Lab3 Individual Report

varsi146

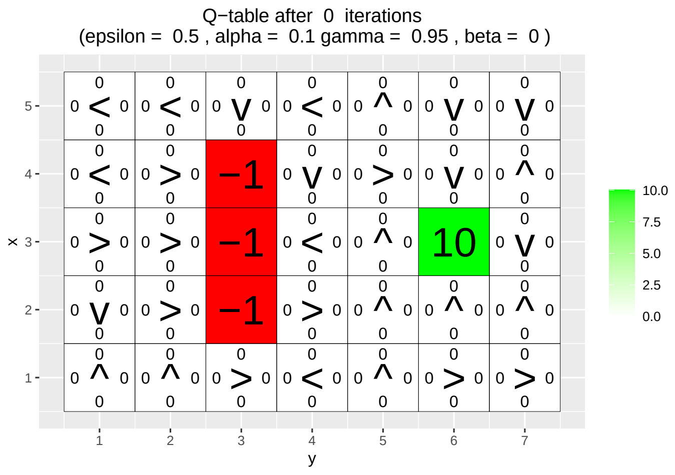
2023-10-18

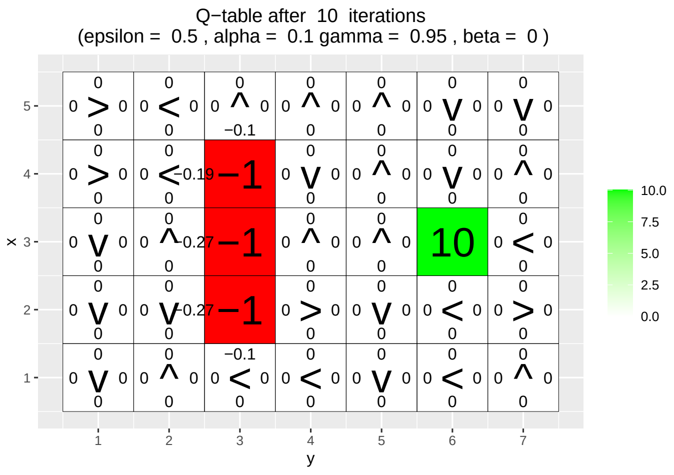
```r
GreedyPolicy <- function(x, y){

  # 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}.

  # Your code here.

  # We first retrieve the q-value from the table for given state coordinates
  q_value <- q_table[x,y,]

  # Choose the maximum q-value
  max_qVal <- max(q_value)

  # To handle ties in the scenario
  # We first check which for maximum qvalues
  max_action <- which(q_value == max_qVal)

  # Sample to break tie by choosing an action randomly
  if (length(max_action) > 1) {
    chosen_action <- sample(max_action, 1)
  }else{
    chosen_action <- max_action
  }
  return(chosen_action)
}

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}.
  # Your code here.

  if (runif(1) < epsilon) {
    chosen_action <- sample(x = c(1:4), size = 1)
    return(chosen_action)
  }else{
    return(GreedyPolicy(x,y))
  }
}

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 <<-.

  # Your code here.
  # Initializing state S
  current_state <- start_state
  reward <- 0
  episode_correction <- 0
  repeat{
    # Follow policy, execute action, get reward.
    # determine an action via Epsilon-Greedy method
    action <- EpsilonGreedyPolicy(x = current_state[1],
                                  y = current_state[2], epsilon = epsilon)
    # After taking an action, observing next state and reward
    next_state <- transition_model(x = current_state[1],
                                   y = current_state[2], action = action,
                                   beta = beta)
    next_reward <- reward_map[next_state[1], next_state[2]]
    # Q-table update.
    q_current <- q_table[current_state[1],current_state[2],action]
    max_a_qnext <- max(q_table[next_state[1],next_state[2],])
    tempDiff_err <- next_reward + ((gamma*max_a_qnext) - q_current)
    # Combining terms above for Q-table update:
    q_table[current_state[1],
            current_state[2], action] <<- q_current + (alpha * tempDiff_err)

    # Accumulate the temporal difference correction for this episode
    episode_correction <- episode_correction + tempDiff_err

    # Move to the next state
    current_state <- next_state

    # Update the episode reward
    reward <- reward + next_reward

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

}
```

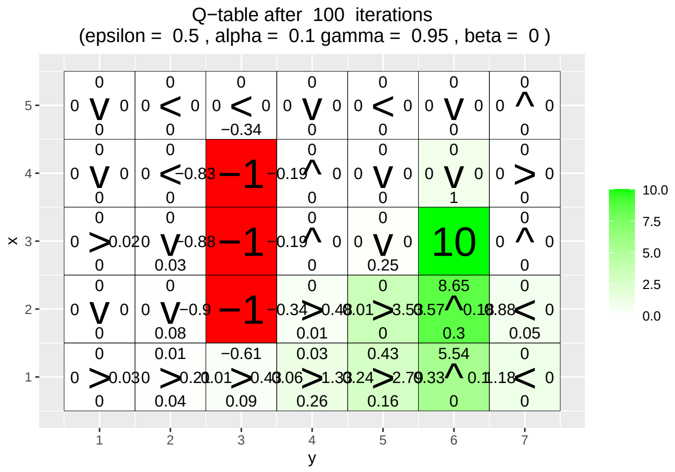
Environment A

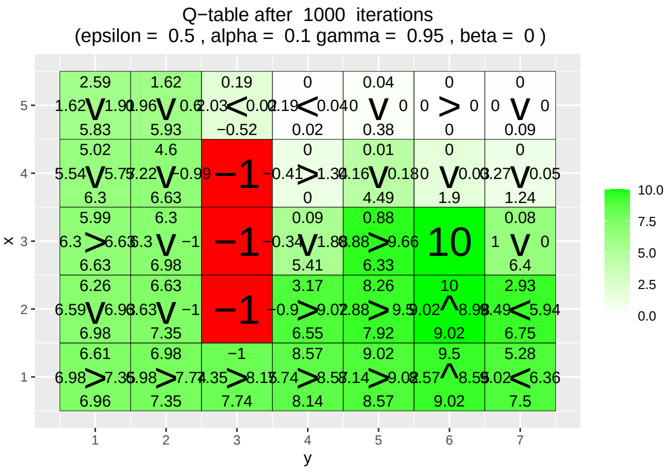
Environment A (learning)

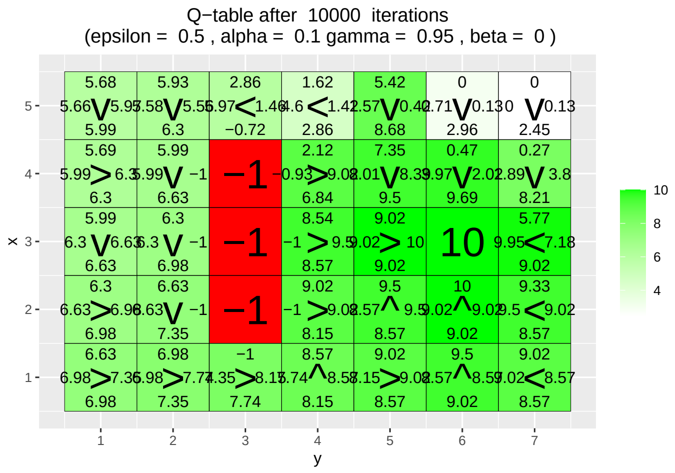
Sub Question 1 : What has the agent learned after the first 10 episodes ? We see below that after 10 iterations, the agent has only learned to avoid the states with a reward of -1 by assigning negative discounted expected reward if an action is leading to that state.











Sub Question 2 : Is the final greedy policy (after 10000 episodes) optimal for all states, i.e. not only for the initial state ? Why / Why not ? Here, we define the optimal policy as a policy where the agent with the objective of reaching the terminal state always chooses an action that maximizes the cumulative discounted return.

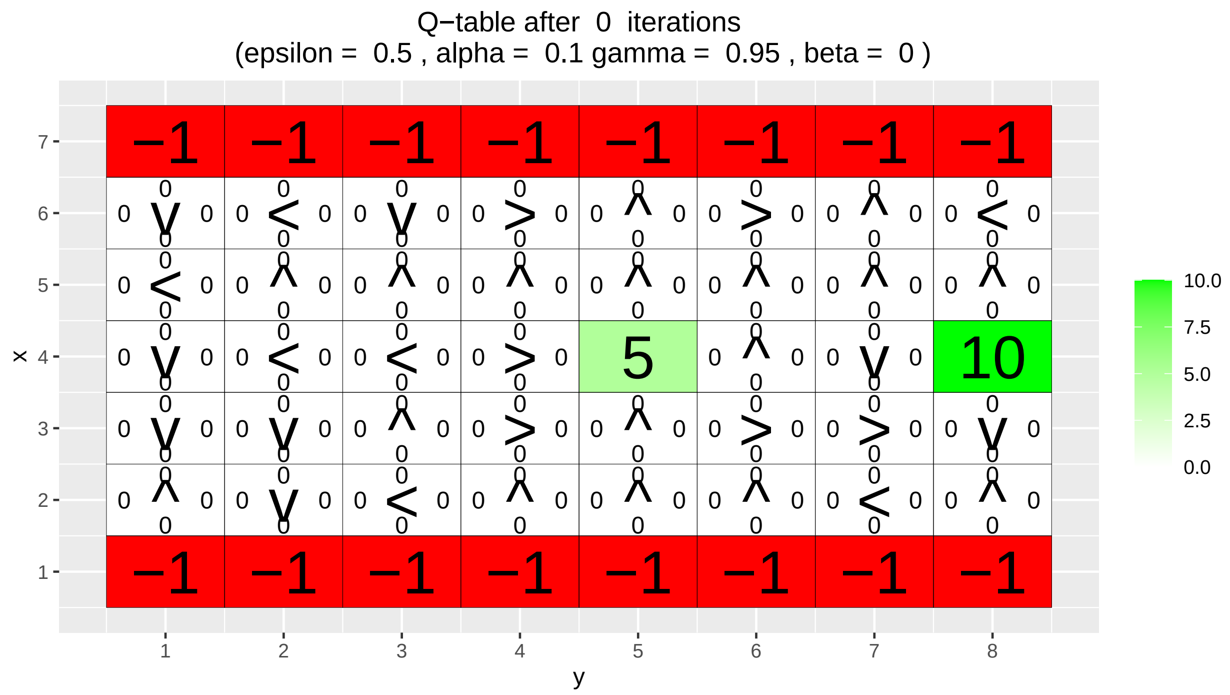
From the environment plot after 10000 iterations, we can see that irrespective of what state the agent starts from, it takes actions that terminate in the terminal state of reward 10 while maximizing the cumulative discounted return.

