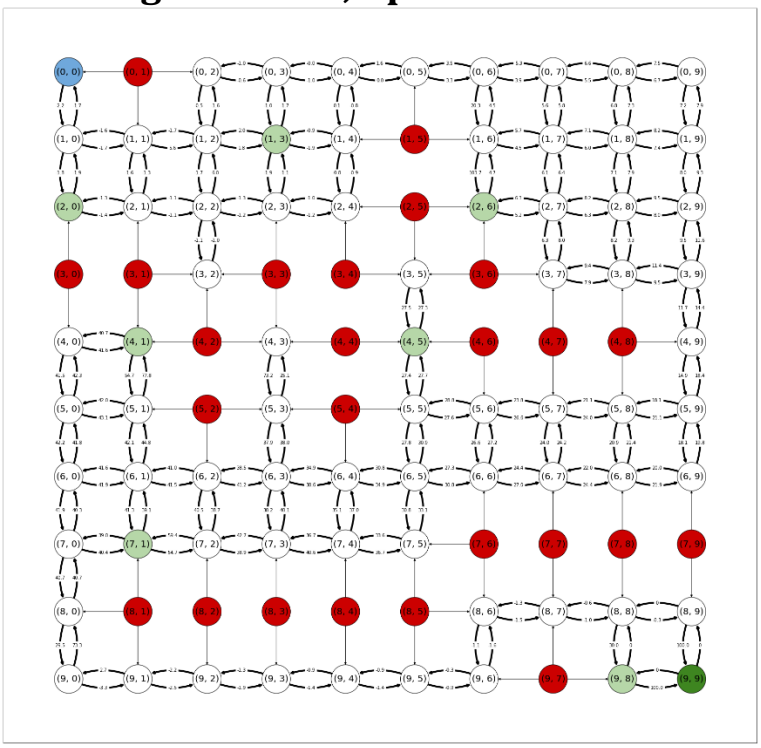
gamma:0.25,alpha:1



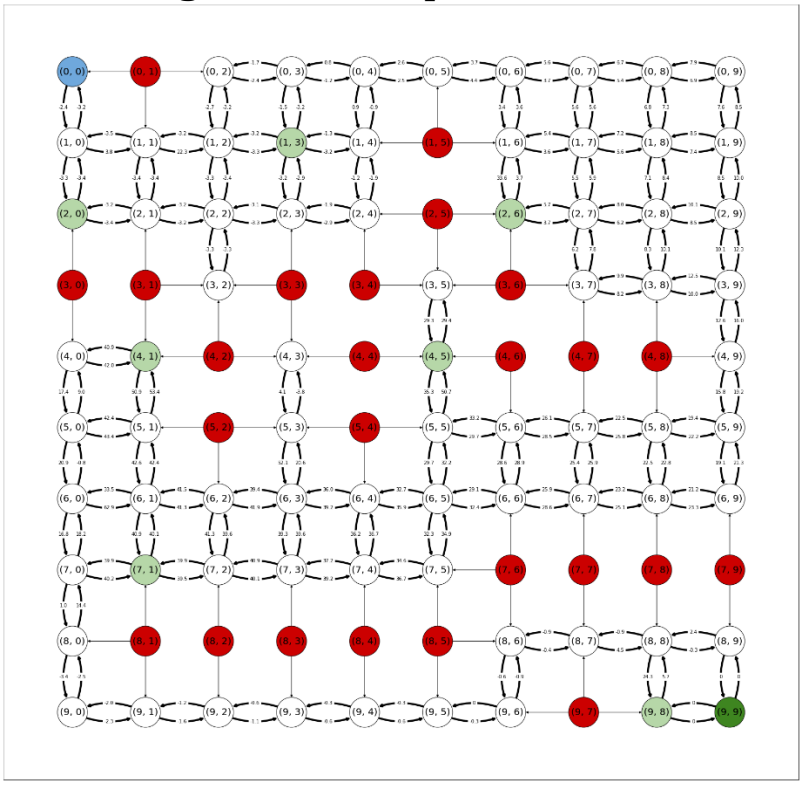
gamma:0.5,alpha:0.1



