732A96 - Advanced Machine Learning - Lab 3 - Group Report

Qinyuan Qi(qinqi464)

2024-10-16

Contribution:

Qinyuan Qi did all the work in this report.

Q-Learning

1. Function Implementation

The following are the implementations of GreedyPolicy and EpsilonGreedyPolicy function.

```r
GreedyPolicy <- function(x, y){
  # Get a greedy action for state (x,y) from q_table.
  # Args:
  #   x, y: state coordinates.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.
  #

  # Find the actions that corresponding to the highest Q-values at the state
  # and choose one randomly if there are more than one suitable actions.
  actions <- which(q_table[x,y,] == max(q_table[x,y,]))

  # must check like this, directly sample(actions, size = 1) will have error
  if(length(actions) > 1){
    return(sample(actions, size = 1))
  }
  return (actions)
}
```

```r
EpsilonGreedyPolicy <- function(x, y, epsilon){
  # Get an epsilon-greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   epsilon: probability of acting randomly.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # if random number < epsilon then do random action
  if (runif(1) < epsilon)
```

```r
    return (sample(1:4, size = 1))
  else
    return (GreedyPolicy(x, y))
}
```

The following is the implementation of q_learning function.

```r
q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                       beta = 0){

  # Perform one episode of Q-learning. The agent should move around in the
  # environment using the given transition model and update the Q-table.
  # The episode ends when the agent reaches a terminal state.
  #
  # Args:
  #   start_state: array with two entries, describing the starting position of the agent.
  #   epsilon (optional): probability of acting randomly.
  #   alpha (optional): learning rate.
  #   gamma (optional): discount factor.
  #   beta (optional): slipping factor.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   reward: reward received in the episode.
  #   correction: sum of the temporal difference correction terms over the episode.
  #   q_table (global variable): Recall that R passes arguments by value. So, q_table being
  #   a global variable can be modified with the superassigment operator <<-.

  # init current position
  current_position <- start_state
  episode_correction <- 0

  repeat{
    # Follow policy, execute action, get reward.
    # choose A from S using policy derived from Q (e.g., epsilon-greedy)
    action <- EpsilonGreedyPolicy(current_position[1], current_position[2], epsilon)
    # get new position
    new_position <- transition_model(current_position[1], current_position[2], action, beta)
    # get new reward
    reward <- reward_map[new_position[1], new_position[2]]
    # get current Q-value
    Q <- q_table[current_position[1], current_position[2], action]

    # update formula:Page 13 of RL_L1_QLearning.pdf
    # q_{*}(s,a) <- q_{*}(s,a) + alpha * (r + gamma * max_{a'} q_{*}(s',a') - q_{*}(s,a))
    correction_term <- reward + gamma * max(q_table[new_position[1],
                                            new_position[2],]) - Q

    # accumulates the sum of all difference correction terms to to plot later
    # it can be removed if not needed
    episode_correction <- episode_correction + correction_term

    # Q-table update.
    q_table[current_position[1], current_position[2], action] <<- Q + alpha * correction_term

    current_position <- new_position

    if(reward!=0)
      # End episode.
      return (c(reward,episode_correction))
  }
}
```

2. Environment A (learning)

Question

- What has the agent learned after the first 10 episodes ?
- Is the final greedy policy (after 10000 episodes) optimal for all states, i.e. not only for the initial state ? Why / Why not ?
- Do the learned values in the Q-table reflect the fact that there are multiple paths (above and below the negative rewards) to get to the positive reward ? If not, what could be done to make it happen ?

Answer

In environment A, the start state is (x=3,y=1), parameters are set as epsilon = 0.5 , alpha = 0.1 ,gamma = 0.95 , beta = 0.

After first 10 episodes, the agent has learned the following policies:

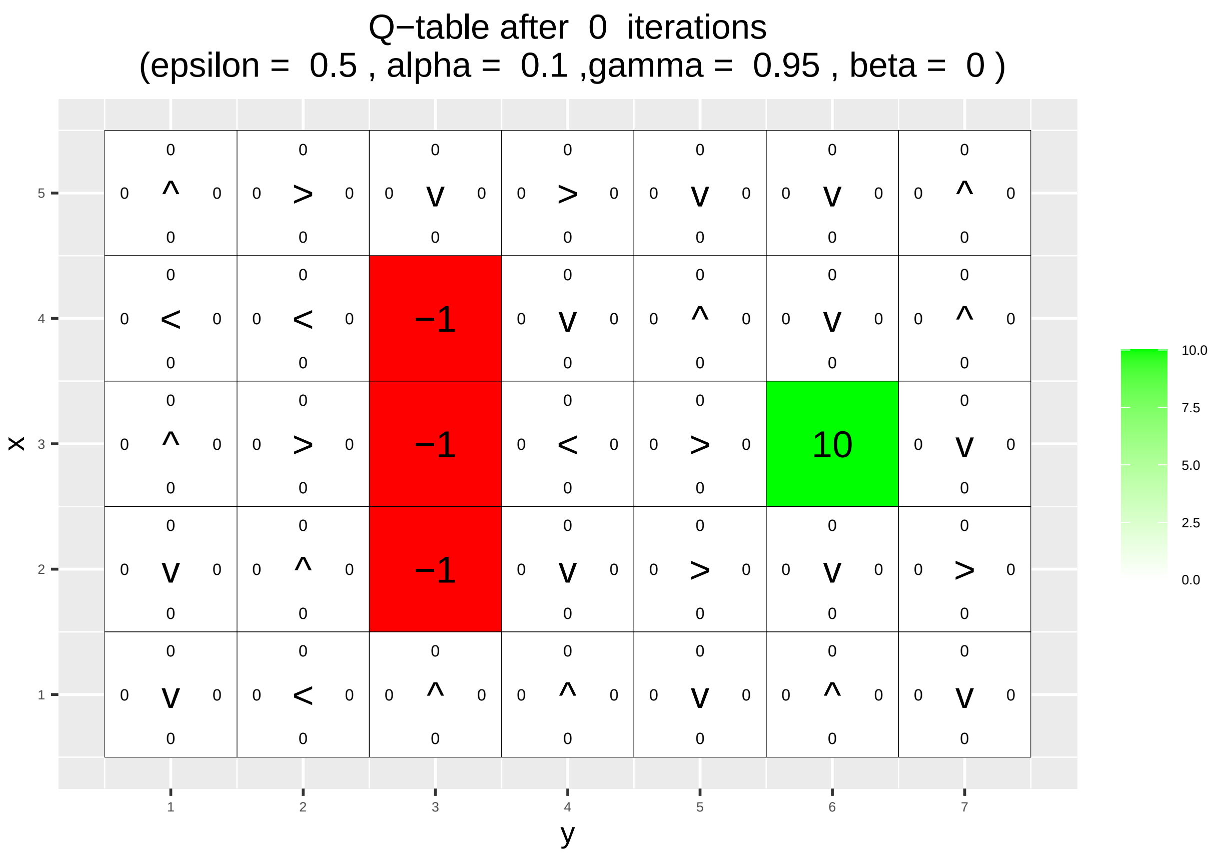
- We found that some of the blocks near red negative blocks began to show negative q-value, this means the agent has already learned to avoid the negative states.
- The agent has learned to move to the right to get the positive reward (+10).(Because of the randomness, this phenomena may not show every time when we recompile the Rmd)

After 10000 episodes, we can see that greedy policy is not optimal for all states. The sample states are start state and (5,7) (Those points may change). The reason why this happens is because the discount factor gamma is set to 0.95, which means the agent will place significant value on future rewards when making decisions, rather than just focusing on short-term gains. Since the optimal policy is aimed at maximizing long-term rewards, more iterations may be needed for the Q-values of all states to converge to their optimal values, ensuring that the greedy policy is optimal across all states.

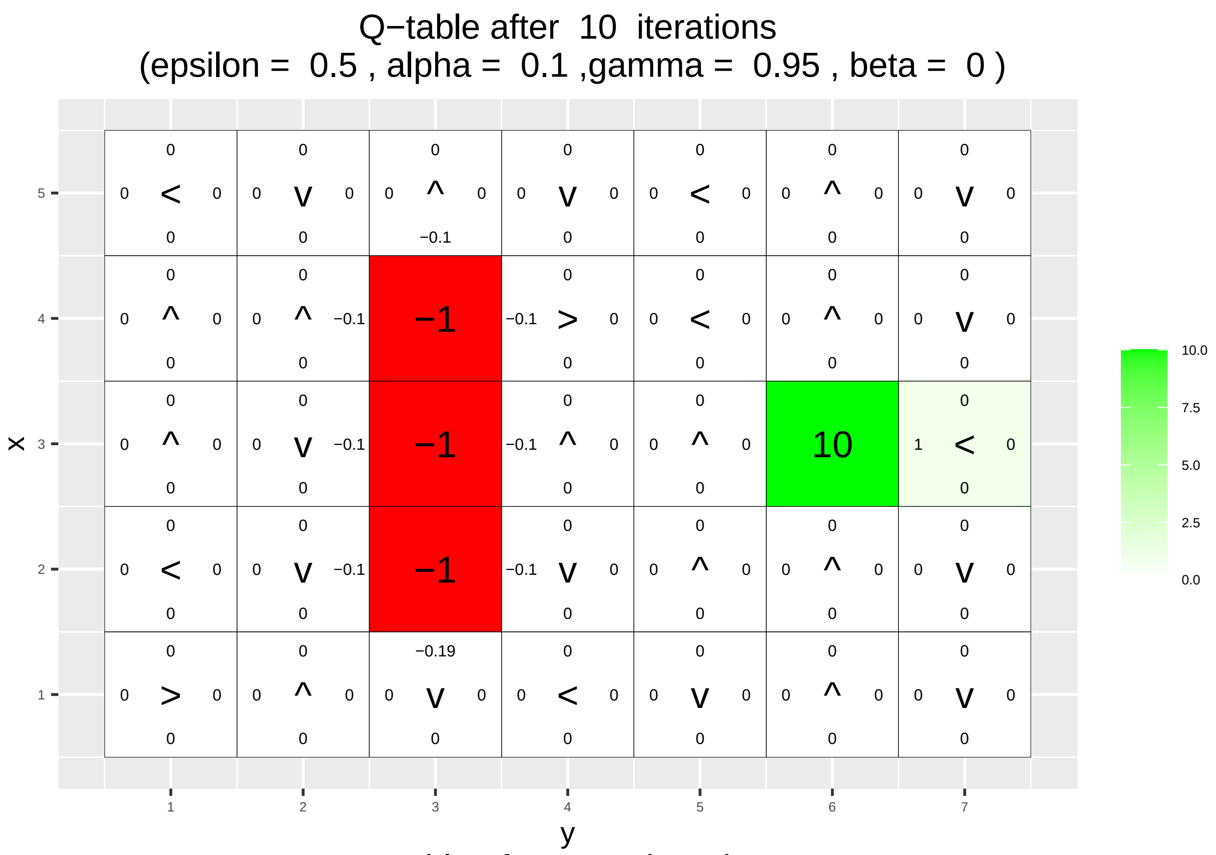
Regarding multiple paths question that the agent learned, the answer is NO. By checking the values and arrows in the Q-table, there is only one clear path to the final state from the start state above the negative reward.

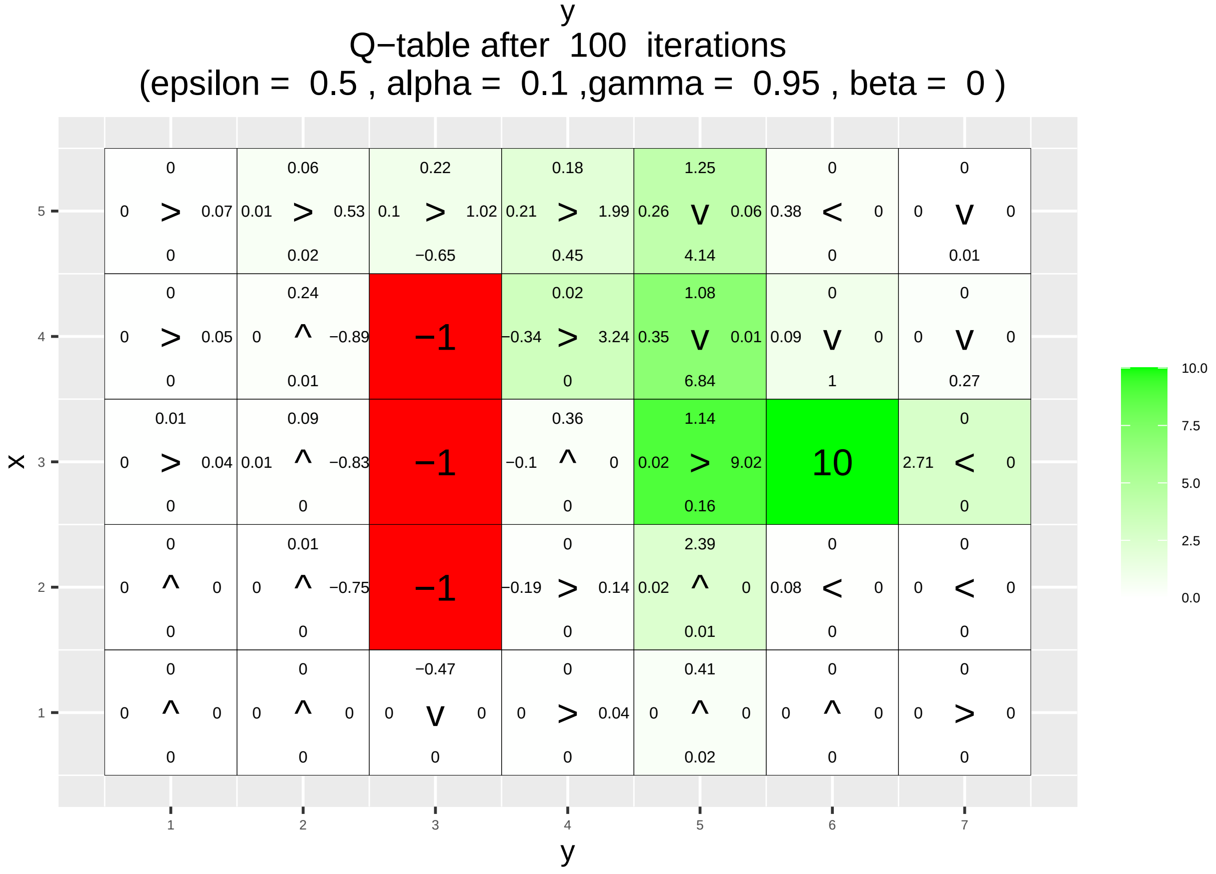
If we want to make the agent find multiple paths, we should set epsilon to a big value, so the agent will have more random actions to explore the new paths. One thing to mention is if we increase the epsilon, the agent may need more iterations to find the optimal paths. Meanwhile, we can change the learning rate to a relatively bigger number so the agent will explore more paths to the target.