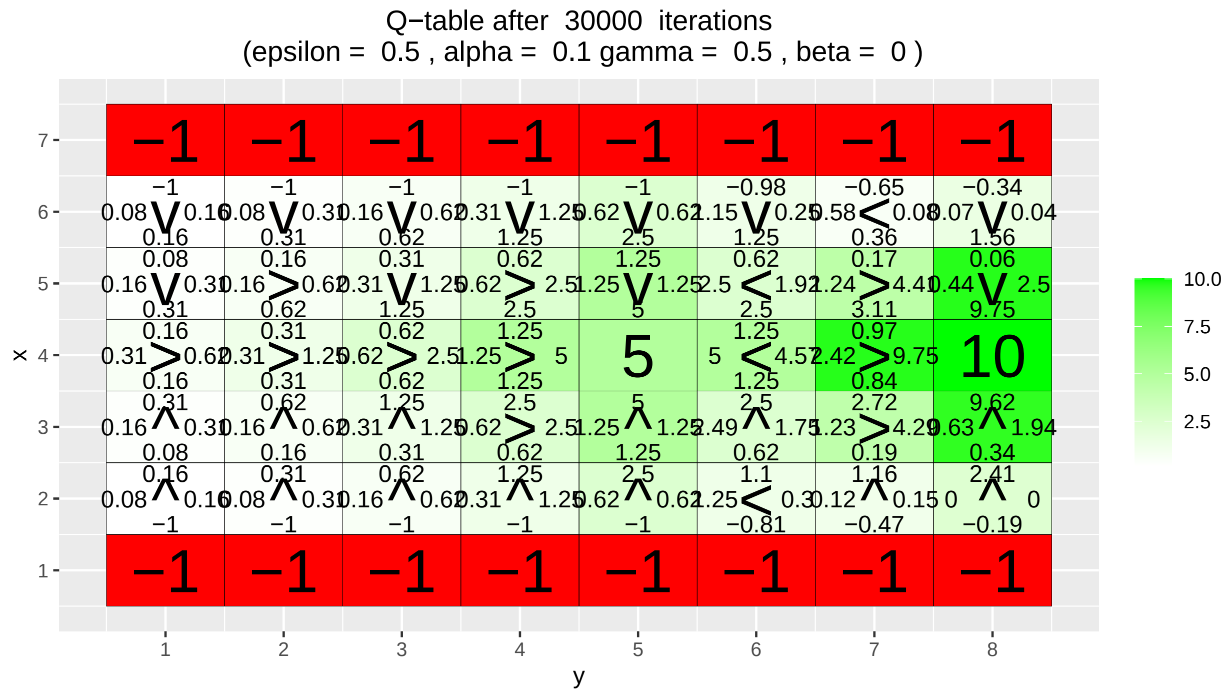
Sub Question 3 : Do the learned values in the Q-table reflect the fact that there are multiple paths (above and below the negative rewards) to get to the positive reward ? If not, what could be done to make it happen ? Assuming that we always start at state (3,1) we can see from the plot that the learned values in the Q-table correspond to a single path i.e., the one below the negative reward block.

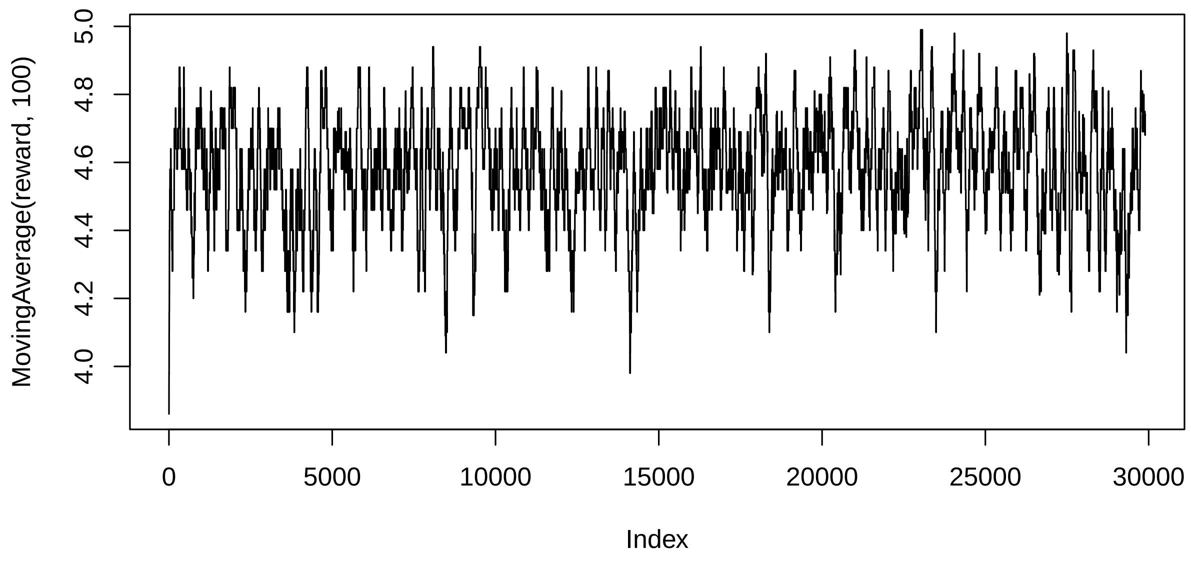
If we do want multiple paths, above and below the negative reward block, we would have to make the agent to be exploratory i.e., by setting a higher ϵ.