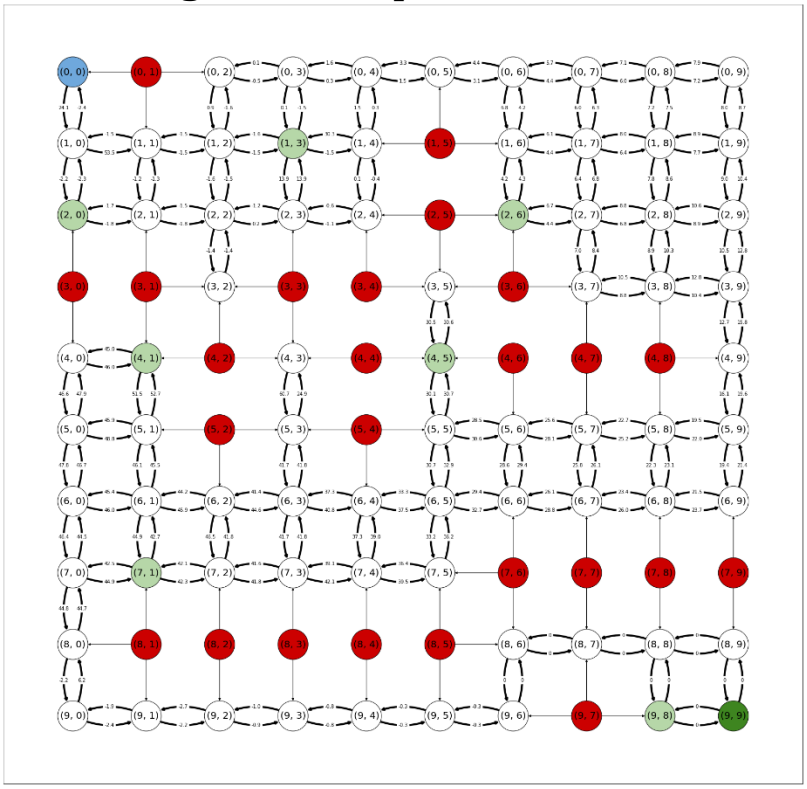
gamma:0.5,alpha:0.5



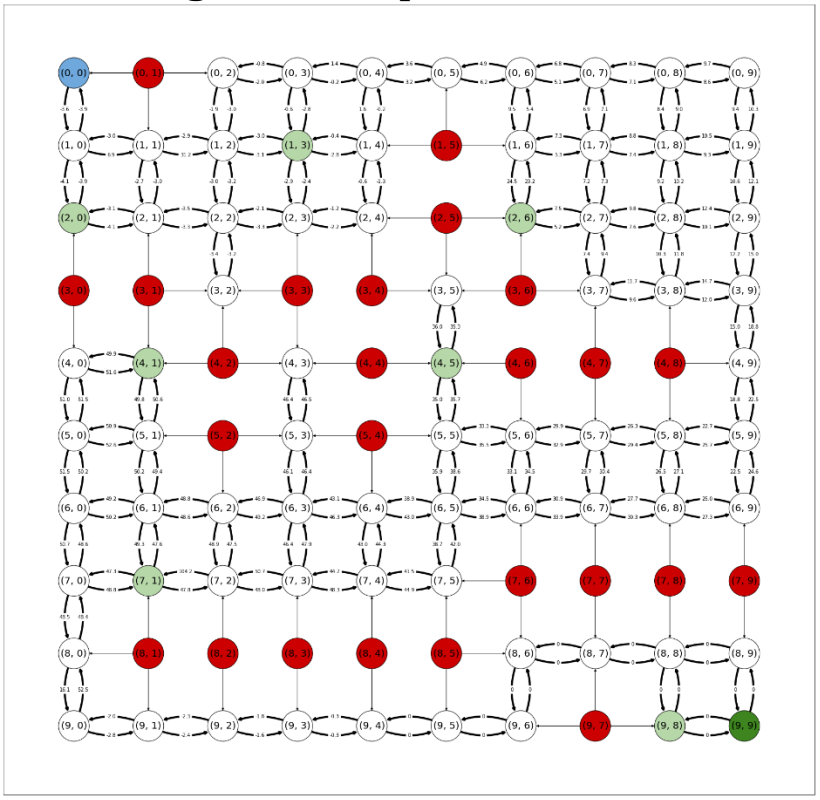
gamma:0.5,alpha:1