Init stage

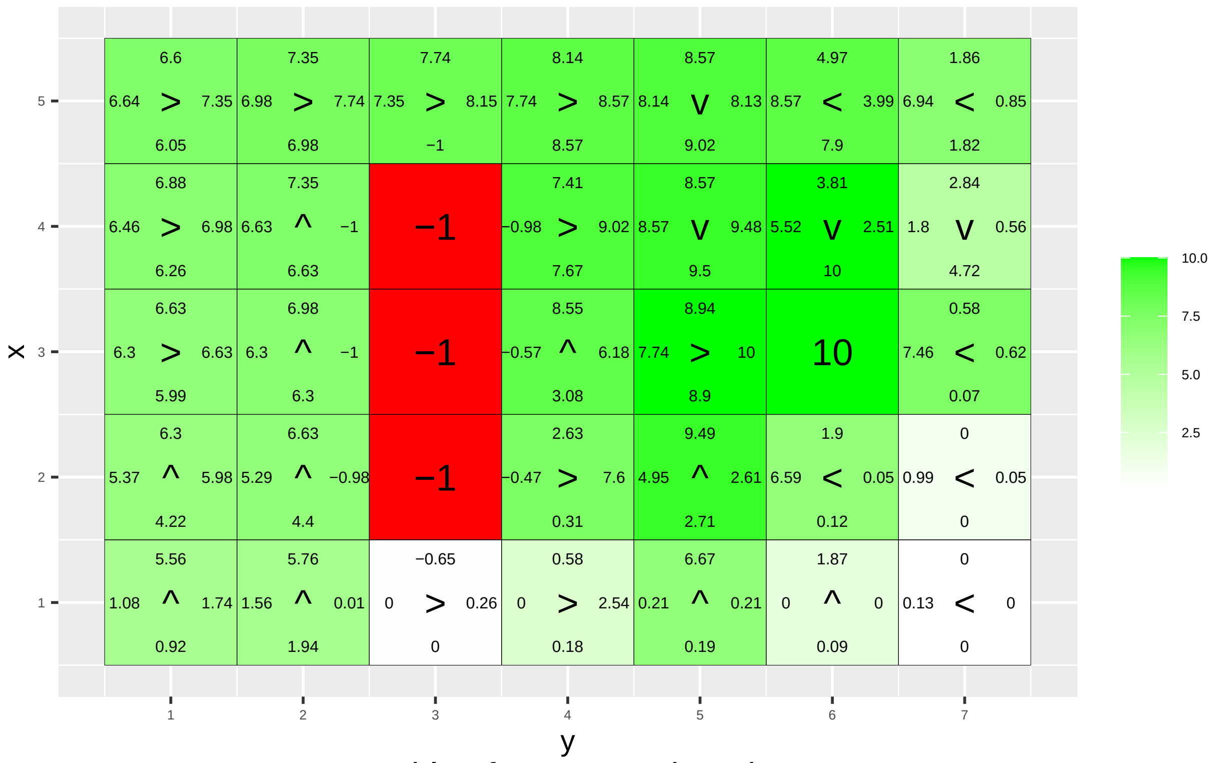


Different stages

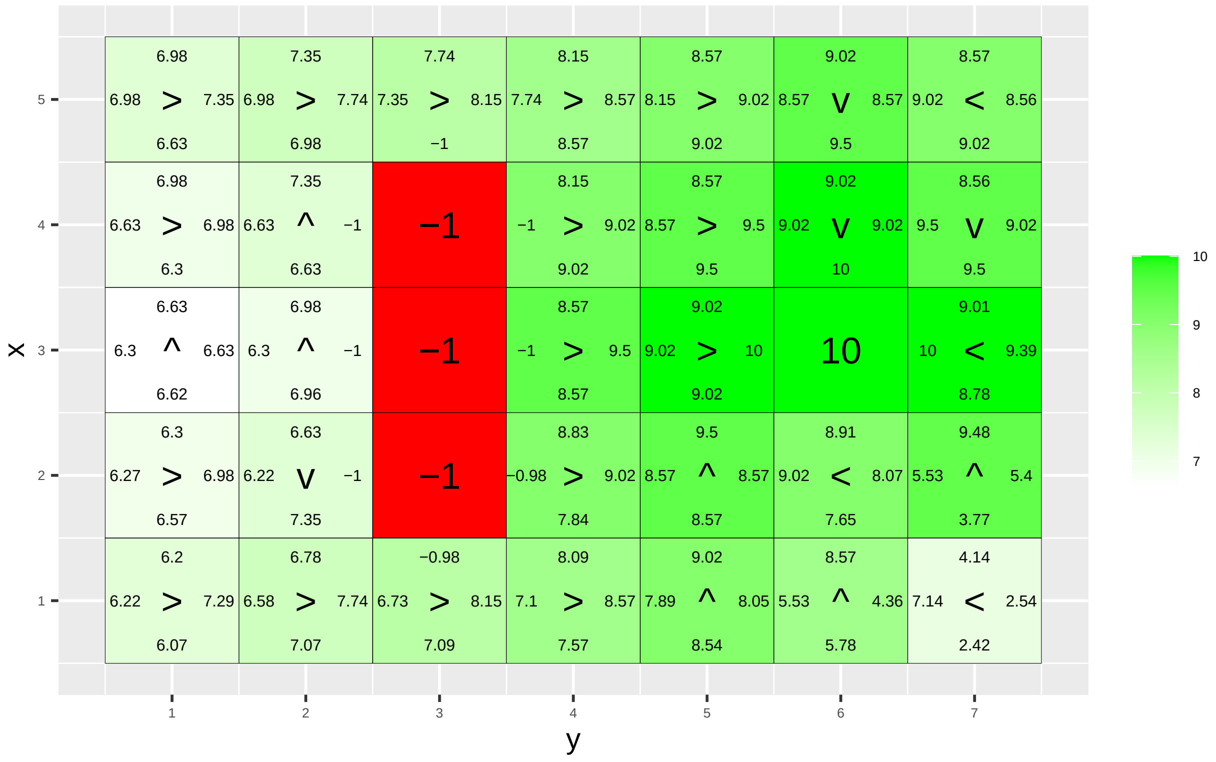

Q−table after 10 iterations
(epsilon = 0.5 , alpha = 0.1 ,gamma = 0.95 , beta = 0)
0 0 0 0 0 0 0
0 < 0 | 0 V 0 | 0 ^ 0 | 0 V 0 | 0 < 0 | 0 ^ 0 | 0 V 0
0 0 −0.1 0 0 0 0
0 0 0 0 0 0
0 ^ 0 | 0 ^ −0.1 | −1 | −0.1 > 0 | 0 < 0 | 0 ^ 0 | 0 V 0
0 0 0 0 0 0
0 0 0 0 0
0 ^ 0 | 0 V −0.1 | −1 | −0.1 ^ 0 | 0 ^ 0 | 10 | 1 < 0
0 0 0 0 0
0 0 0 0 0 0
0 < 0 | 0 V −0.1 | −1 | −0.1 V 0 | 0 ^ 0 | 0 ^ 0 | 0 V 0
0 0 0 0 0 0
0 0 −0.19 0 0 0 0
0 > 0 | 0 ^ 0 | 0 V 0 | 0 < 0 | 0 V 0 | 0 ^ 0 | 0 V 0
0 0 0 0 0 0 0
x
5
4
3
2
1
1 2 3 4 5 6 7
y
10.0
7.5
5.0
2.5
0.0


Q−table after 100 iterations
(epsilon = 0.5 , alpha = 0.1 ,gamma = 0.95 , beta = 0)
0 0.06 0.22 0.18 1.25 0 0
0 > 0.07 | 0.01 > 0.53 | 0.1 > 1.02 | 0.21 > 1.99 | 0.26 V 0.06 | 0.38 < 0 | 0 V 0
0 0.02 −0.65 0.45 4.14 0 0.01
0 0.24 0.02 1.08 0 0
0 > 0.05 | 0 ^ −0.89 | −1 | −0.34 > 3.24 | 0.35 V 0.01 | 0.09 V 0 | 0 V 0
0 0.01 0 6.84 1 0.27
0.01 0.09 0.36 1.14 0
0 > 0.04 | 0.01 ^ −0.83 | −1 | −0.1 ^ 0 | 0.02 > 9.02 | 10 | 2.71 < 0
0 0 0 0.16 0
0 0.01 0 2.39 0 0
0 ^ 0 | 0 ^ −0.75 | −1 | −0.19 > 0.14 | 0.02 ^ 0 | 0.08 < 0 | 0 < 0
0 0 0 0.01 0 0
0 0 −0.47 0 0.41 0 0
0 ^ 0 | 0 ^ 0 | 0 V 0 | 0 > 0.04 | 0 ^ 0 | 0 ^ 0 | 0 > 0
0 0 0 0 0.02 0 0
x
5
4
3
2
1
1 2 3 4 5 6 7
y
10.0
7.5
5.0
2.5
0.0

Q−table after 1000 iterations
(epsilon = 0.5 , alpha = 0.1 ,gamma = 0.95 , beta = 0)



Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 ,gamma = 0.95 , beta = 0)



3. Environment B (epsilon and gamma)

Question

This is a 7×8 environment where the top and bottom rows have negative rewards. In this environment, the agent starts each episode in the state (4, 1). There are two positive rewards, of 5 and 10. The reward of 5 is easily reachable, but the agent has to navigate around the first reward in order to find the reward worth 10.

Investigate how the ϵ and γ parameters affect the learned policy by running 30000 episodes of Q-learning with $\epsilon = 0.1, 0.5$, $\gamma = 0.5, 0.75, 0.95$, $\beta = 0$ and $\alpha = 0.1$ and explain observations.

Answer

In environment B, the start state is (x=4,y=1), and parameters are set as epsilon = 0.1, 0.5, alpha = 0.1, gamma = 0.5, 0.75, 0.95, beta = 0.

Gamma and epsilon have a significant impact on the learned policy.

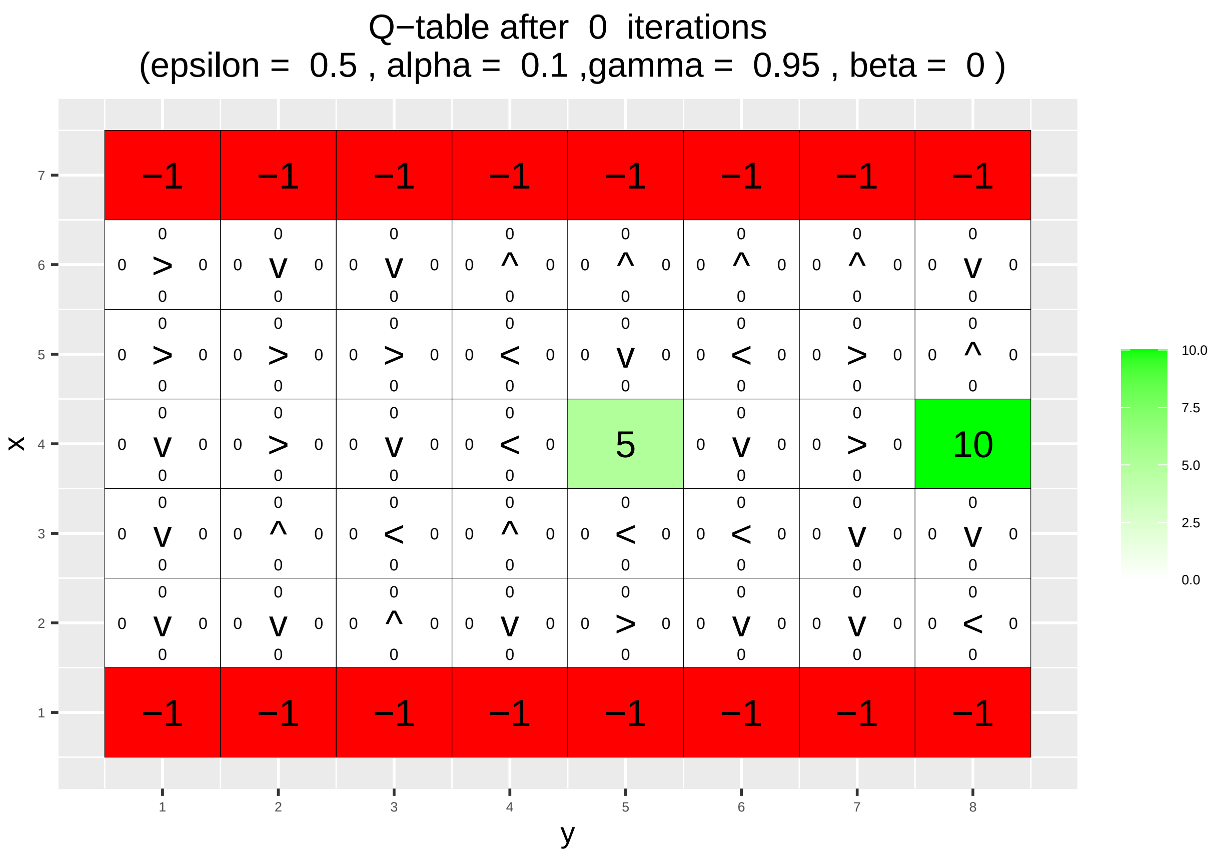
Comparing the different gamma values, and all other parameters fixed, we can find that when gamma is very high(0.95), reward 5 will be ignored, and the agent prefers reward 10.

When gamma is small(0.5), the agent will have some chances to choose reward 5 and then stop, but it has more chance to choose reward 10. The reason is that when gamma gets very high, the agent will place significant value on future rewards when making decisions, the future reward in this case is reward 10 instead of reward 5. If gamma is small, the agent will focus more on short-term gains, so it will choose reward 5 first.

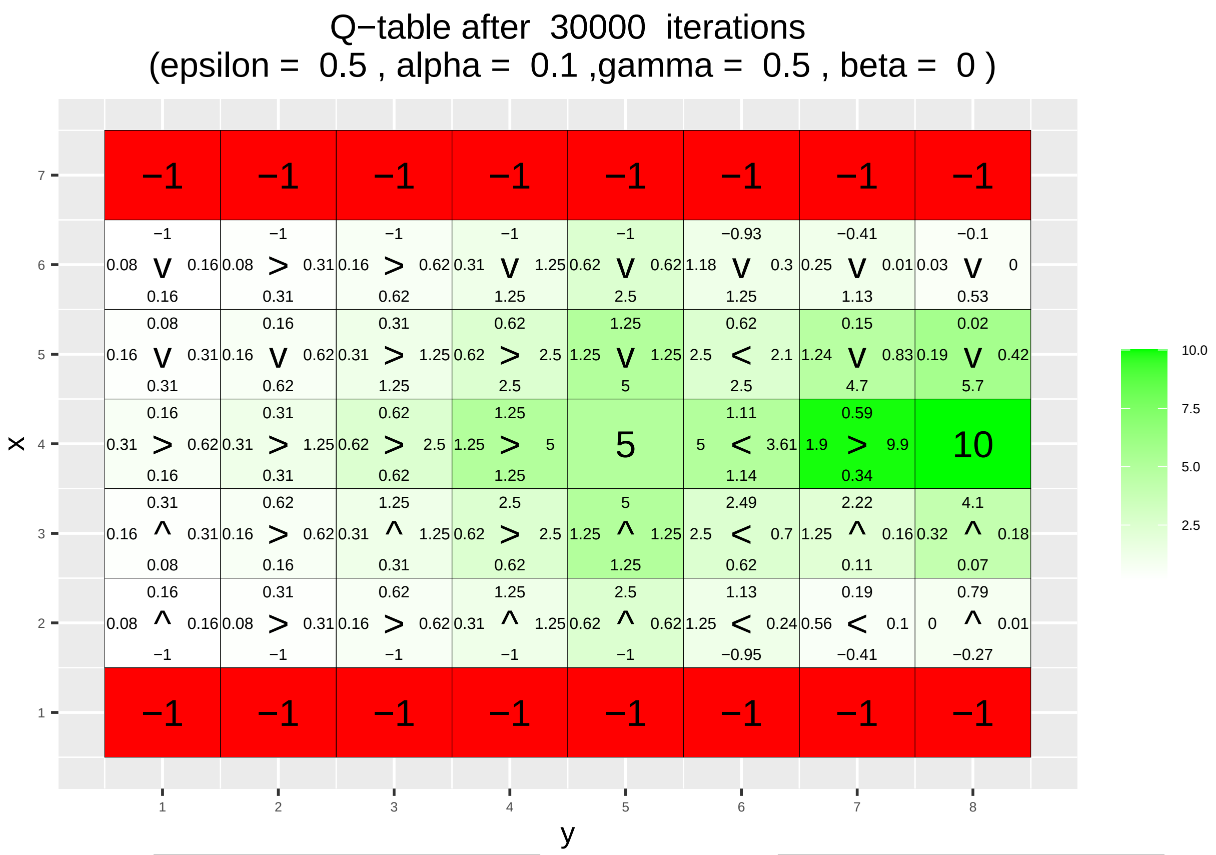
This can be seen from the Moving Average plot(gamma = 0.95 and epsilon = 0.5) vs (gamma = 0.5 and epsilon = 0.5). When gamma is 0.5, the average reward keep at a similar level, but when gamma is 0.95 when the episode increases to 25000, the agent finds a new high reward path, and then the average reward afterwards increases to a new higher level, and the correction plot also reflect this situation, which means the q-table change a lot in this time.