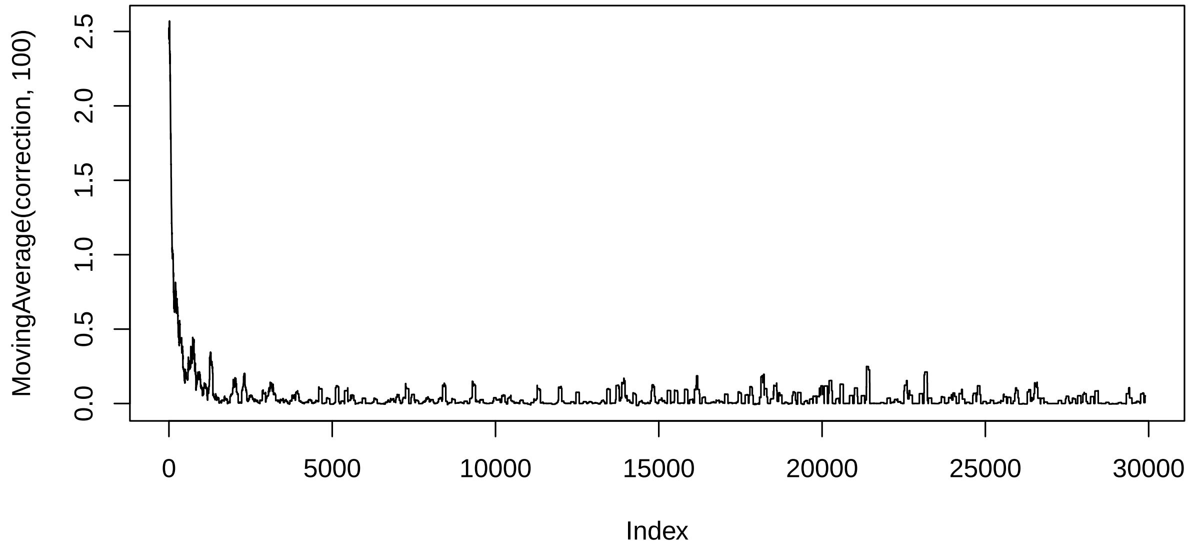
Environment B

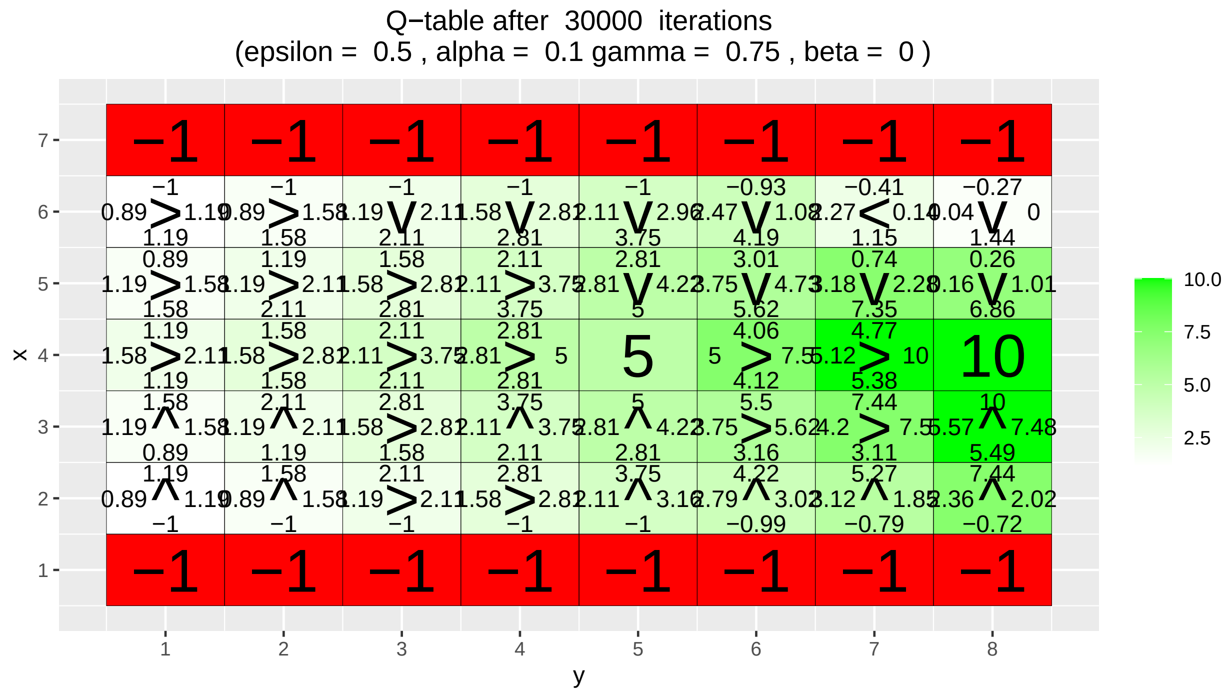
For $\epsilon = 0.5$

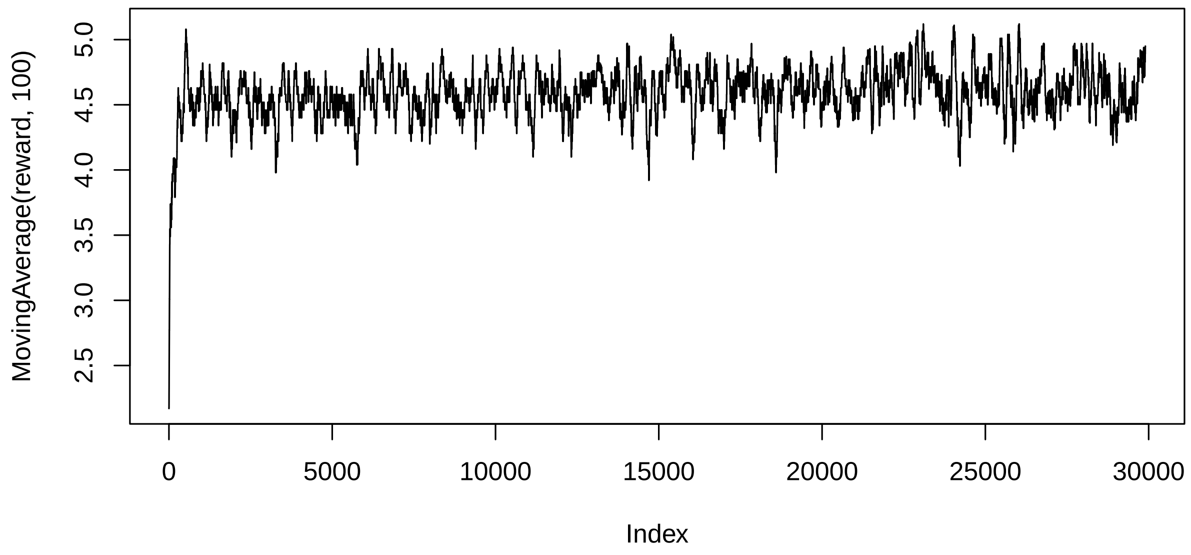


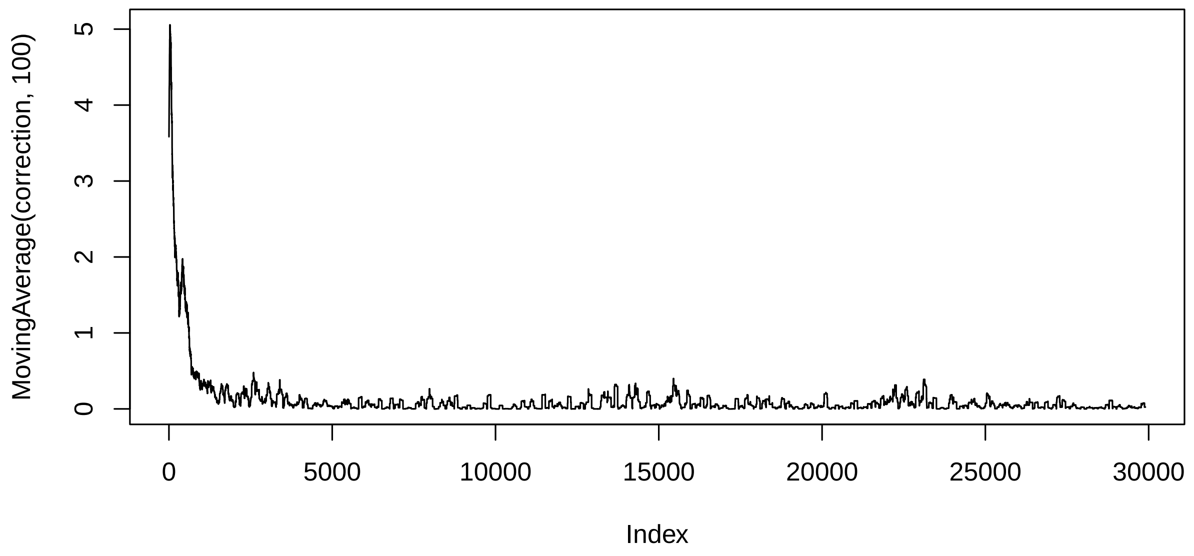


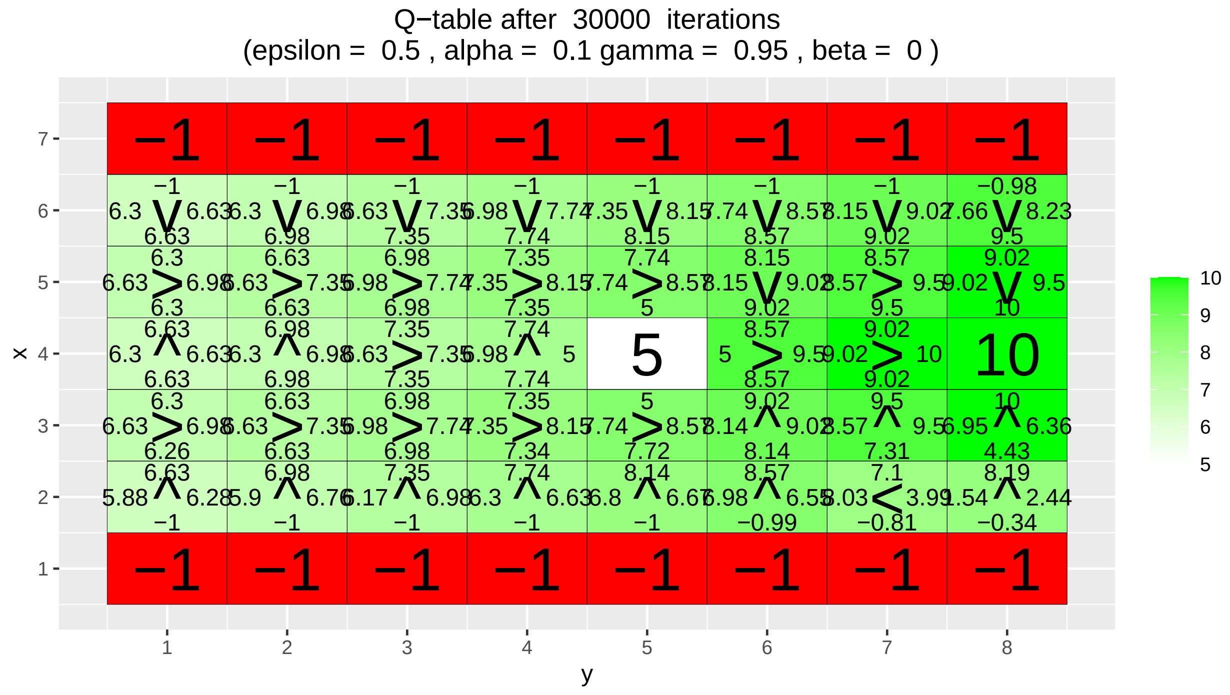

MovingAverage(reward, 100)
4.0
4.2
4.4
4.6
4.8
5.0
0
5000
10000
15000
20000
25000
30000
Index