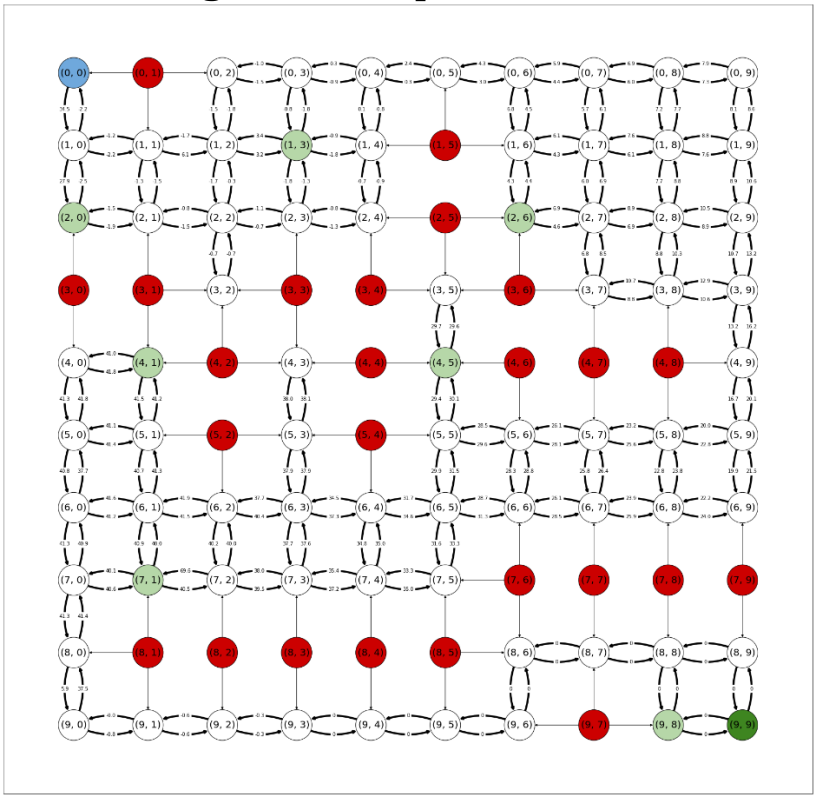
gamma:1,alpha:0.1



gamma:1,alpha:0.5



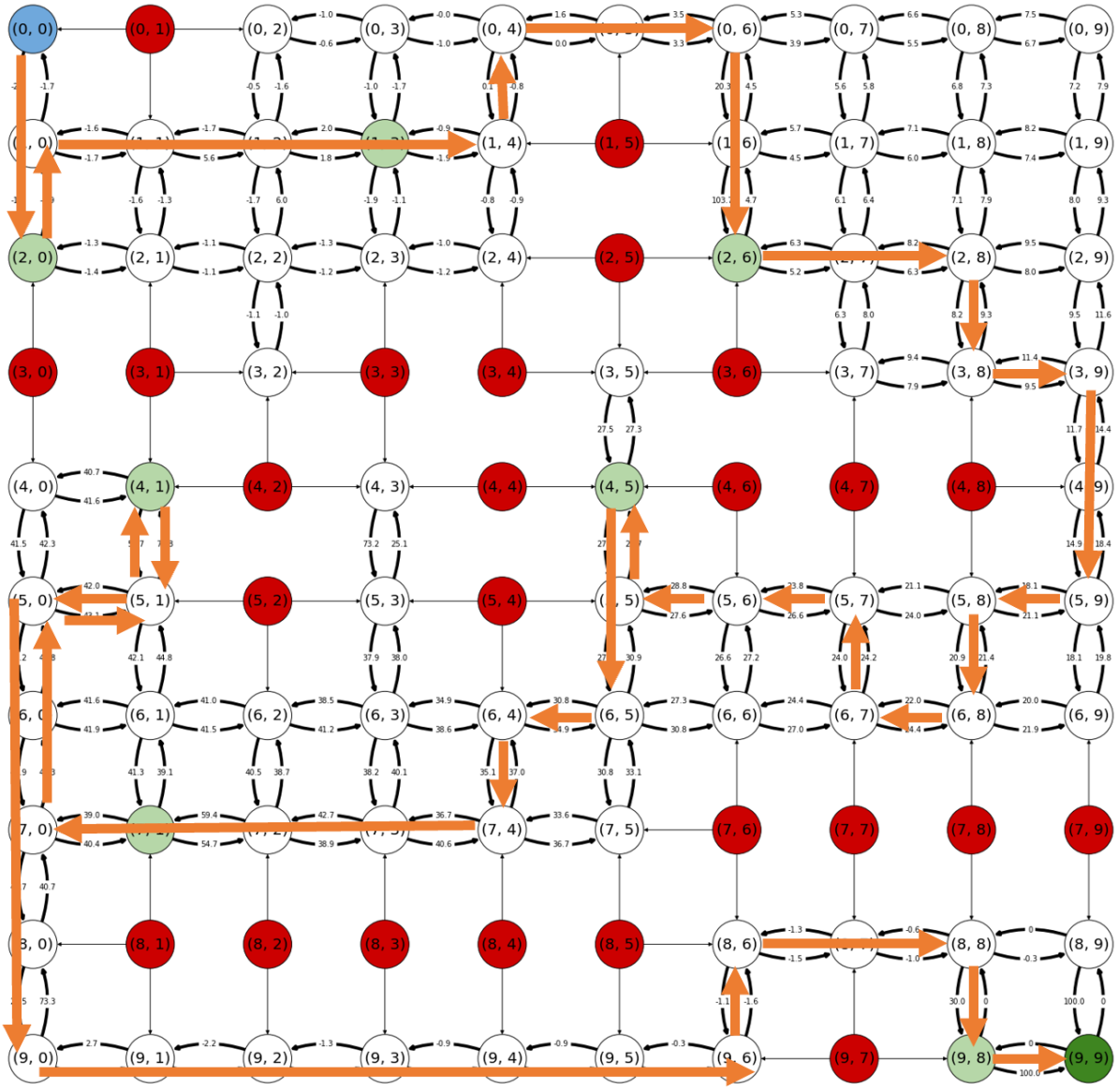
gamma:1,alpha:1



6.