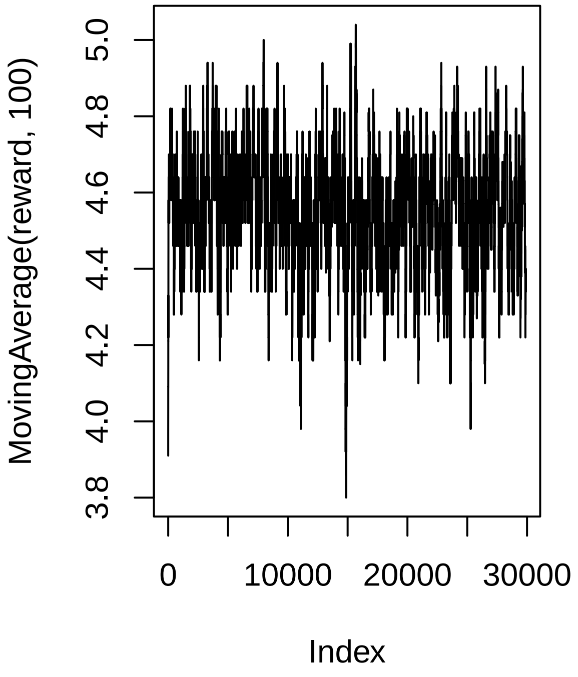
Comparing the different epsilon values(0.5 and 0.1)'s q-table, the high epsilon q-table has been highly explored, while the low-value epsilon q-table has been less explored. The reason behind this is that epsilon is the probability of acting randomly, when epsilon is high, the agent will have more chance to choose random actions, which means it has more chance to explore new paths. When the epsilon is small, the agent will have more chance to choose greedy policy rather than random actions, so it will prefer the existing optimized path.

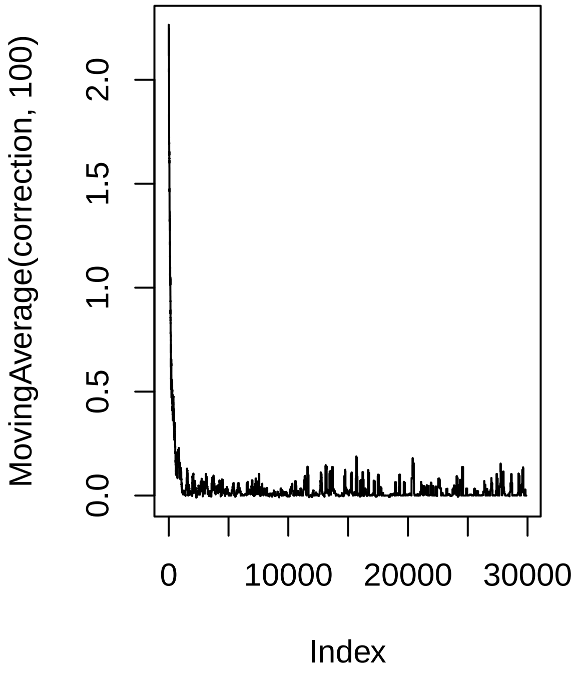
Init stage

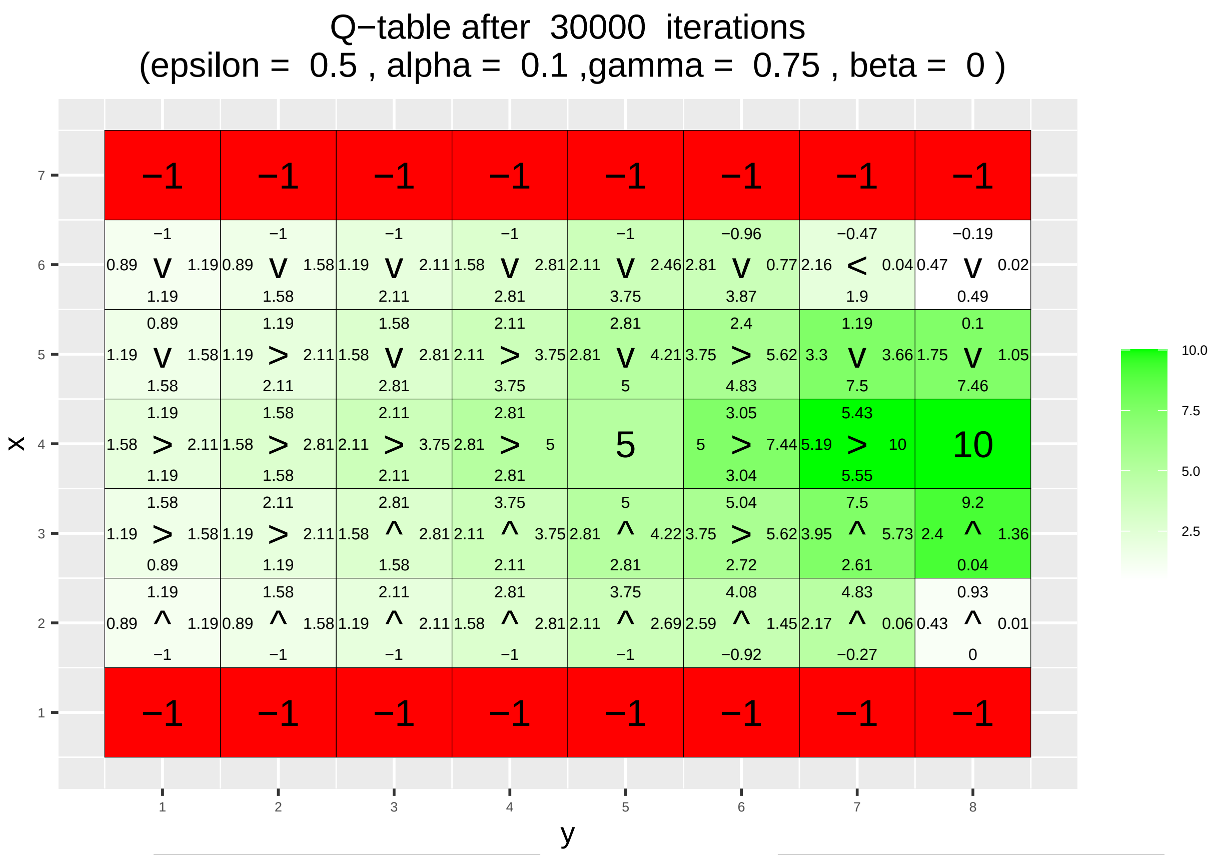


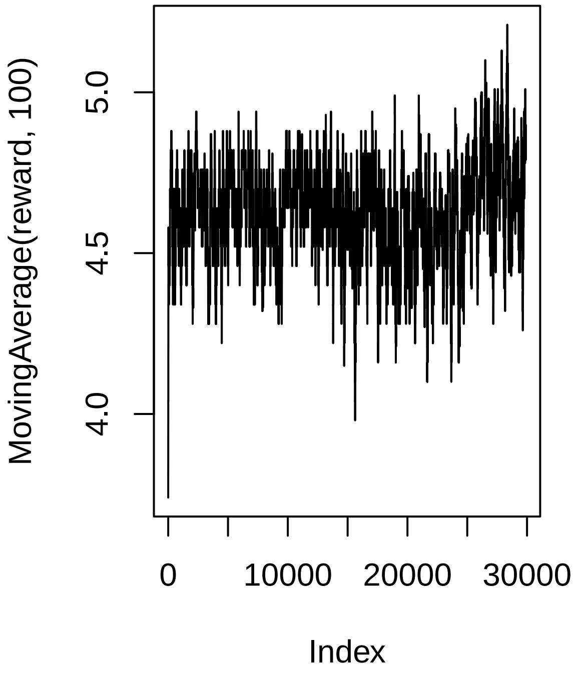
Different gamma for epsilon = 0.5

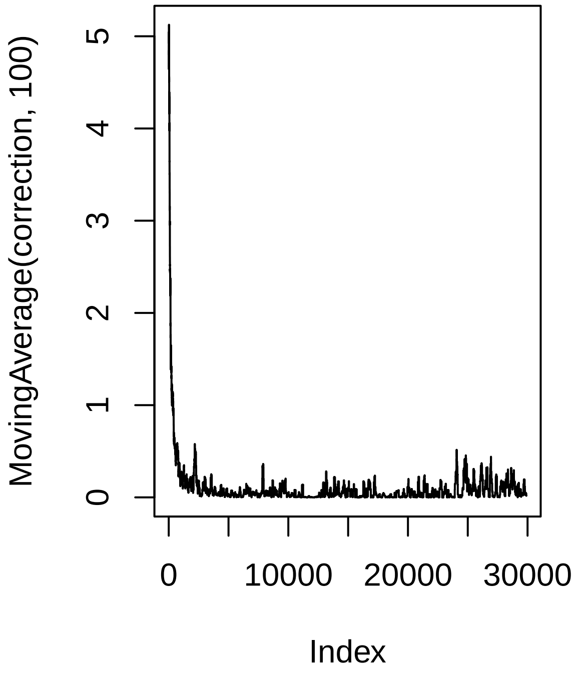

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 ,gamma = 0.5 , beta = 0)
x
y
7: −1 −1 −1 −1 −1 −1 −1 −1
6: −1 / 0.08 V 0.16 / 0.16 | −1 / 0.08 > 0.31 / 0.31 | −1 / 0.16 > 0.62 / 0.62 | −1 / 0.31 V 1.25 / 1.25 | −1 / 0.62 V 0.62 / 2.5 | −0.93 / 1.18 V 0.3 / 1.25 | −0.41 / 0.25 V 0.01 / 1.13 | −0.1 / 0.03 V 0 / 0.53
5: 0.08 / 0.16 V 0.31 / 0.31 | 0.16 / 0.16 V 0.62 / 0.62 | 0.31 / 0.31 > 1.25 / 1.25 | 0.62 / 0.62 > 2.5 / 2.5 | 1.25 / 1.25 V 1.25 / 5 | 0.62 / 2.5 < 2.1 / 2.5 | 0.15 / 1.24 V 0.83 / 4.7 | 0.02 / 0.19 V 0.42 / 5.7
4: 0.16 / 0.31 > 0.62 / 0.16 | 0.31 / 0.31 > 1.25 / 0.31 | 0.62 / 0.62 > 2.5 / 0.62 | 1.25 / 1.25 > 5 / 1.25 | 5 | 1.11 / 5 < 3.61 / 1.14 | 0.59 / 1.9 > 9.9 / 0.34 | 10
3: 0.31 / 0.16 ^ 0.31 / 0.08 | 0.62 / 0.16 > 0.62 / 0.16 | 1.25 / 0.31 ^ 1.25 / 0.31 | 2.5 / 0.62 > 2.5 / 0.62 | 5 / 1.25 ^ 1.25 / 1.25 | 2.49 / 2.5 < 0.7 / 0.62 | 2.22 / 1.25 ^ 0.16 / 0.11 | 4.1 / 0.32 ^ 0.18 / 0.07
2: 0.16 / 0.08 ^ 0.16 / −1 | 0.31 / 0.08 > 0.31 / −1 | 0.62 / 0.16 > 0.62 / −1 | 1.25 / 0.31 ^ 1.25 / −1 | 2.5 / 0.62 ^ 0.62 / −1 | 1.13 / 1.25 < 0.24 / −0.95 | 0.19 / 0.56 < 0.1 / −0.41 | 0.79 / 0 ^ 0.01 / −0.27
1: −1 −1 −1 −1 −1 −1 −1 −1
1 2 3 4 5 6 7 8
10.0
7.5
5.0
2.5