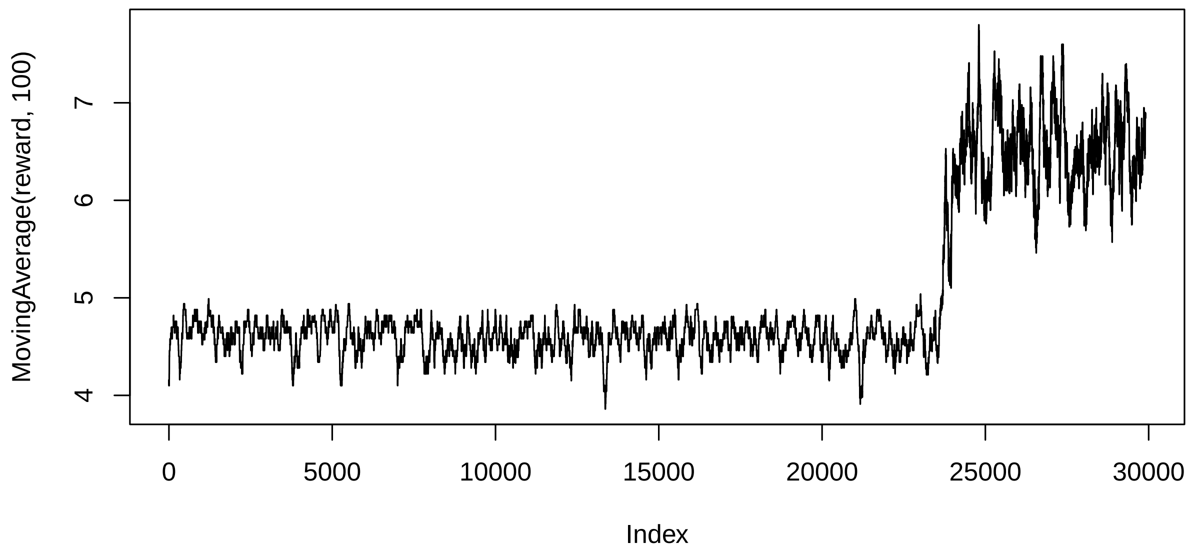

MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
2.0
2.5
0
5000
10000
15000
20000
25000
30000
Index

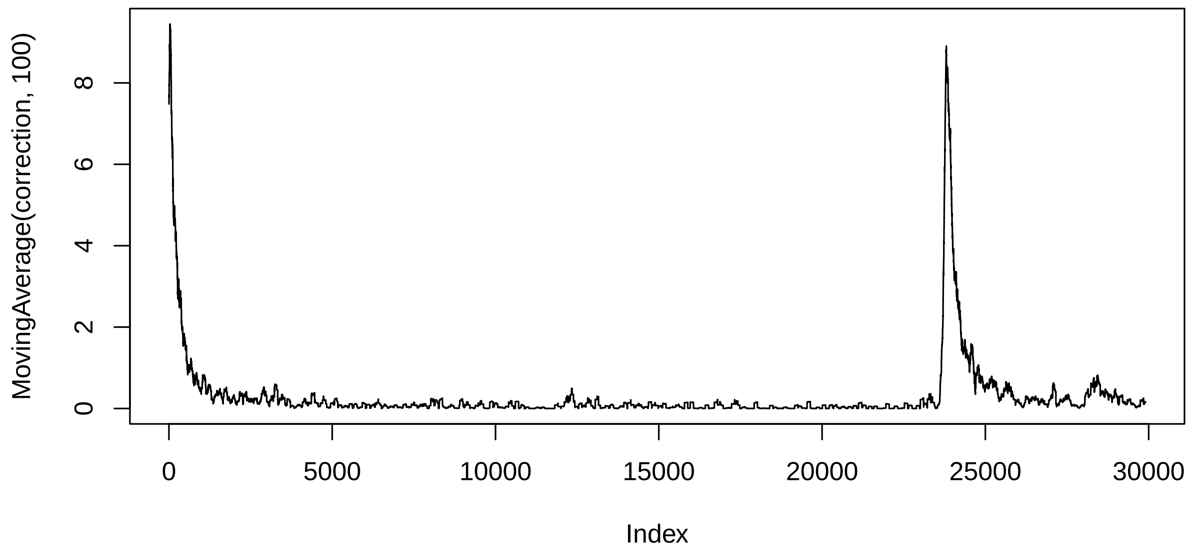

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5


MovingAverage(reward, 100)
Index


MovingAverage(correction, 100)
Index
0
5000
10000
15000
20000
25000
30000
0
1
2
3
4
5


Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
5
10
10
9
8
7
6
5





For $\gamma = 0.5$, $\beta = 0$, $\alpha = 0$ From the Moving average plot of rewards for the given configuration, we can see that agent has on average, learnt the path to reward block 5. This is because the agent prioritizes short term returns more than the long term returns because of the discount factor.

The corrections represents the sum of the difference between the predicted Q-value for a state-action pair and the updated Q-value based on the observed rewards and estimated future rewards. Furthermore, it provides insight into how much the Q-values are being updated during the learning process in each episode.

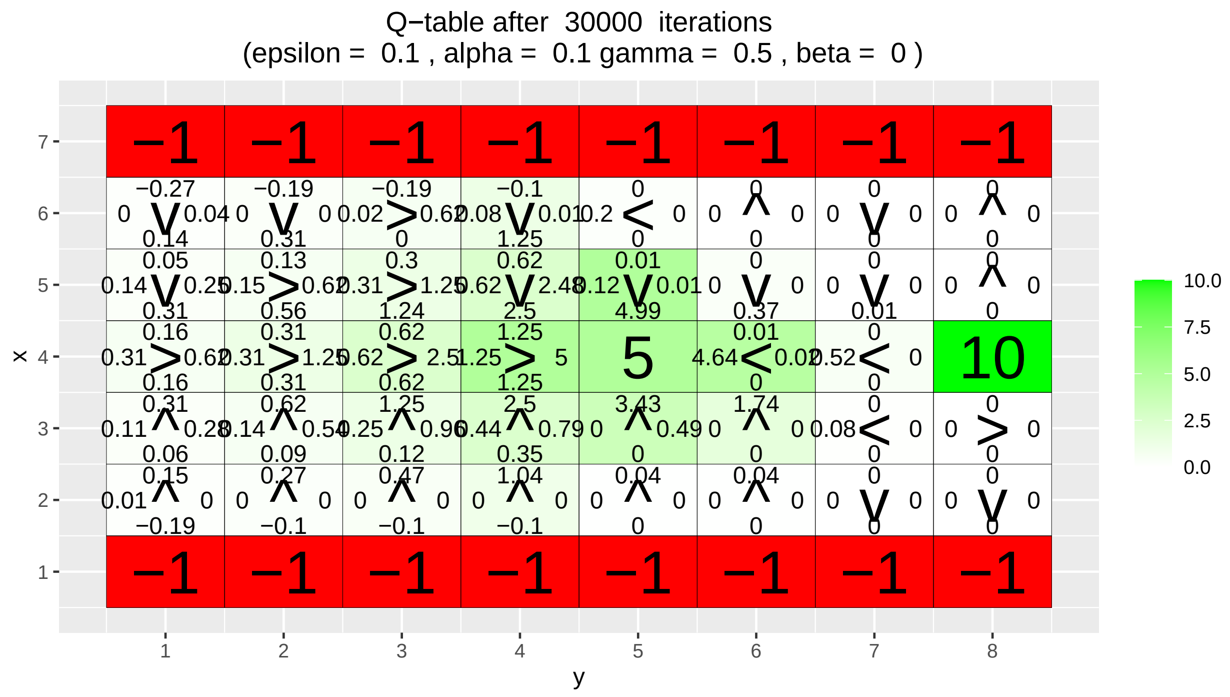
From the Moving average plot of corrections for the given configuration, we can see that in the initial few

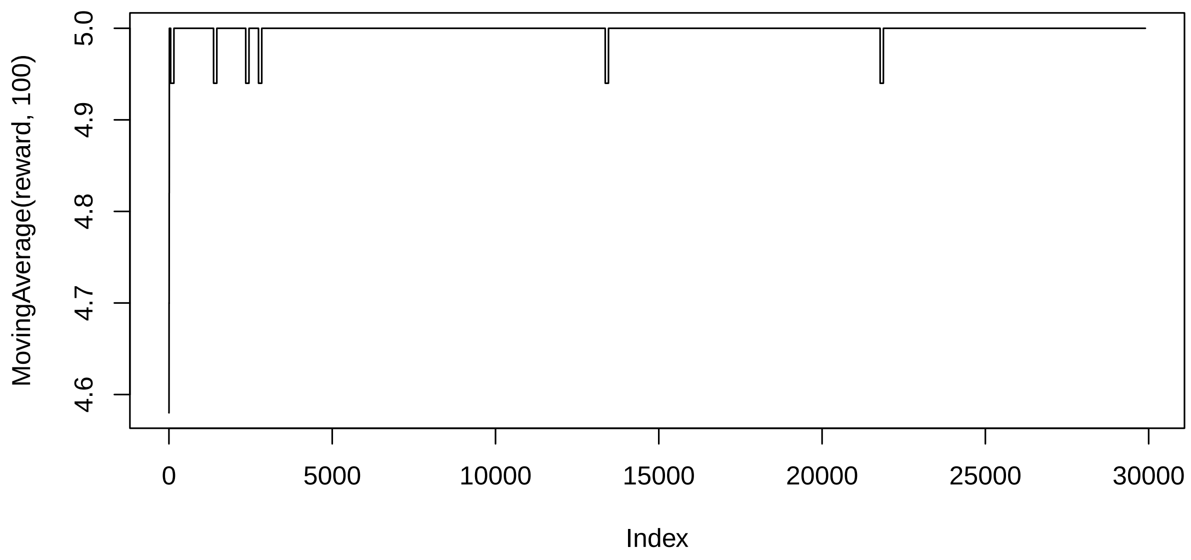
1000 episodes, the correction is very high given that the agent is still learning the optimal policy(Q-table values) signifying unstableness. However, after 5000 episodes, we can say that the q-values have stabilized and we observe only minor fluctuations in the corrections henceforth.