It seems that alpha=gamma=0.5 gave us best Q_table. Let's call get_path on this model to see the result path:

```
[((0, 0), 'D'), ((1, 0), 'D'), ((2, 0), 'U'), ((1, 0), 'R'), ((1, 1), 'R'), ((1, 2), 'R'),
((1, 3), 'R'), ((1, 4), 'U'), ((0, 4), 'R'), ((0, 5), 'R'), ((0, 6), 'D'), ((1, 6), 'D'), (
(2, 6), 'R'), ((2, 7), 'R'), ((2, 8), 'D'), ((3, 8), 'R'), ((3, 9), 'D'), ((4, 9), 'D'), ((
5, 9), 'L'), ((5, 8), 'D'), ((6, 8), 'L'), ((6, 7), 'U'), ((5, 7), 'L'), ((5, 6), 'L'), ((5
, 5), 'U'), ((4, 5), 'D'), ((5, 5), 'D'), ((6, 5), 'L'), ((6, 4), 'D'), ((7, 4), 'L'), ((7,
3), 'L'), ((7, 2), 'L'), ((7, 1), 'L'), ((7, 0), 'U'), ((6, 0), 'U'), ((5, 0), 'R'), ((5, 1
), 'U'), ((4, 1), 'D'), ((5, 1), 'L'), ((5, 0), 'D'), ((6, 0), 'D'), ((7, 0), 'D'), ((8, 0)
, 'D'), ((9, 0), 'R'), ((9, 1), 'R'), ((9, 2), 'R'), ((9, 3), 'R'), ((9, 4), 'R'), ((9, 5),
'R'), ((9, 6), 'U'), ((8, 6), 'R'), ((8, 7), 'R'), ((8, 8), 'D'), ((9, 8), 'R')]
```



5.

Also we can show Q_table in this form:

	L	R	U	D
(0, 0)	0.00	0.00	0.00	-1.00
(0, 1)	0.00	0.00	0.00	0.00
(0, 2)	0.00	-1.54	0.00	-2.06
(0, 3)	-2.19	-0.20	0.00	-2.49
(0, 4)	-0.77	1.25	0.00	-0.64
(0, 5)	1.35	6.08	0.00	0.00
(0, 6)	2.88	2.81	0.00	6.70
(0, 7)	4.16	4.58	0.00	4.37
(0, 8)	6.11	6.05	0.00	5.93
(0, 9)	6.93	0.00	0.00	7.09
(1, 0)	0.00	-2.97	-2.99	5.00
(1, 1)	-0.93	-3.58	0.00	-2.83
(1, 2)	-2.40	0.20	-2.41	-2.37
(1, 3)	-2.92	48.25	10.30	-2.16
(1, 4)	56.00	0.00	-1.90	-1.75
(1, 5)	0.00	0.00	0.00	0.00
(1, 6)	0.00	3.13	3.08	4.00
(1, 7)	4.53	4.57	4.30	4.66
(1, 8)	6.25	6.54	6.34	6.54
(1, 9)	7.82	0.00	8.12	8.15
(2, 0)	0.00	-0.70	-2.95	0.00

Bonos(Moving object):

In this scenario, the environment includes a moving object that the agent can interact with. Hitting the object can change the environment, potentially opening up new paths for the agent or closing off existing ones.

One solution to this problem would be to incorporate the state of the moving object into the state representation. Each state would then consist of the agent's position as well as the position of the moving object.

The actions that the agent can take in this scenario could include hitting the moving object and moving in the corresponding direction, as well as the standard "up", "down", "left", and "right" movements.

Rewards can be assigned based on the effect of hitting the moving object on the agent's progress towards the goal. For example, a positive reward can be given when hitting the moving object opens up a new path towards the goal, while a negative reward can be given when hitting the object closes off an existing path.

The Q-Learning algorithm can then be used to learn the optimal policy for the agent, taking into account the state of the moving object and the rewards associated with different actions. The goal state in this scenario would be the same as before, reaching the end point represented by the letter "T".

This approach can help the agent learn to make decisions that take into account the effects of hitting the moving object on its progress towards the goal, allowing it to find the best path to the end point even when the environment is dynamically changing.