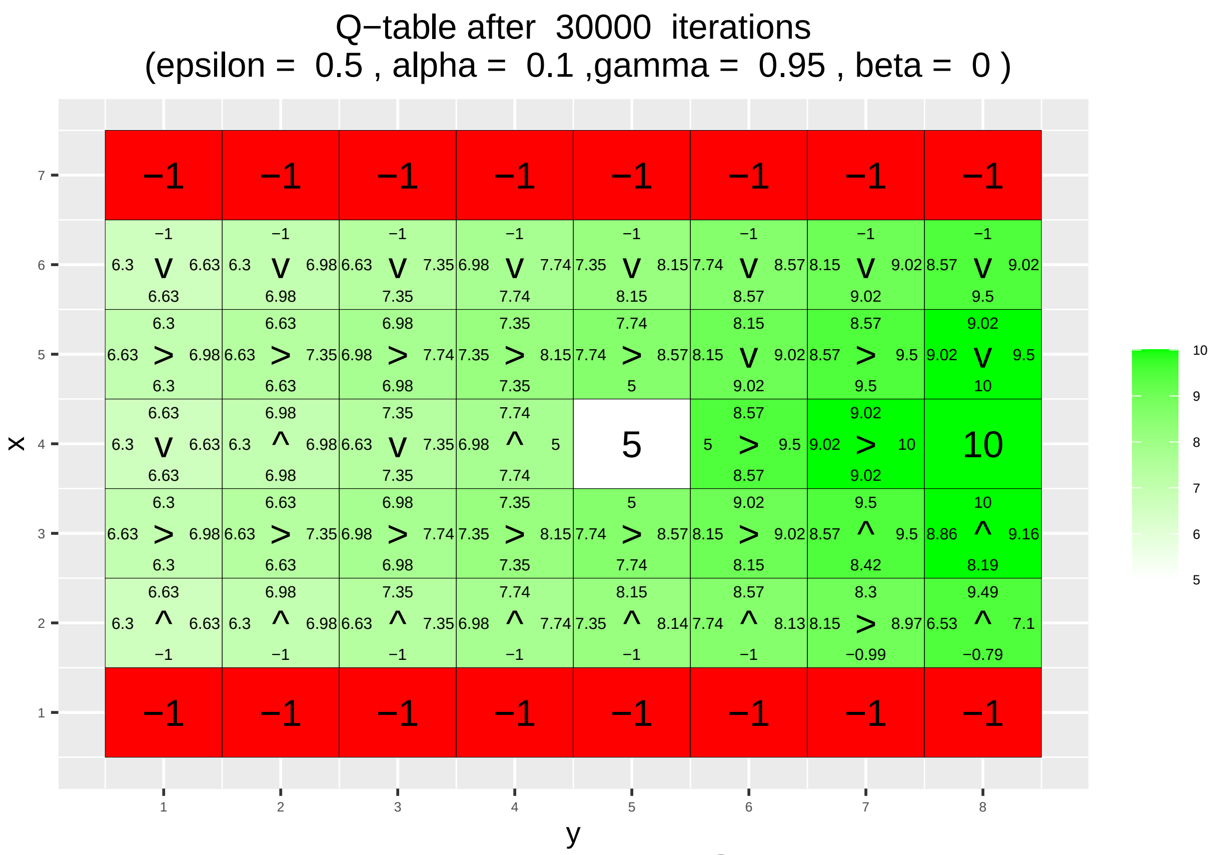

MovingAverage(reward, 100)
3.8
4.0
4.2
4.4
4.6
4.8
5.0
0
10000
20000
30000
Index

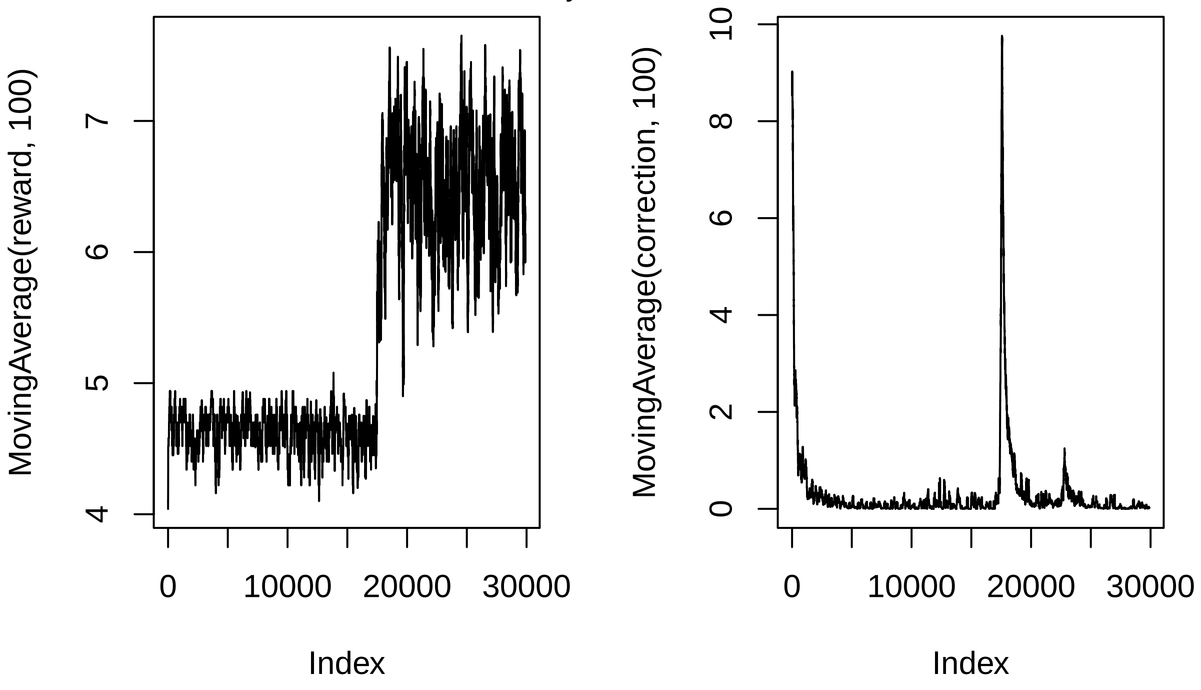

MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
2.0
0
10000
20000
30000
Index


Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 ,gamma = 0.75 , beta = 0)
x
y
1 2 3 4 5 6 7 8
7 6 5 4 3 2 1
10.0
7.5
5.0
2.5
−1 −1 −1 −1 −1 −1 −1 −1
−1 0.89 V 1.19 1.19 | −1 0.89 V 1.58 1.58 | −1 1.19 V 2.11 2.11 | −1 1.58 V 2.81 2.81 | −1 2.11 V 2.46 3.75 | −0.96 2.81 V 0.77 3.87 | −0.47 2.16 < 0.04 1.9 | −0.19 0.47 V 0.02 0.49
0.89 1.19 V 1.58 1.58 | 1.19 1.19 > 2.11 2.11 | 1.58 1.58 V 2.81 2.81 | 2.11 2.11 > 3.75 3.75 | 2.81 2.81 V 4.21 5 | 2.4 3.75 > 5.62 4.83 | 1.19 3.3 V 3.66 7.5 | 0.1 1.75 V 1.05 7.46
1.19 1.58 > 2.11 1.19 | 1.58 1.58 > 2.81 1.58 | 2.11 2.11 > 3.75 2.11 | 2.81 2.81 > 5 2.81 | 5 | 3.05 5 > 7.44 3.04 | 5.43 5.19 > 10 5.55 | 10
1.58 1.19 > 1.58 0.89 | 2.11 1.19 > 2.11 1.19 | 2.81 1.58 ^ 2.81 1.58 | 3.75 2.11 ^ 3.75 2.11 | 5 2.81 ^ 4.22 2.81 | 5.04 3.75 > 5.62 2.72 | 7.5 3.95 ^ 5.73 2.61 | 9.2 2.4 ^ 1.36 0.04
1.19 0.89 ^ 1.19 −1 | 1.58 0.89 ^ 1.58 −1 | 2.11 1.19 ^ 2.11 −1 | 2.81 1.58 ^ 2.81 −1 | 3.75 2.11 ^ 2.69 −1 | 4.08 2.59 ^ 1.45 −0.92 | 4.83 2.17 ^ 0.06 −0.27 | 0.93 0.43 ^ 0.01 0
−1 −1 −1 −1 −1 −1 −1 −1