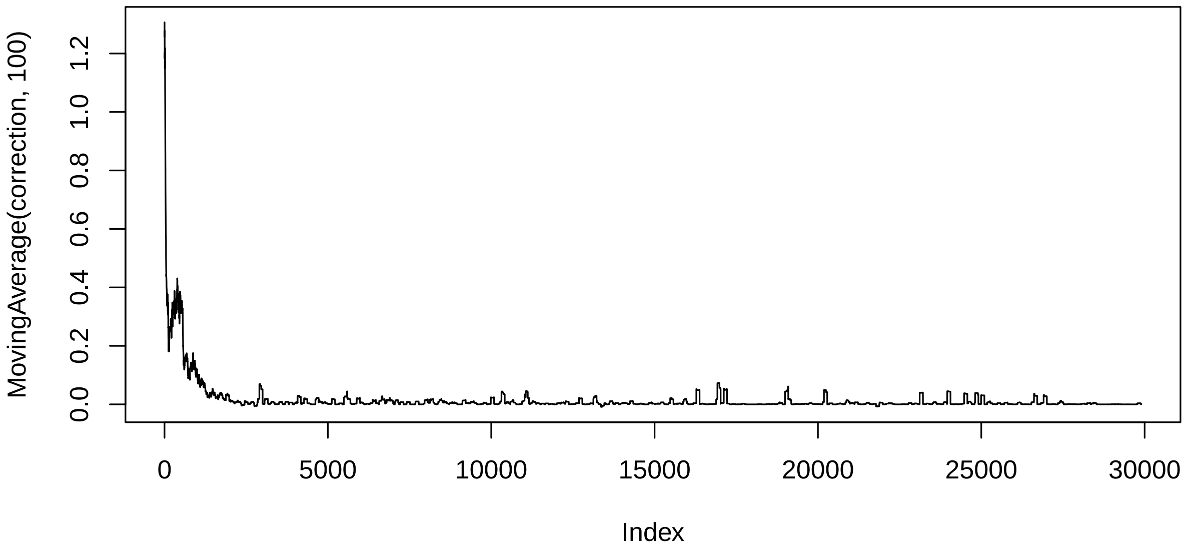
When $\gamma = 0.75$ the behavior of the agent with respect to moving average of rewards and corrections are similar to the previous case. However, now the agent assigns higher q-values to actions leading to reward block 10 in the vicinity of reward block 5. This signifies that the agent is now starting to delay its immediate rewards given the increase in the discount factor.

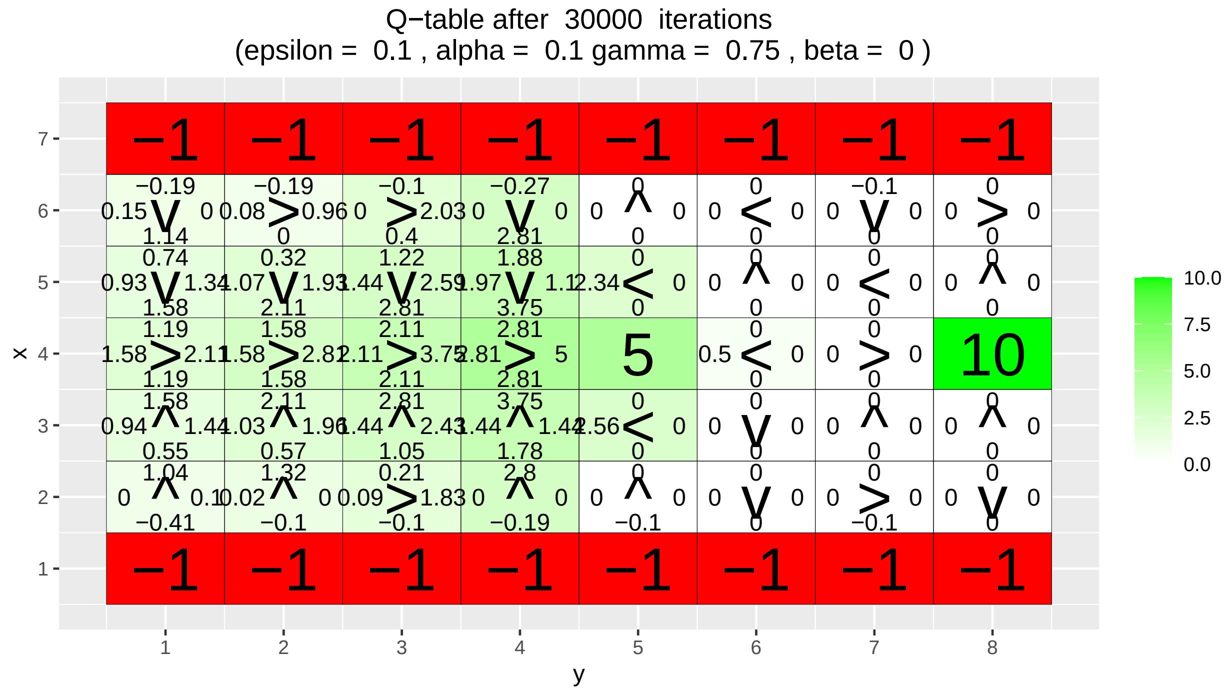
When $\gamma = 0.95$, we can see from the moving average plots for rewards and corrections, that after 2500 episodes the agent starts to acknowledge the terminal state of reward 10. Since, the discount factor is higher the agent does not prioritize immediate rewards i.e., reward block 5 and hence always tries to reach the reward block 10.