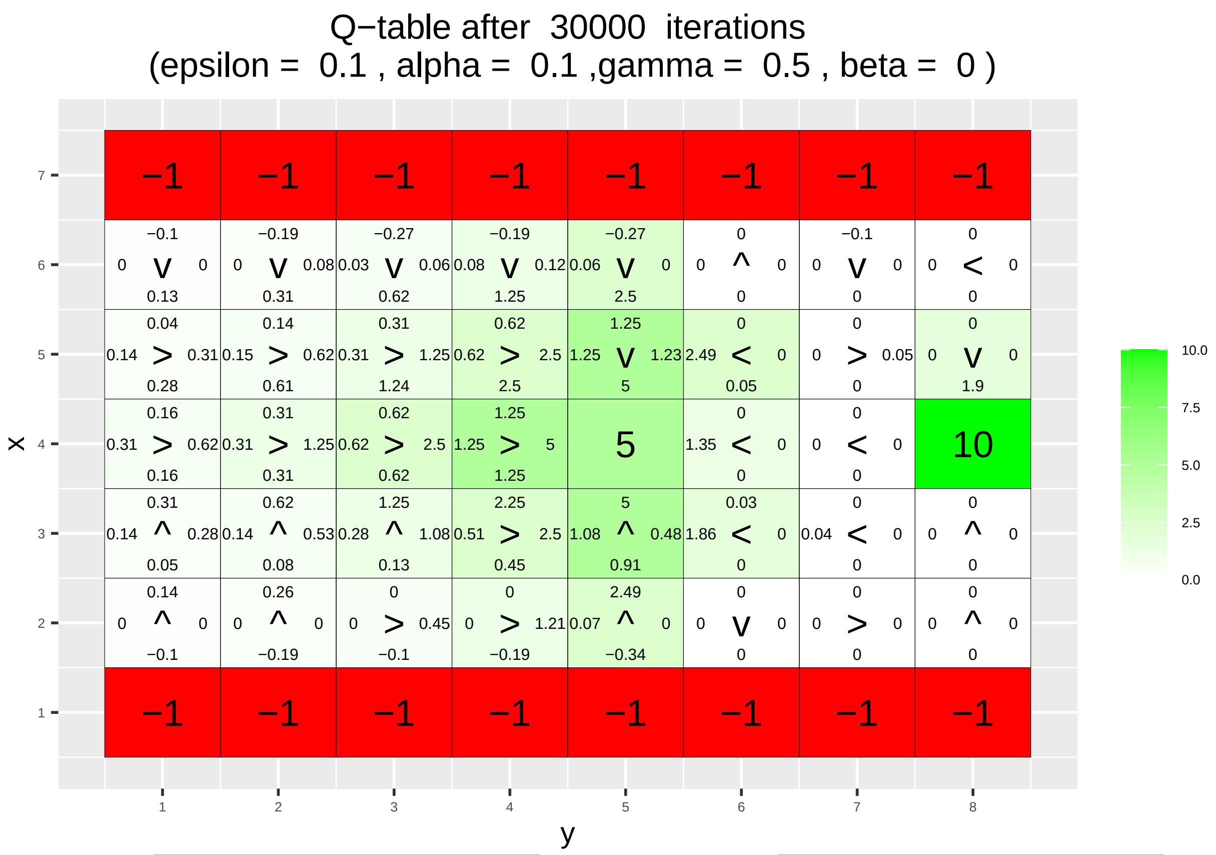

MovingAverage(reward, 100)
4.0
4.5
5.0
0 10000 20000 30000
Index

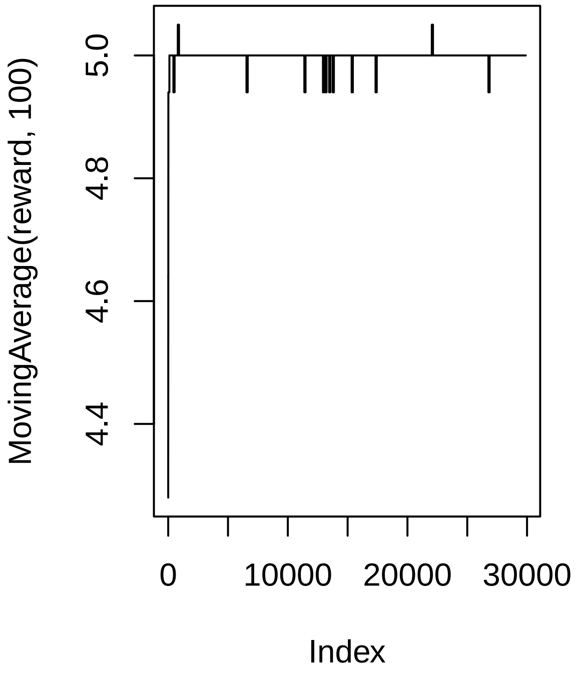

MovingAverage(correction, 100)
0 1 2 3 4 5
0 10000 20000 30000
Index

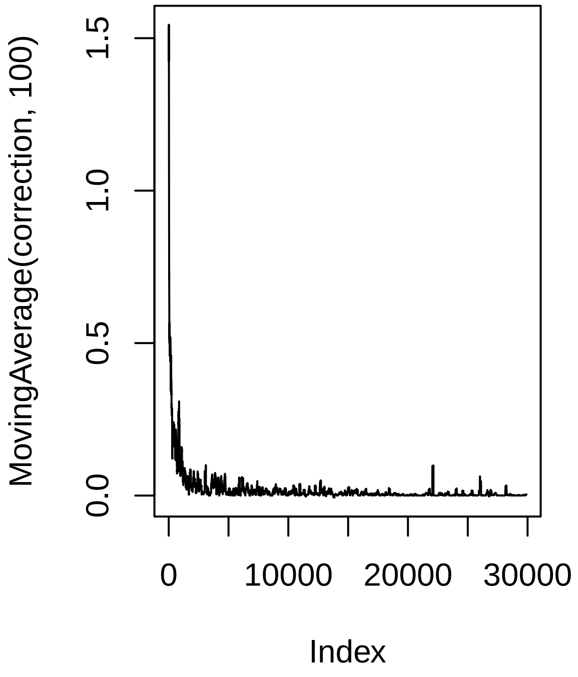


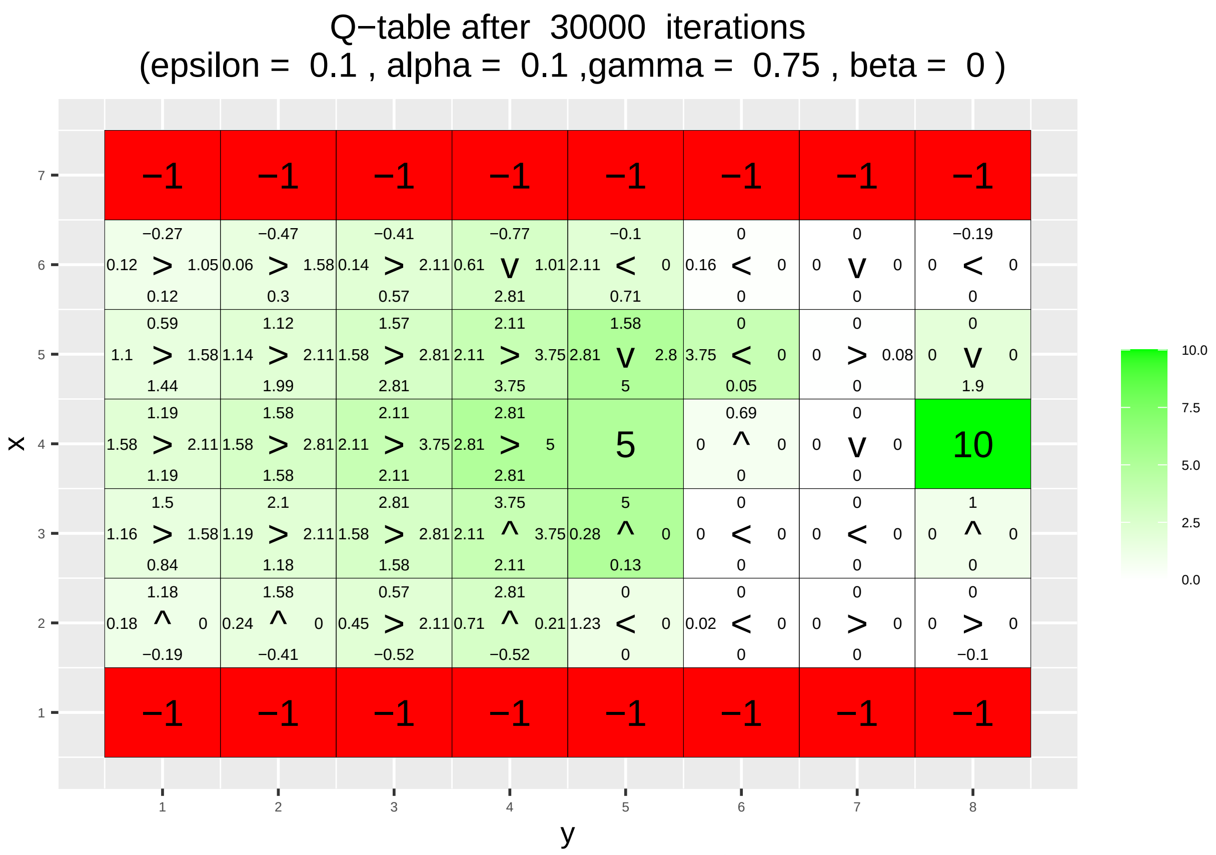


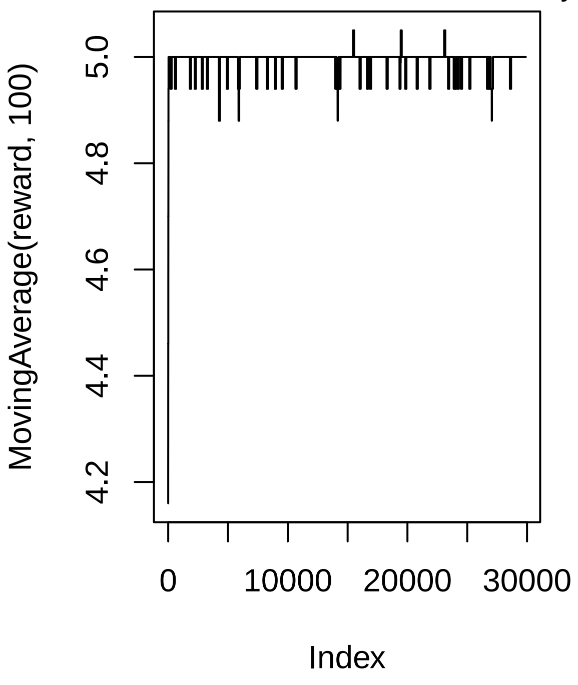
Different gamma for epsilon = 0.1

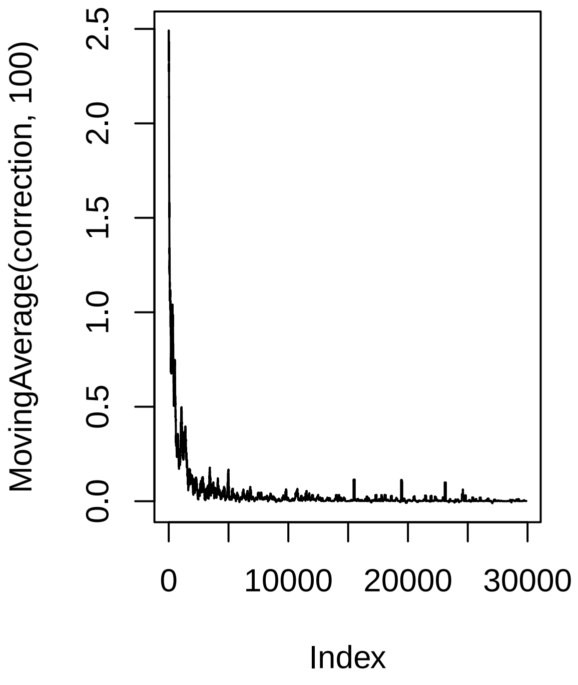

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 ,gamma = 0.5 , beta = 0)
x
y
7: −1 −1 −1 −1 −1 −1 −1 −1
6: −0.1 0 V 0 0.13 | −0.19 0 V 0.08 0.31 | −0.27 0.03 V 0.06 0.62 | −0.19 0.08 V 0.12 1.25 | −0.27 0.06 V 0 2.5 | 0 0 ^ 0 0 | −0.1 0 V 0 0 | 0 0 < 0 0
5: 0.04 0.14 > 0.31 0.28 | 0.14 0.15 > 0.62 0.61 | 0.31 0.31 > 1.25 1.24 | 0.62 0.62 > 2.5 2.5 | 1.25 1.25 V 1.23 5 | 0 2.49 < 0 0.05 | 0 0 > 0.05 0 | 0 0 V 0 1.9
4: 0.16 0.31 > 0.62 0.16 | 0.31 0.31 > 1.25 0.31 | 0.62 0.62 > 2.5 0.62 | 1.25 1.25 > 5 1.25 | 5 | 0 1.35 < 0 0 | 0 0 < 0 0 | 10
3: 0.31 0.14 ^ 0.28 0.05 | 0.62 0.14 ^ 0.53 0.08 | 1.25 0.28 ^ 1.08 0.13 | 2.25 0.51 > 2.5 0.45 | 5 1.08 ^ 0.48 0.91 | 0.03 1.86 < 0 0 | 0 0.04 < 0 0 | 0 0 ^ 0 0
2: 0.14 0 ^ 0 −0.1 | 0.26 0 ^ 0 −0.19 | 0 0 > 0.45 −0.1 | 0 0 > 1.21 −0.19 | 2.49 0.07 ^ 0 −0.34 | 0 0 V 0 0 | 0 0 > 0 0 | 0 0 ^ 0 0
1: −1 −1 −1 −1 −1 −1 −1 −1
1 2 3 4 5 6 7 8
10.0
7.5
5.0
2.5
0.0

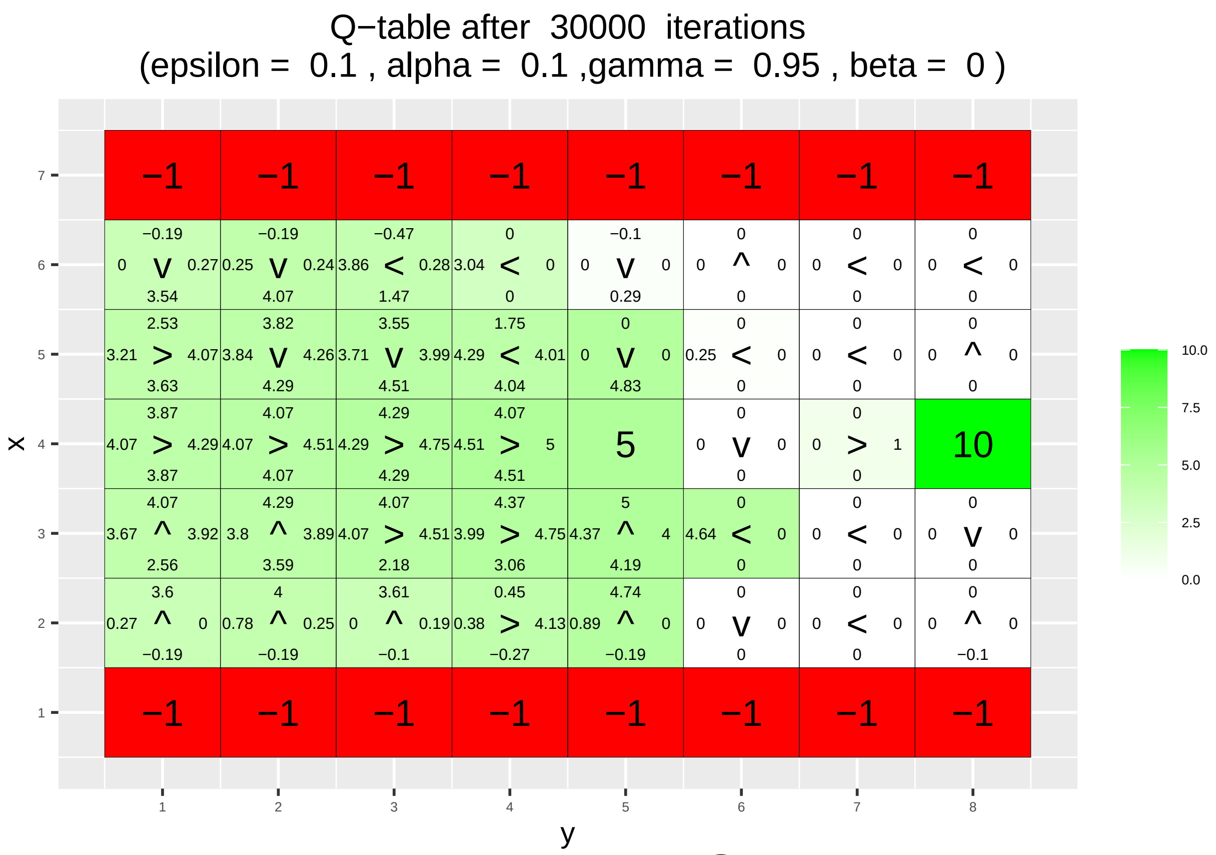

MovingAverage(reward, 100)
4.4
4.6
4.8
5.0
0
10000
20000
30000
Index

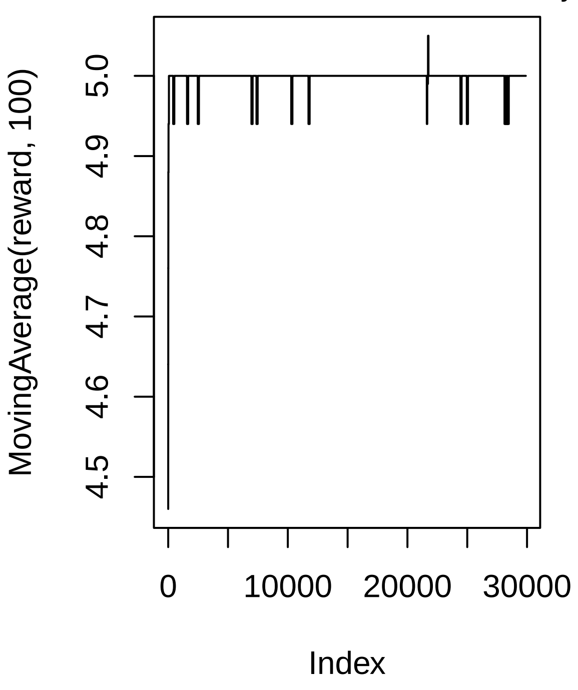

MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
0
10000
20000
30000
Index

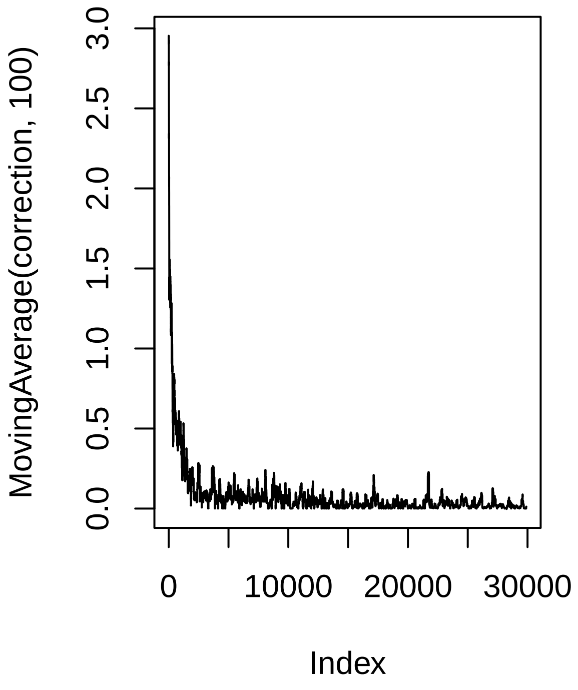

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 ,gamma = 0.75 , beta = 0)
x
y
7: −1 −1 −1 −1 −1 −1 −1 −1
6: −0.27 0.12 > 1.05 0.12 | −0.47 0.06 > 1.58 0.3 | −0.41 0.14 > 2.11 0.57 | −0.77 0.61 V 1.01 2.81 | −0.1 2.11 < 0 0.71 | 0 0.16 < 0 0 | 0 0 V 0 0 | −0.19 0 < 0 0
5: 0.59 1.1 > 1.58 1.44 | 1.12 1.14 > 2.11 1.99 | 1.57 1.58 > 2.81 2.81 | 2.11 2.11 > 3.75 3.75 | 1.58 2.81 V 2.8 5 | 0 3.75 < 0 0.05 | 0 0 > 0.08 0 | 0 0 V 0 1.9
4: 1.19 1.58 > 2.11 1.19 | 1.58 1.58 > 2.81 1.58 | 2.11 2.11 > 3.75 2.11 | 2.81 2.81 > 5 2.81 | 5 | 0.69 0 ^ 0 0 | 0 0 V 0 0 | 10
3: 1.5 1.16 > 1.58 0.84 | 2.1 1.19 > 2.11 1.18 | 2.81 1.58 > 2.81 1.58 | 3.75 2.11 ^ 3.75 2.11 | 5 0.28 ^ 0 0.13 | 0 0 < 0 0 | 0 0 < 0 0 | 1 0 ^ 0 0
2: 1.18 0.18 ^ 0 −0.19 | 1.58 0.24 ^ 0 −0.41 | 0.57 0.45 > 2.11 −0.52 | 2.81 0.71 ^ 0.21 −0.52 | 0 1.23 < 0 0 | 0 0.02 < 0 0 | 0 0 > 0 0 | 0 0 > 0 −0.1
1: −1 −1 −1 −1 −1 −1 −1 −1
1 2 3 4 5 6 7 8
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
4.2
4.4
4.6
4.8
5.0
0
10000
20000
30000
Index


MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
2.0
2.5
0
10000
20000
30000
Index







4. Environment C (beta)

Question

This is a smaller 3 x 6 environment. Here the agent starts each episode in the state (1,1). Your task is to investigate how the β parameter affects the learned policy by running 10000 episodes of Q-learning with

$\beta = 0, 0.2, 0.4, 0.66$, $\epsilon = 0.5$, $\gamma = 0.6$ and $\alpha = 0.1$.

In environment C, the start state is (x=1,y=1).

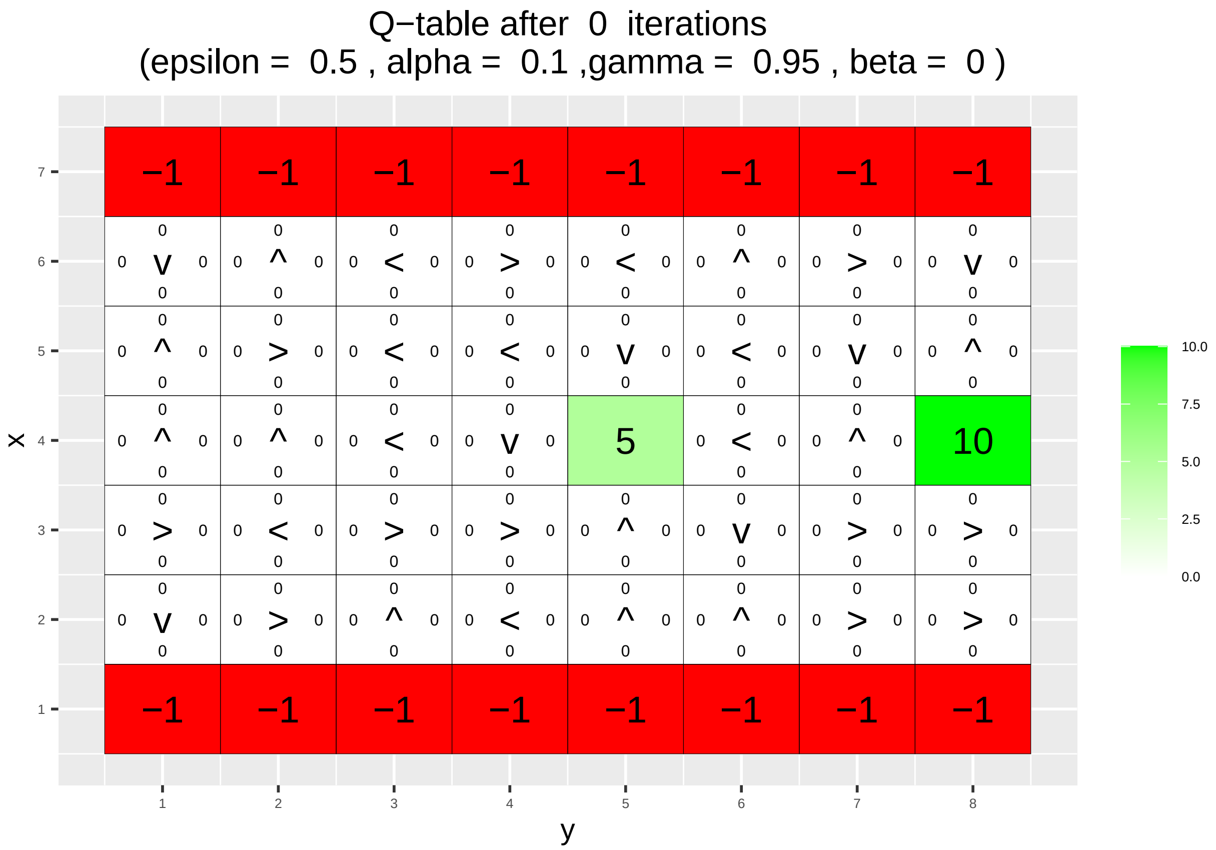
Answer

From the plots below, we can get the conclusion that beta (0, 0.2, 0.4, 0.66), did not show a significant impact on the learned policy.

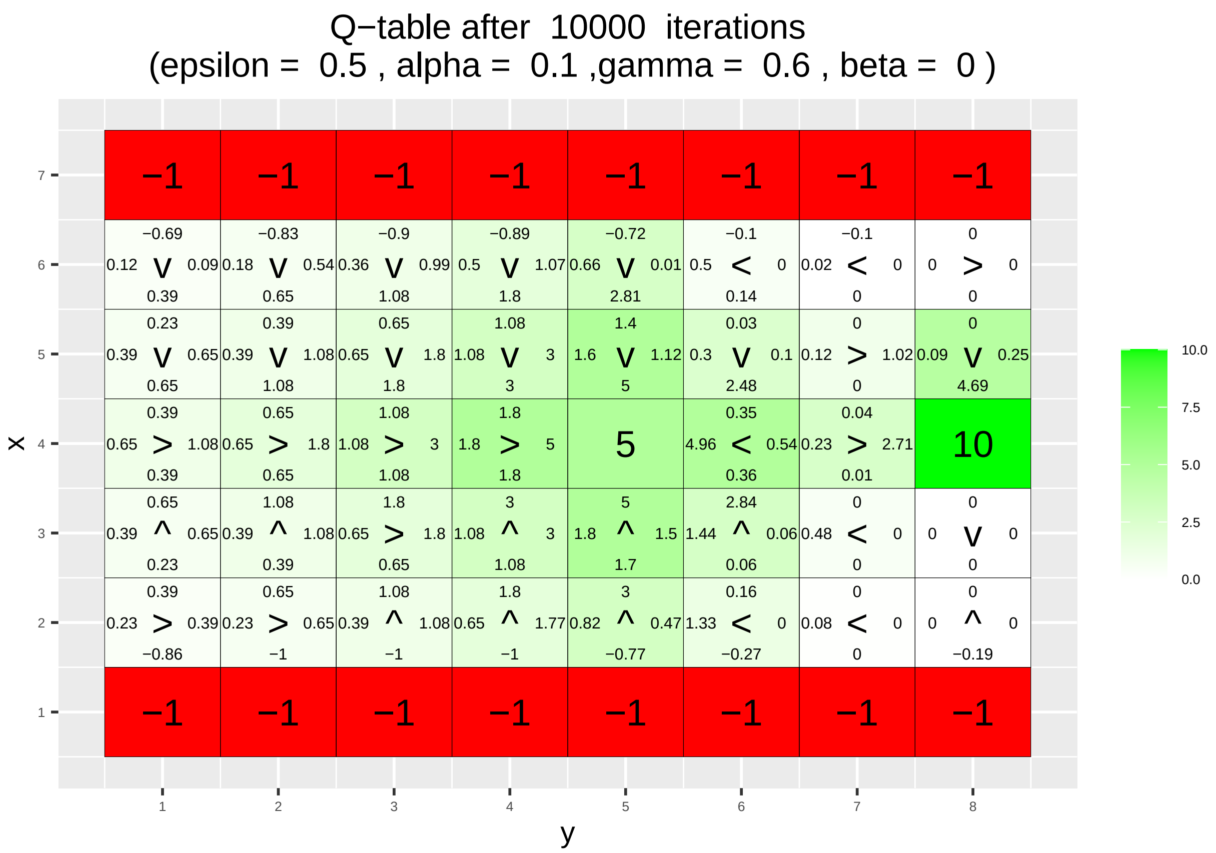
when beta increases, the agent has more chance to slip to the side, however, in our case, our world is small, and it is easy for the agent to find the optimal path to the target state, randomly turning left and turning right will help to find the optimal path, So the agent will not greatly be affected by the beta value.

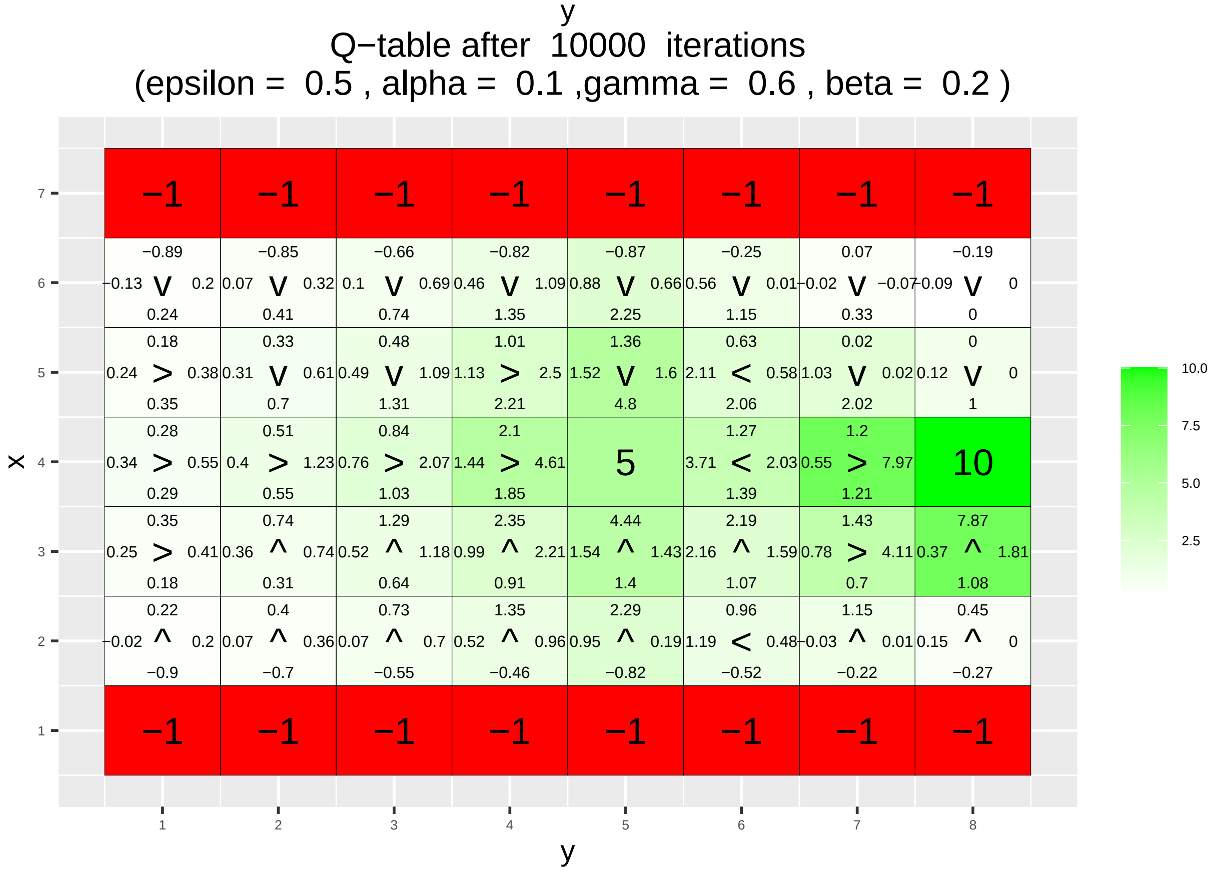
Meanwhile, gamma is 0.6, which is not very high, and the agent will not focus on the long-term reward, so the agent will not explore more paths to the target state which results in several not fully explored states in the q-table.

Init stage

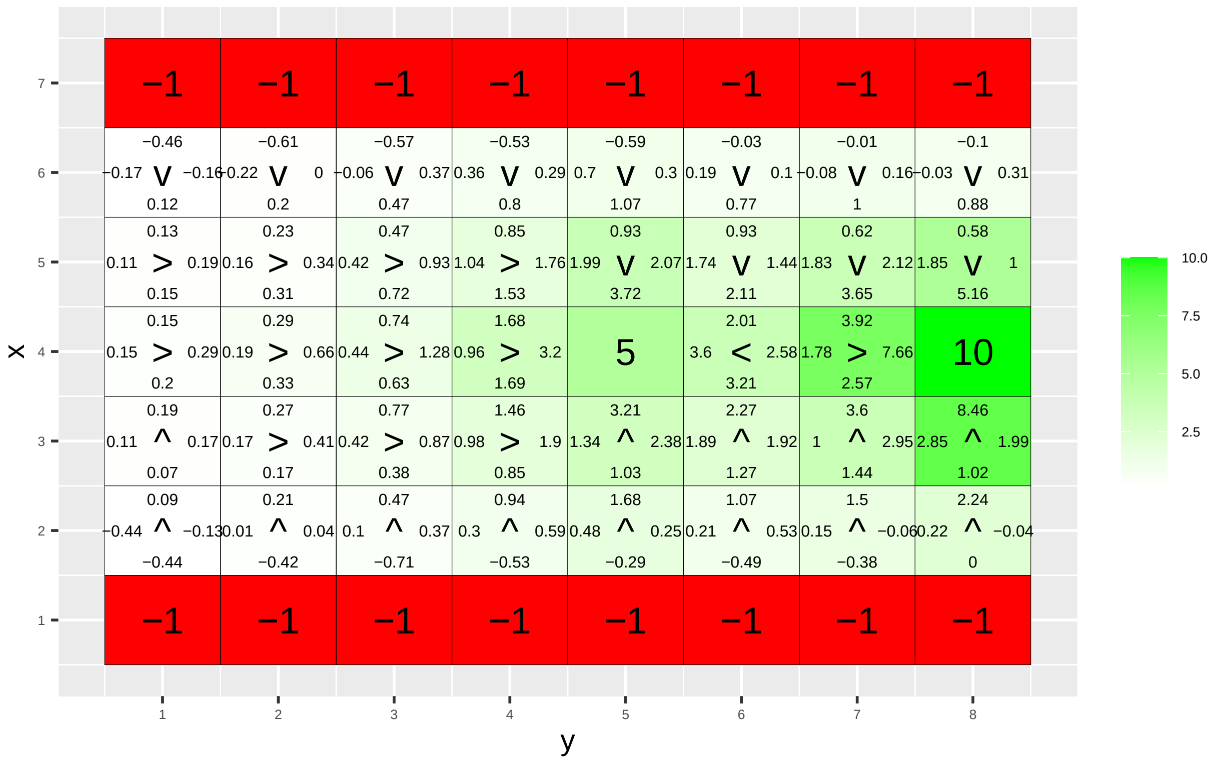


Different beta

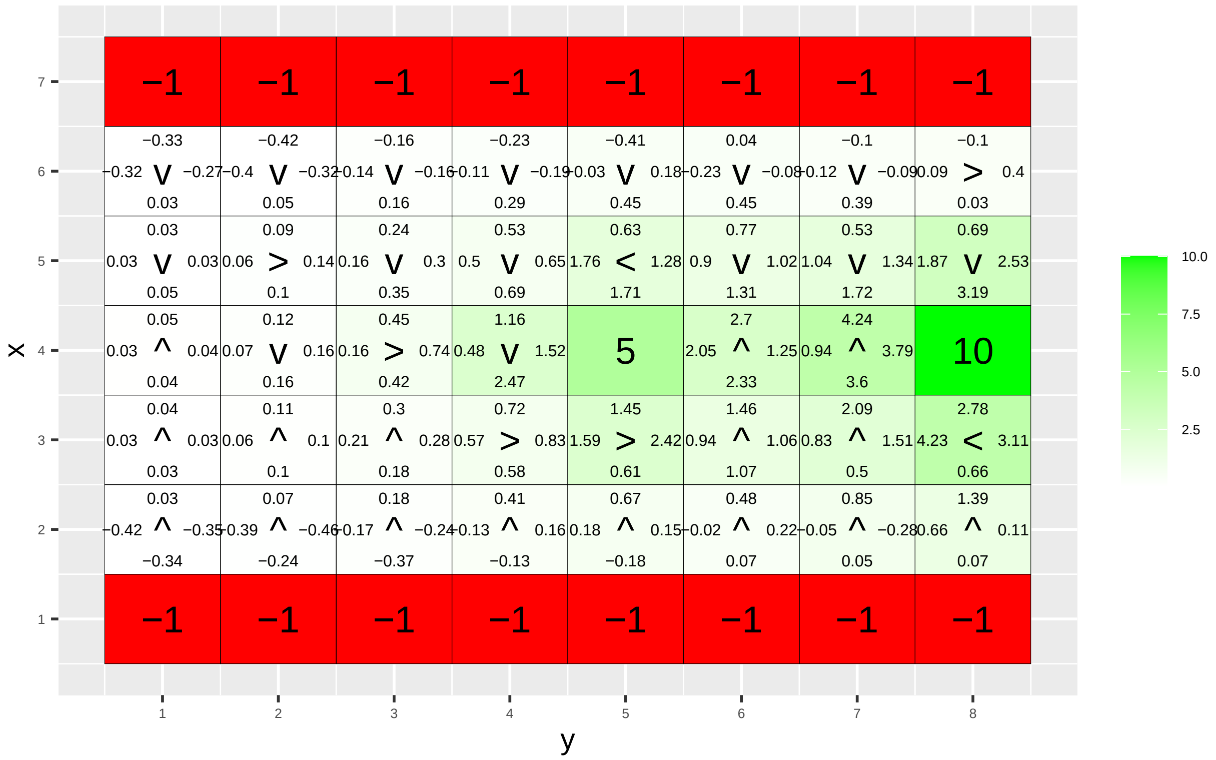

Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 ,gamma = 0.6 , beta = 0)
−1 −1 −1 −1 −1 −1 −1 −1
−0.69 −0.83 −0.9 −0.89 −0.72 −0.1 −0.1 0
0.12 V 0.09 0.18 V 0.54 0.36 V 0.99 0.5 V 1.07 0.66 V 0.01 0.5 < 0 0.02 < 0 0 > 0
0.39 0.65 1.08 1.8 2.81 0.14 0 0
0.23 0.39 0.65 1.08 1.4 0.03 0 0
0.39 V 0.65 0.39 V 1.08 0.65 V 1.8 1.08 V 3 1.6 V 1.12 0.3 V 0.1 0.12 > 1.02 0.09 V 0.25
0.65 1.08 1.8 3 5 2.48 0 4.69
0.39 0.65 1.08 1.8 0.35 0.04
0.65 > 1.08 0.65 > 1.8 1.08 > 3 1.8 > 5 5 4.96 < 0.54 0.23 > 2.71 10
0.39 0.65 1.08 1.8 0.36 0.01
0.65 1.08 1.8 3 5 2.84 0 0
0.39 ^ 0.65 0.39 ^ 1.08 0.65 > 1.8 1.08 ^ 3 1.8 ^ 1.5 1.44 ^ 0.06 0.48 < 0 0 V 0
0.23 0.39 0.65 1.08 1.7 0.06 0 0
0.39 0.65 1.08 1.8 3 0.16 0 0
0.23 > 0.39 0.23 > 0.65 0.39 ^ 1.08 0.65 ^ 1.77 0.82 ^ 0.47 1.33 < 0 0.08 < 0 0 ^ 0
−0.86 −1 −1 −1 −0.77 −0.27 0 −0.19
−1 −1 −1 −1 −1 −1 −1 −1
x
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8
y
10.0
7.5
5.0
2.5
0.0


Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 ,gamma = 0.6 , beta = 0.2)
−1 −1 −1 −1 −1 −1 −1 −1
−0.89 −0.85 −0.66 −0.82 −0.87 −0.25 0.07 −0.19
−0.13 V 0.2 0.07 V 0.32 0.1 V 0.69 0.46 V 1.09 0.88 V 0.66 0.56 V 0.01 −0.02 V −0.07 −0.09 V 0
0.24 0.41 0.74 1.35 2.25 1.15 0.33 0
0.18 0.33 0.48 1.01 1.36 0.63 0.02 0
0.24 > 0.38 0.31 V 0.61 0.49 V 1.09 1.13 > 2.5 1.52 V 1.6 2.11 < 0.58 1.03 V 0.02 0.12 V 0
0.35 0.7 1.31 2.21 4.8 2.06 2.02 1
0.28 0.51 0.84 2.1 1.27 1.2
0.34 > 0.55 0.4 > 1.23 0.76 > 2.07 1.44 > 4.61 5 3.71 < 2.03 0.55 > 7.97 10
0.29 0.55 1.03 1.85 1.39 1.21
0.35 0.74 1.29 2.35 4.44 2.19 1.43 7.87
0.25 > 0.41 0.36 ^ 0.74 0.52 ^ 1.18 0.99 ^ 2.21 1.54 ^ 1.43 2.16 ^ 1.59 0.78 > 4.11 0.37 ^ 1.81
0.18 0.31 0.64 0.91 1.4 1.07 0.7 1.08
0.22 0.4 0.73 1.35 2.29 0.96 1.15 0.45
−0.02 ^ 0.2 0.07 ^ 0.36 0.07 ^ 0.7 0.52 ^ 0.96 0.95 ^ 0.19 1.19 < 0.48 −0.03 ^ 0.01 0.15 ^ 0
−0.9 −0.7 −0.55 −0.46 −0.82 −0.52 −0.22 −0.27
−1 −1 −1 −1 −1 −1 −1 −1
x
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8
y
10.0
7.5
5.0
2.5

Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 ,gamma = 0.6 , beta = 0.4)



Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 ,gamma = 0.6 , beta = 0.66)



Reinforce

The file RL Lab2 Colab.ipynb in the course website contains an implementation of the REINFORCE algorithm. The file also contains the result of running the code,so that you do not have to run it. So, you do not need to run it if you do not want. Your task is to study the code and the results obtained, and answer some questions.

We will work with a 4×4 grid. We want the agent to learn to navigate to a random goalposition in the grid. The agent will start in a random position and it will be told the goal position.

The agent receives a reward of 5 when it reaches the goal. Since the goal position can be any position, we need a way to tell the agent where the goal is. Since our agent does not have any memory mechanism, we provide the goal coordinates as part of the state at every time step, i.e. a state consists now of four coordinates: Two for the position of the agent, and two for the goal position. The actions of the agent can however only impact its own position, i.e. the actions do not modify the goal position. Note that the agent initially does not know that the last two coordinates of a state indicate the position with maximal reward, i.e. the goal position. It has to learn it. It also has to learn a policy to reach the goal position from the initial position. Moreover, the policy has to depend on the goal position, because it is chosen at random in each episode. Since we only have a single non-zero reward, we do not specify a reward map. Instead, the goal coordinates are passed to the functions that need to access the reward function.

5. Environment D

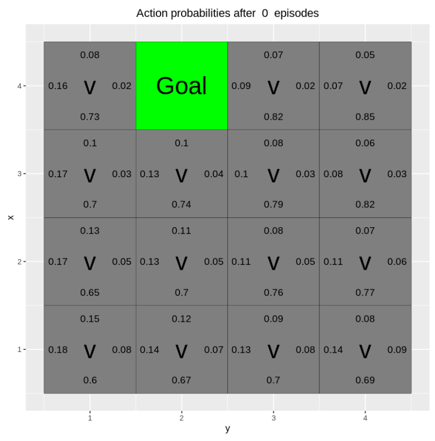
Question

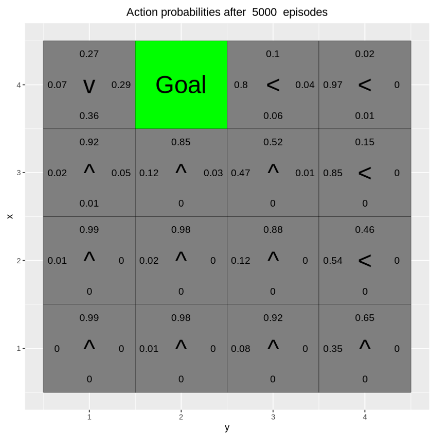
In this task, we will use eight goal positions for training and, then, validate the learned policy on the remaining eight possible goal positions. The training and validation goal positions are stored in the lists train goals and val goals in the code in the file RL Lab2 Colab.ipynb. The results provided in the file correspond to running the REINFORCE algorithm for 5000 episodes with $\beta = 0$ and $\gamma = 0.95$. Each training episode uses a random goal position from train goals. The initial position for the episode is also chosen at random. When training is completed, the code validates the learned policy for the goal positions in val goals. This is done by with the help of the function vis prob, which shows the grid, goal position and learned policy. Note that each non-terminal tile has four values. These represent the action probabilities associated to the tile (state). Note also that each non-terminal tile has an arrow. This indicates the action with the largest probability for the tile (ties are broken at random). Finally, answer the following questions:

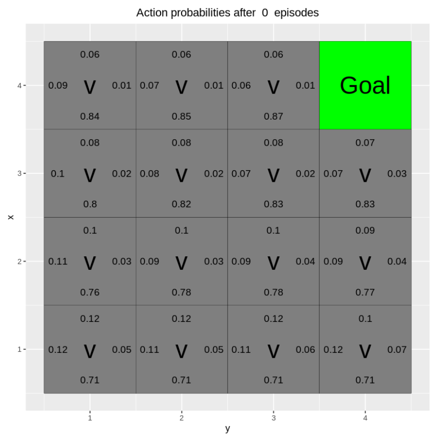
- 1. Has the agent learned a good policy? Why / Why not ?
- 2. Could you have used the Q-learning algorithm to solve this task ?

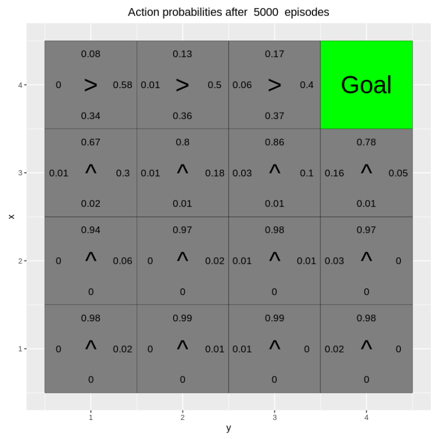
Answer

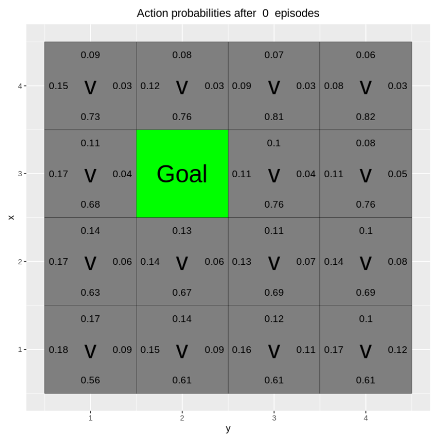
The plots screen shots as below.

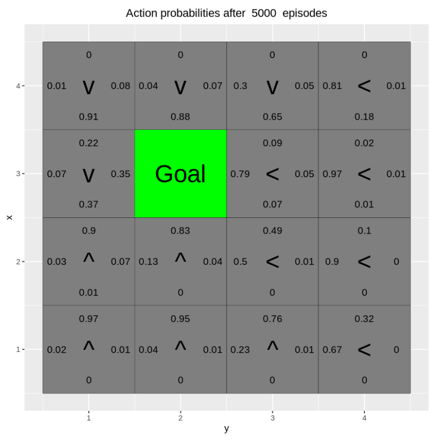


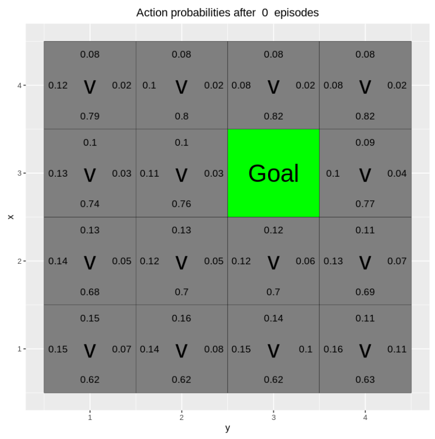


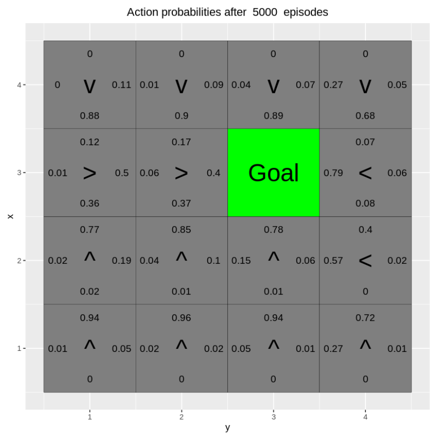

Action probabilities after 0 episodes
Goal
x
y

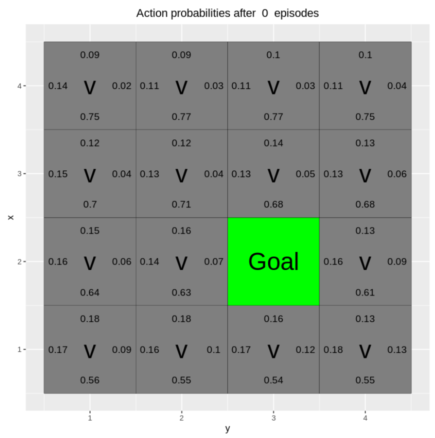

Action probabilities after 5000 episodes
Goal
x
y

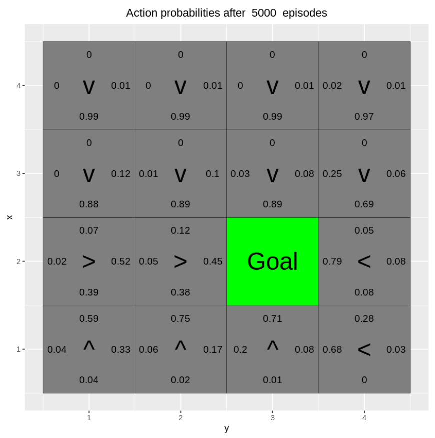

Action probabilities after 0 episodes
Goal
x
y

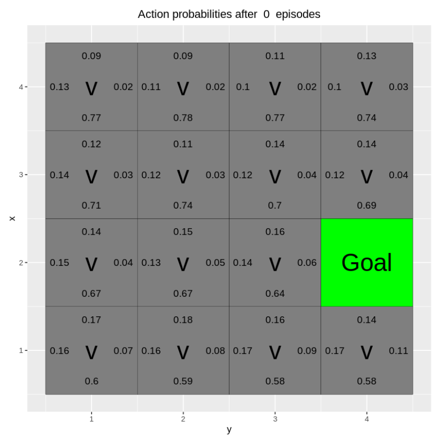

Action probabilities after 5000 episodes
Goal
x
y

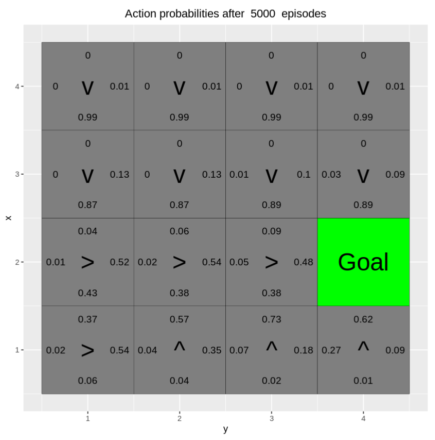

Action probabilities after 0 episodes
Goal
x
y

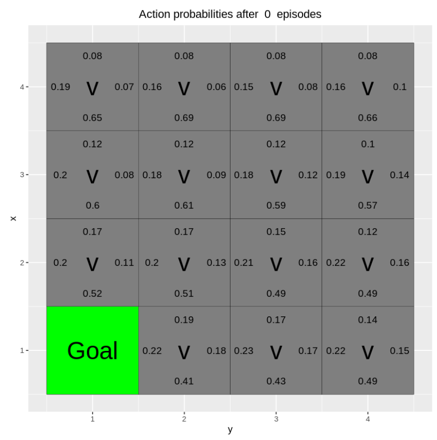

Action probabilities after 5000 episodes
Goal
x
y

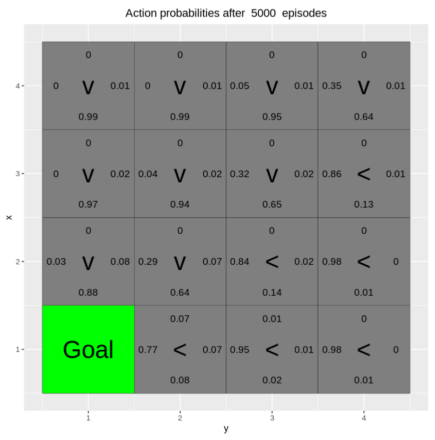

Action probabilities after 0 episodes
Goal
x
y

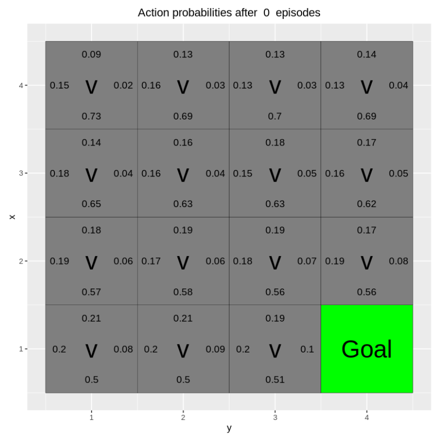

Action probabilities after 5000 episodes
Goal
x
y

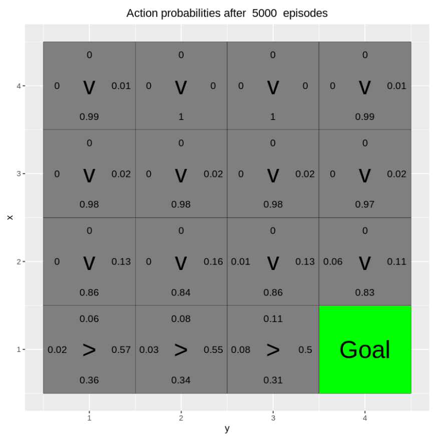

Action probabilities after 0 episodes
Goal
x
y


Action probabilities after 5000 episodes
Goal
x
y


Action probabilities after 0 episodes
Goal
x
y


Action probabilities after 5000 episodes
Goal
x
y





According to the plots provided, and compare the result of val_goals(which is real location of reward) and the result of RF given the init position(train_goals), after checking all the arrows, we can say that the agent has learned a good policy(for almost all the states).