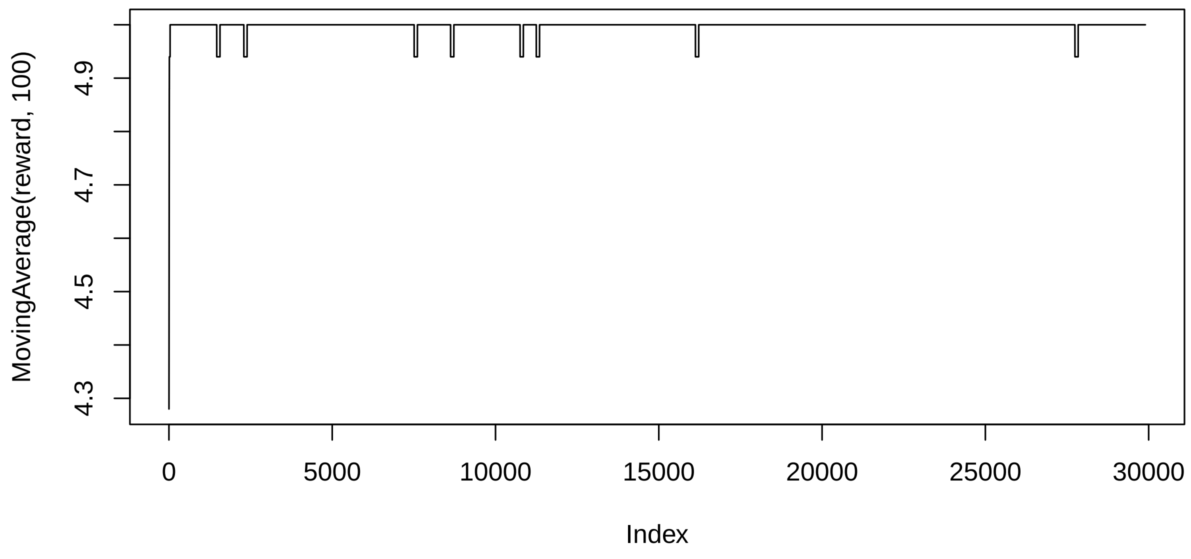
For $\epsilon = 0.1$

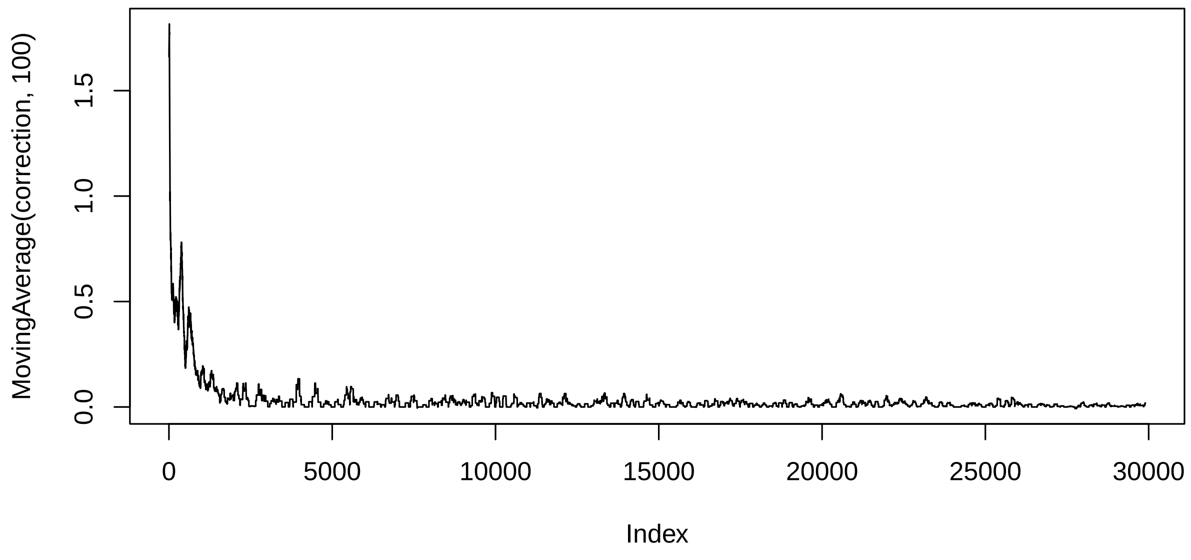




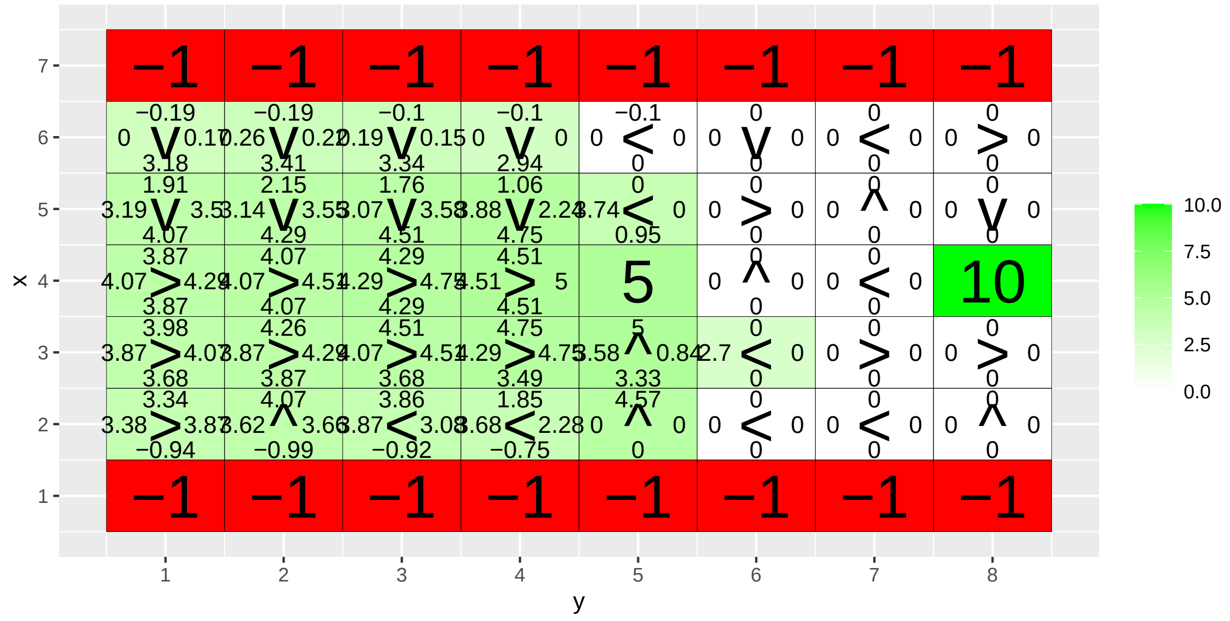

MovingAverage(correction, 100)
Index

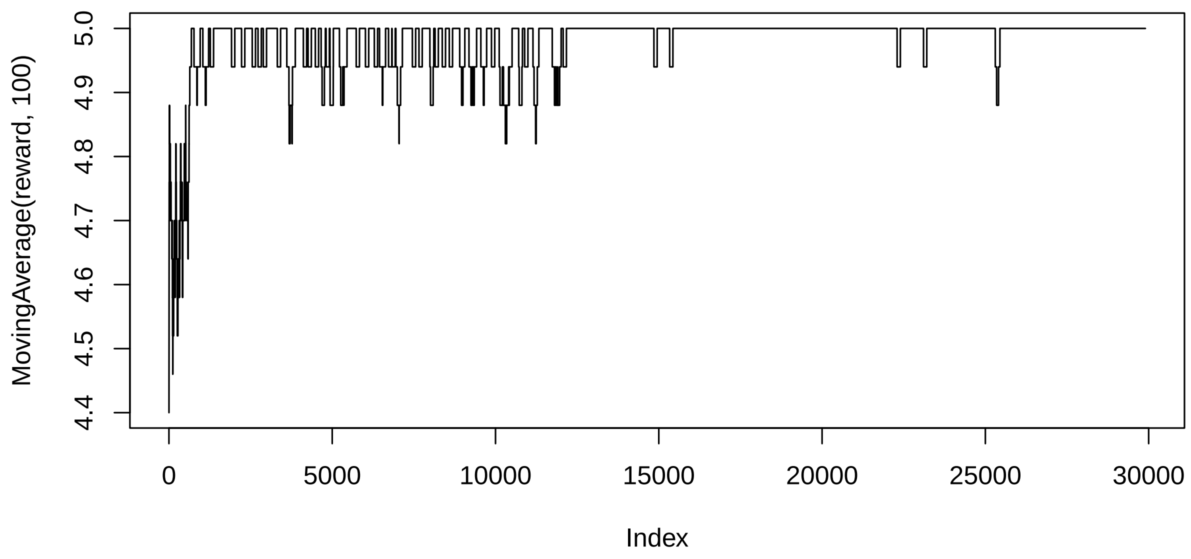

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0

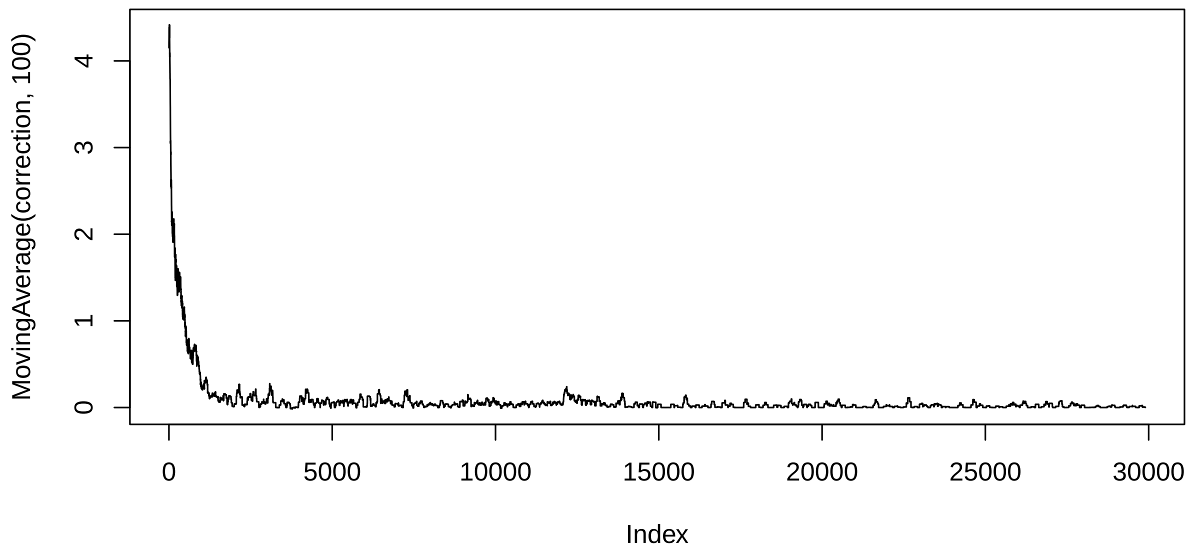

MovingAverage(reward, 100)
4.3
4.5
4.7
4.9
0
5000
10000
15000
20000
25000
30000
Index


MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
0
5000
10000
15000
20000
25000
30000
Index

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.95 , beta = 0)





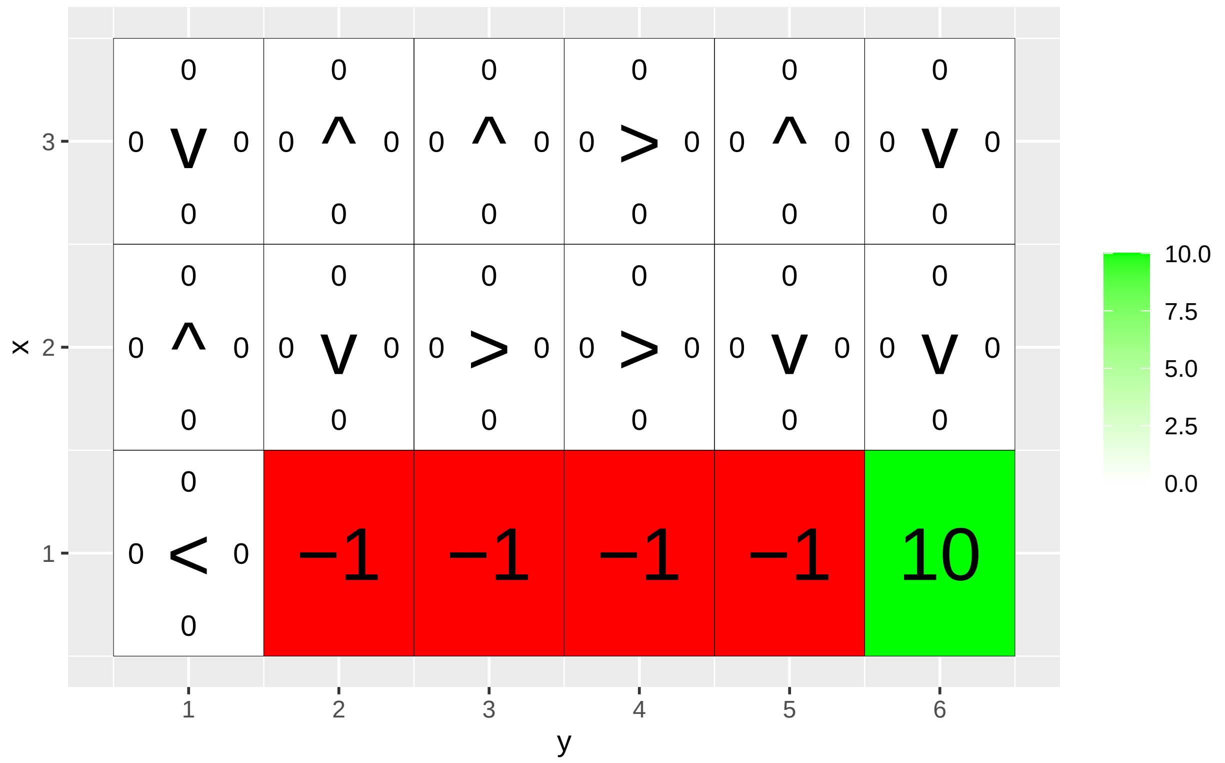


When $\epsilon = 0.1$, the agent becomes non exploratory in the sense that once it reaches the terminal state of reward block 5, it starts to exploit the path with the highest q-values leading to the same reward block and doesn't explore for another terminal state i.e., in this case reward block 10.

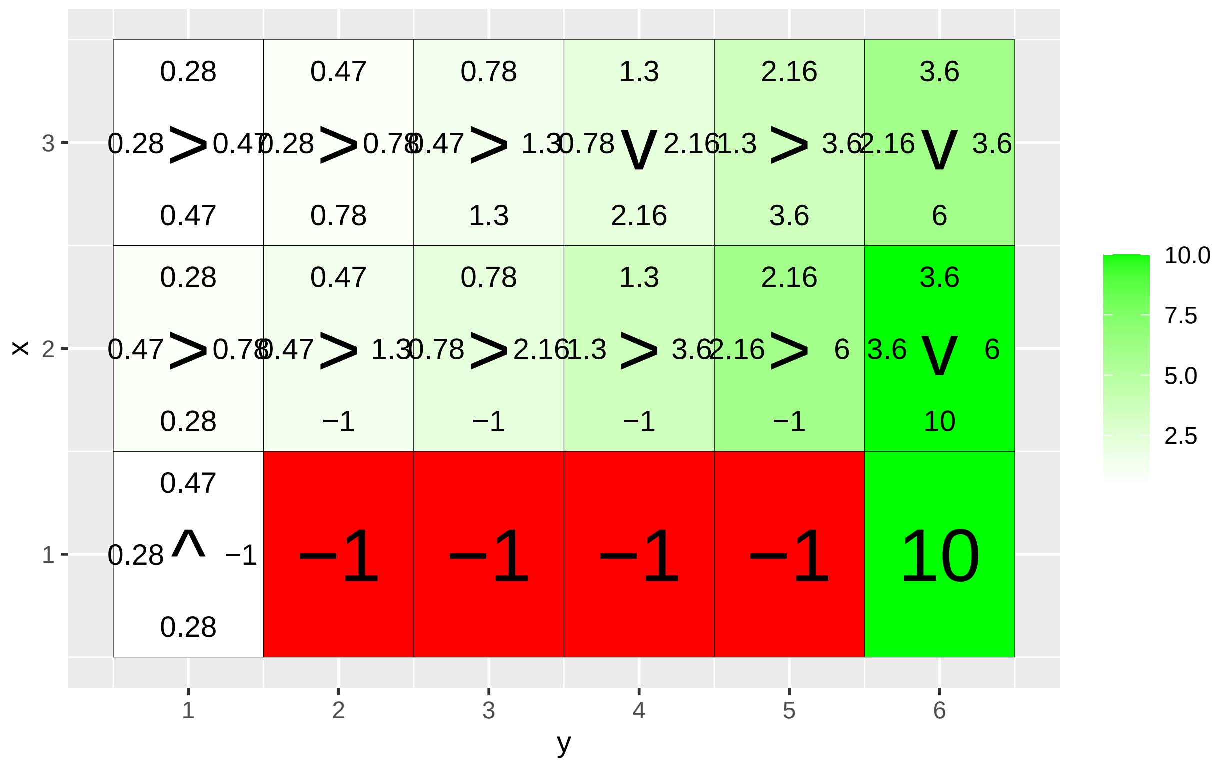
Because of this exploitative nature of the agent, increasing the discount factor doesn't change its behavior as it almost always reaches the terminal state of reward block 5. The same can confirmed from the moving average plots for rewards.