We also notice that the agent find the optimal path by minimize the categorical_crossentropy and using sgd as optimizer.

Q-Learning is not suitable to solve this dynamic problem. Q-Learning is a model-free method ,and it is more suitable to handle small state space, with not so sparse rewards, and the goal position is known in advance.

In our case, since there is only one known reward in the environment, which means it is a sparse reward matrix, the agent has to rely on exploration to find the goal, when the dimension of the grid increases, the agent needs more steps to find the goal position. when dimension increases, q-learning needs more episodes to find the goal position, meanwhile, we also need to have all goal positions in our training data. However, we only have a tiny 4x4 grid, so it's ok to use q-learning in our case, but it is inefficient.

Another problem is start position is not fixed(sampled 5000 times), in this dynamic environment, Q-learning doesn't training well in every state, to get a good performance, we may need more episodes to train the agent, which is not practical.

So use function approximation method like DQN(Deep Q-Networks) is more suitable for this kind of tasks.

6. Environment E

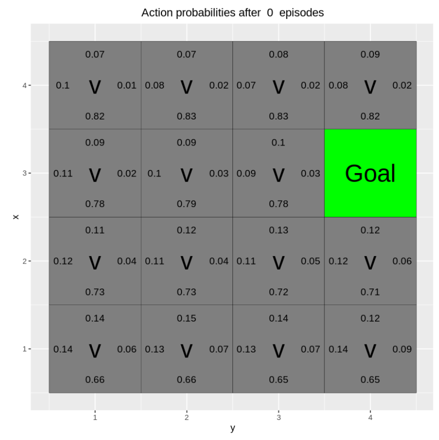
Question

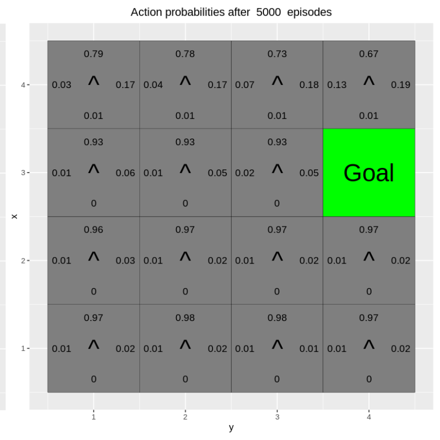
In this task, the goals for training are all from the top row of the grid. The validation goals are three positions from the rows below. To solve this task, simply study the code and results provided in the file RL Lab2 Colab.ipynb and answer the following questions:

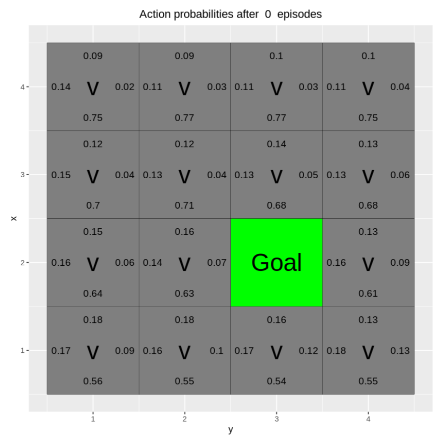
- 1. Has the agent learned a good policy? Why / Why not ?
- 2. If the results obtained for environments D and E differ, explain why.

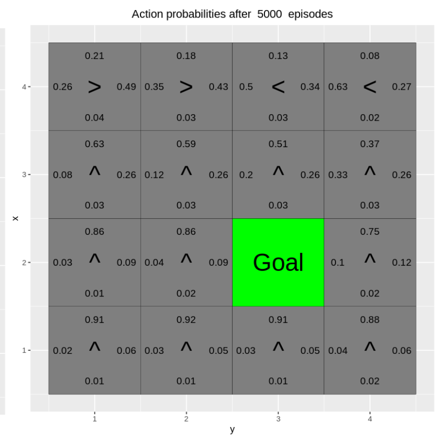
Answer

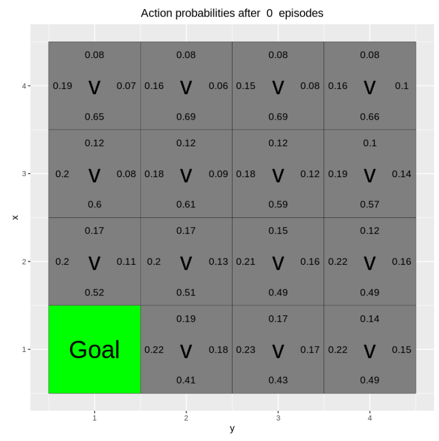
The plots screen shots as below.

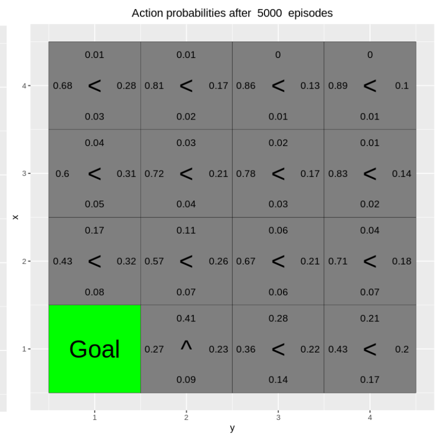












From the result, we found that the agent did not learned a good policy.

The reason behind this is that agent only trained with the goal position from the top row, so the agent only learned to find the goal position from very limited states. while the validation goals are all from the bottom row, so the agent did not learn to find a good policy.

On the other hand, the policy learned in environment D is better, since it trains with different start points, and different goal points across the grid, so the agent learned to find the goal position from different states, which is more general.

Appendix: All code for this report

```r
###########################  Init code For Assignment #########################
rm(list = ls())
knitr::opts_chunk$set(echo = TRUE)
library(ggplot2)
library(gridExtra)
###########################  Common Codes #####################################
MovingAverage <- function(x, n){
  cx <- c(0,cumsum(x))
  rsum <- (cx[(n+1):length(cx)] - cx[1:(length(cx) - n)]) / n
  return (rsum)
}
###########################  Code For Q-Learning Implementation ################
# By Jose M. Peña and Joel Oskarsson.
# For teaching purposes.
# jose.m.pena@liu.se.

#####################################################################################
# Q-learning
#####################################################################################

# install.packages("ggplot2")
# install.packages("vctrs")
library(ggplot2)

arrows <- c("^", ">", "v", "<")
action_deltas <- list(c(1,0), # up
                      c(0,1), # right
                      c(-1,0), # down
                      c(0,-1)) # left

vis_environment <- function(iterations=0, epsilon = 0.5, alpha = 0.1,
                            gamma = 0.95, beta = 0){

  # Visualize an environment with rewards.
  # Q-values for all actions are displayed on the edges of each tile.
  # The (greedy) policy for each state is also displayed.
  #
  # Args:
  # iterations, epsilon, alpha ,gamma, beta (optional): for the figure title.
  # reward_map (global variable):a HxW array containing the reward given at each
  #                               state.
  # q_table (global variable):a HxWx4 array containing Q-values for each
  #                               state-action pair.
  # H, W (global variables): environment dimensions.

  df <- expand.grid(x=1:H,y=1:W)

  # calculate Q-values for each action at every position
  # for terminal states,reward states,penalty states, Q-values are NA
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,
                                    q_table[x,y,1],NA),df$x,df$y)
  df$val1 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,
                                     q_table[x,y,2],NA),df$x,df$y)
  df$val2 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,
                                     q_table[x,y,3],NA),df$x,df$y)
  df$val3 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,
                                     q_table[x,y,4],NA),df$x,df$y)
  df$val4 <- as.vector(round(foo, 2))

  # Policy Best Direction or Reward
  foo <- mapply(function(x,y)
    ifelse(reward_map[x,y] == 0,arrows[GreedyPolicy(x,y)],reward_map[x,y]),df$x,df$y)
  df$val5 <- as.vector(foo)

  # Grid Colouring
  # Negative rewards are NA,which are coloured red(according to na.value below)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,max(q_table[x,y,]),
                                     ifelse(reward_map[x,y]<0,NA,reward_map[x,y])),df$x,df$y)
  df$val6 <- as.vector(foo)

  print(ggplot(df,aes(x = y,y = x)) +
          scale_fill_gradient(low = "white", high = "green", na.value = "red", name = "") +
          geom_tile(aes(fill=val6)) +
          geom_text(aes(label = val1),size = 2,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 2,nudge_x = .35,na.rm = TRUE) +
          geom_text(aes(label = val3),size = 2,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 2,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 5) + # arrow or number in centre
          geom_tile(fill = 'transparent', colour = 'black') +
          ggtitle(paste("Q-table after ",iterations," iterations\n",
                        "(epsilon = ",epsilon,", alpha = ",alpha,",gamma = ",
                         gamma,", beta = ",beta,")")) +
          theme(plot.title = element_text(hjust = 0.5),
                axis.text.x = element_text(size = 5),
                axis.text.y = element_text(size = 5),
                legend.text = element_text(size = 5),) +
          scale_x_continuous(breaks = c(1:W),labels = c(1:W)) +
          scale_y_continuous(breaks = c(1:H),labels = c(1:H)))
}
GreedyPolicy <- function(x, y){
  # Get a greedy action for state (x,y) from q_table.
  # Args:
  #   x, y: state coordinates.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.
  #

  # Find the actions that corresponding to the highest Q-values at the state
  # and choose one randomly if there are more than one suitable actions.
  actions <- which(q_table[x,y,] == max(q_table[x,y,]))
  # must check like this, directly sample(actions, size = 1) will have error
  if(length(actions) > 1){
    return(sample(actions, size = 1))
  }
  return (actions)
}
EpsilonGreedyPolicy <- function(x, y, epsilon){
  # Get an epsilon-greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   epsilon: probability of acting randomly.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # if random number < epsilon then do random action
  if (runif(1) < epsilon)
    return (sample(1:4, size = 1))
  else
    return (GreedyPolicy(x, y))
}
transition_model <- function(x, y, action, beta){
  # Computes the new state after given action is taken. The agent will follow the action
  # with probability (1-beta) and slip to the right or left with probability beta/2 each.
  #
  # Args:
  #   x, y: state coordinates.
  #   action: which action the agent takes (in {1,2,3,4}).
  #   beta: probability of the agent slipping to the side when trying to move.
  #   H, W (global variables): environment dimensions.
  #
  # Returns:
  #   The new state after the action has been taken.

  delta <- sample(-1:1, size = 1, prob = c(0.5*beta, 1-beta, 0.5*beta))

  # sample:
  # action = up(1),delta = turn left(-1) => left(4)
  # action = right(2),delta = turn left(-1) => up(1)
  final_action <- ((action + delta + 3) %% 4) + 1

  # get new position
  foo <- c(x,y) + unlist(action_deltas[final_action])

  # make sure the new position is within the environment
  foo <- pmax(c(1,1),pmin(foo,c(H,W)))
  return (foo)
}
q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                       beta = 0){

  # Perform one episode of Q-learning. The agent should move around in the
  # environment using the given transition model and update the Q-table.
  # The episode ends when the agent reaches a terminal state.
  #
  # Args:
  #   start_state: array with two entries, describing the starting position of the agent.
  #   epsilon (optional): probability of acting randomly.
  #   alpha (optional): learning rate.
  #   gamma (optional): discount factor.
  #   beta (optional): slipping factor.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   reward: reward received in the episode.
  #   correction: sum of the temporal difference correction terms over the episode.
  #   q_table (global variable): Recall that R passes arguments by value. So, q_table being
  #   a global variable can be modified with the superassigment operator <<-.

  # init current position
  current_position <- start_state
  episode_correction <- 0

  repeat{
    # Follow policy, execute action, get reward.
    # choose A from S using policy derived from Q (e.g., epsilon-greedy)
    action <- EpsilonGreedyPolicy(current_position[1], current_position[2], epsilon)
    # get new position
    new_position <- transition_model(current_position[1], current_position[2], action, beta)
    # get new reward
    reward <- reward_map[new_position[1], new_position[2]]
    # get current Q-value
    Q <- q_table[current_position[1], current_position[2], action]

    # update formula:Page 13 of RL_L1_QLearning.pdf
    # q_{*}(s,a) <- q_{*}(s,a) + alpha * (r + gamma * max_{a'} q_{*}(s',a') - q_{*}(s,a))
    correction_term <- reward + gamma * max(q_table[new_position[1],
                                        new_position[2],]) - Q

    # accumulates the sum of all difference correction terms to to plot later
    # it can be removed if not needed
    episode_correction <- episode_correction + correction_term

    # Q-table update.
    q_table[current_position[1], current_position[2], action] <<- Q + alpha * correction_term

    current_position <- new_position

    if(reward!=0)
      # End episode.
      return (c(reward,episode_correction))
  }
}
###########################  Code For env-a ##############################
```

```r
# Environment A (learning)
H <- 5
W <- 7

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[3,6] <- 10
reward_map[2:4,3] <- -1

q_table <- array(0,dim = c(H,W,4))
# init env plot
# epsilon = 0.5, alpha = 0.1, gamma = 0.95, beta = 0 is default value
vis_environment()
#init plot list
for(i in 1:10000){
  foo <- q_learning(start_state = c(3,1))

  if(any(i==c(10,100,1000,10000)))
    vis_environment(i)
}
###########################  Code For env-b ##############################
H <- 7
W <- 8

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,] <- -1
reward_map[7,] <- -1
reward_map[4,5] <- 5
reward_map[4,8] <- 10

q_table <- array(0,dim = c(H,W,4))
# init env plot
vis_environment()
# default epsilon = 0.5
for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }
  vis_environment(i, gamma = j)
  par(mfrow = c(1,2))
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
  par(mfrow = c(1,1))
}
# change epsilon to 0.1
for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(epsilon = 0.1, gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }

  vis_environment(i, epsilon = 0.1, gamma = j)
  par(mfrow = c(1,2))
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
  par(mfrow = c(1,1))
}
###########################  Code For env-c ##############################
H <- 3
W <- 6

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,2:5] <- -1
reward_map[1,6] <- 10
q_table <- array(0,dim = c(H,W,4))
vis_environment()
# different beta = 0, 0.2, 0.4, 0.66
for(j in c(0,0.2,0.4,0.66)){
  # init q_table to 0
  q_table <- array(0,dim = c(H,W,4))
  for(i in 1:10000)
    foo <- q_learning(gamma = 0.6, beta = j, start_state = c(1,1))
  vis_environment(i, gamma = 0.6, beta = j)
}
knitr::include_graphics("images/1.png")
knitr::include_graphics("images/2.png")
knitr::include_graphics("images/3.png")
knitr::include_graphics("images/4.png")
knitr::include_graphics("images/5.png")
knitr::include_graphics("images/6.png")
knitr::include_graphics("images/7.png")
knitr::include_graphics("images/8.png")
knitr::include_graphics("images/9.png")
knitr::include_graphics("images/10.png")
knitr::include_graphics("images/11.png")
```