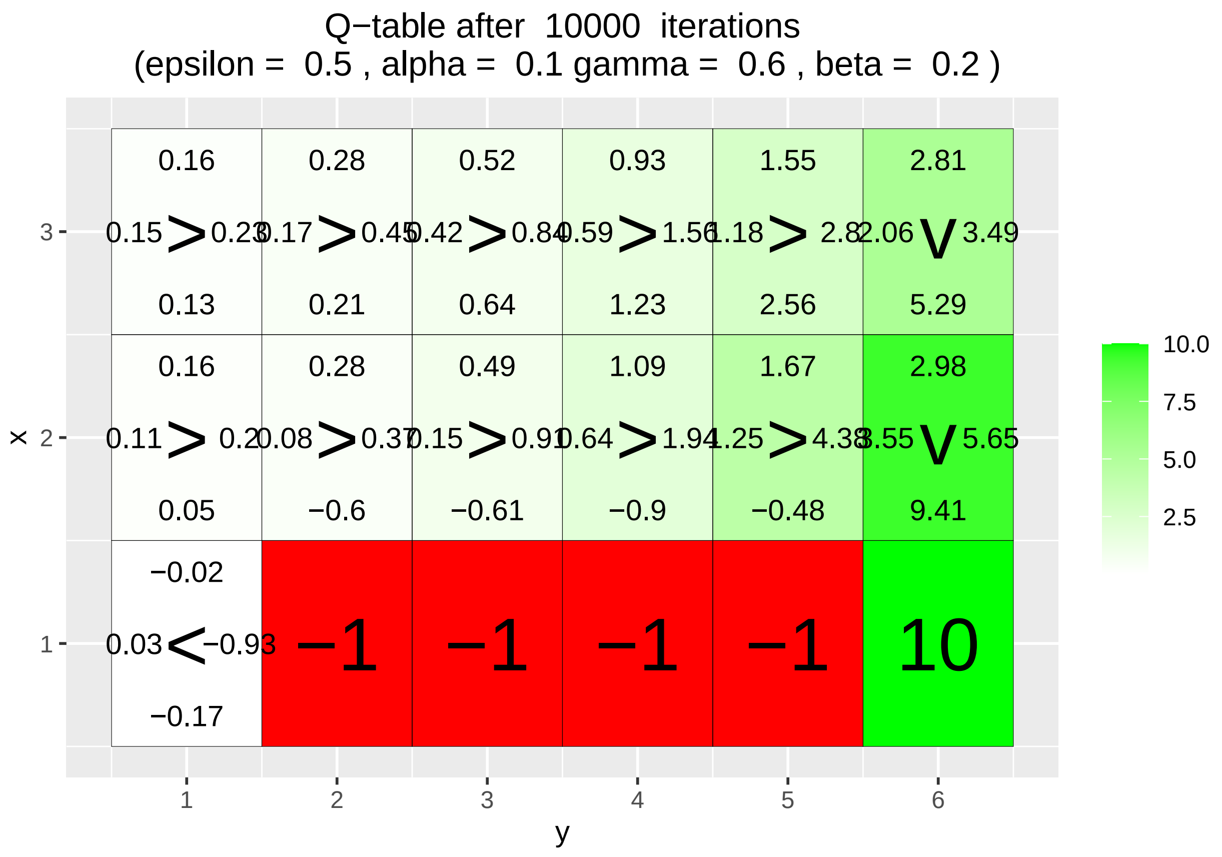
Environment C

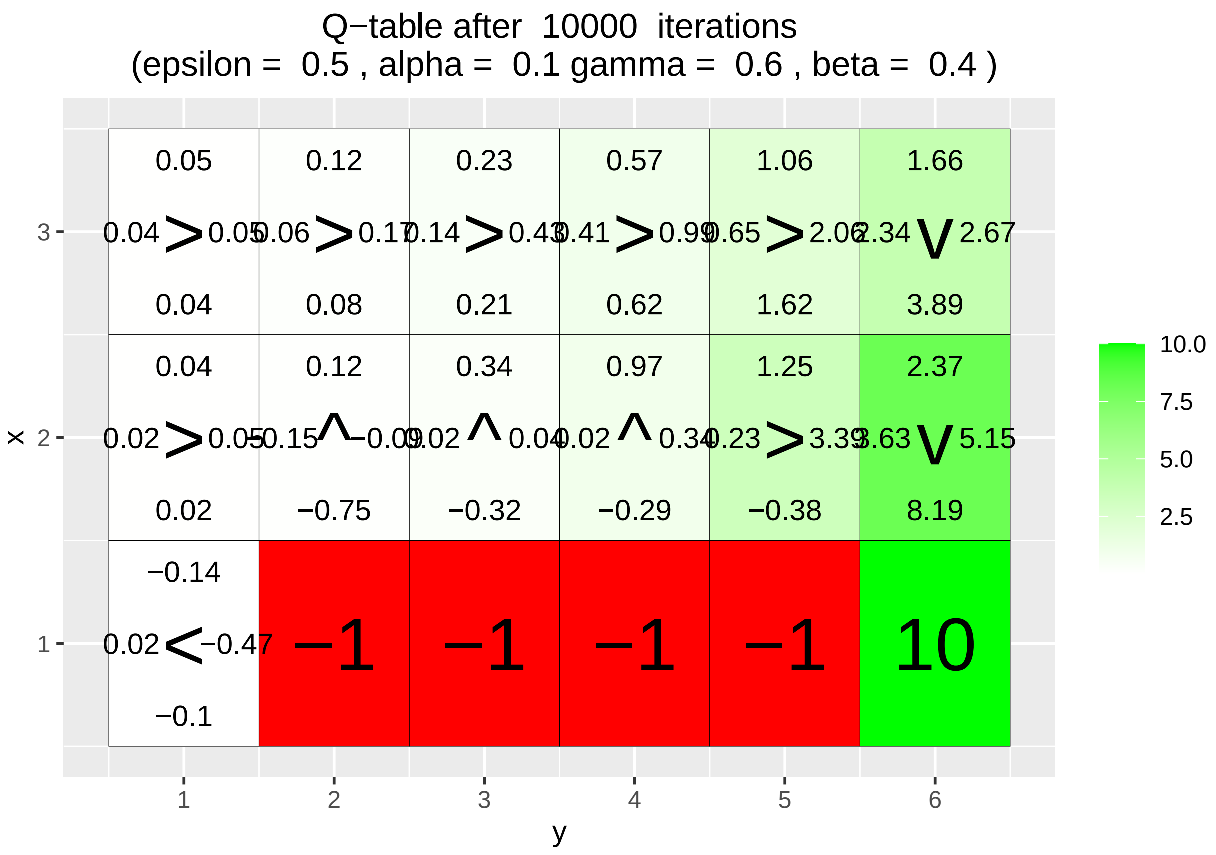
Q−table after 0 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)

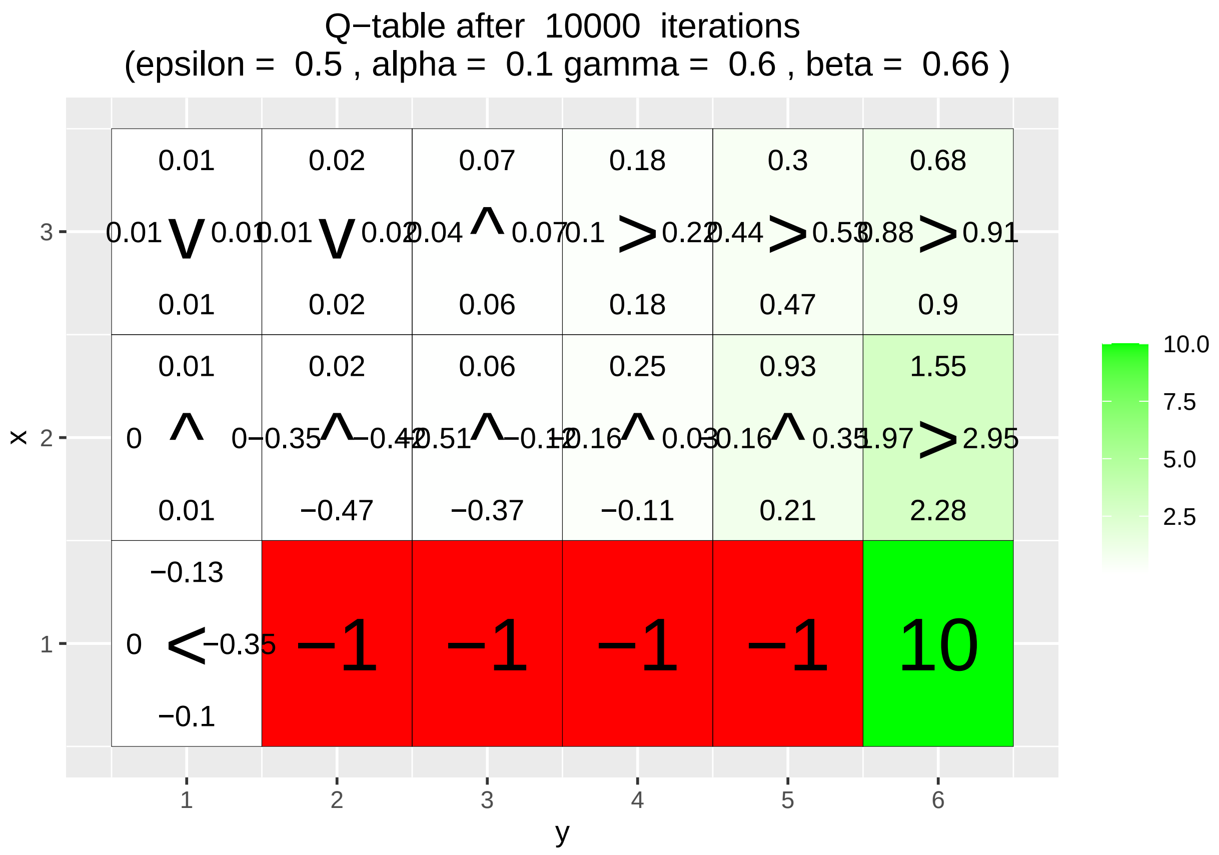


Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0)




Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.2)
0.16
0.15 > 0.23
0.13
0.28
0.17 > 0.45
0.21
0.52
0.42 > 0.84
0.64
0.93
0.59 > 1.56
1.23
1.55
1.18 > 2.8
2.56
2.81
2.06 V 3.49
5.29
0.16
0.11 > 0.2
0.05
0.28
0.08 > 0.37
−0.6
0.49
0.15 > 0.91
−0.61
1.09
0.64 > 1.94
−0.9
1.67
1.25 > 4.38
−0.48
2.98
3.55 V 5.65
9.41
−0.02
0.03 < −0.93
−0.17
−1
−1
−1
−1
10
10.0
7.5
5.0
2.5
x
3
2
1
y
1
2
3
4
5
6


Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.4)
0.05
0.12
0.23
0.57
1.06
1.66
0.04 > 0.05
0.06 > 0.17
0.14 > 0.43
0.41 > 0.99
0.65 > 2.06
2.34 V 2.67
0.04
0.08
0.21
0.62
1.62
3.89
0.04
0.12
0.34
0.97
1.25
2.37
0.02 > 0.05
0.15 ^ −0.09
0.02 ^ 0.04
0.02 ^ 0.34
0.23 > 3.39
3.63 V 5.15
0.02
−0.75
−0.32
−0.29
−0.38
8.19
−0.14
0.02 < −0.47
−0.1
−1
−1
−1
−1
10
x
y
10.0
7.5
5.0
2.5



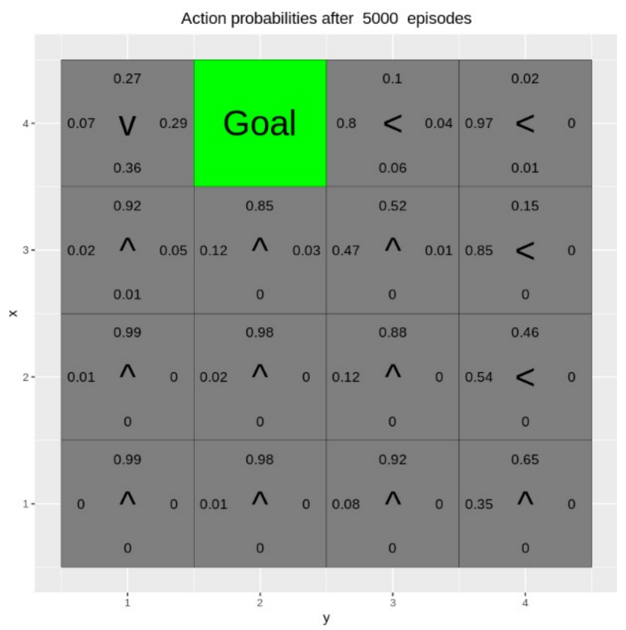
When $\beta = 0$, there is no chance of "slipping" and the learned policy is optimal.

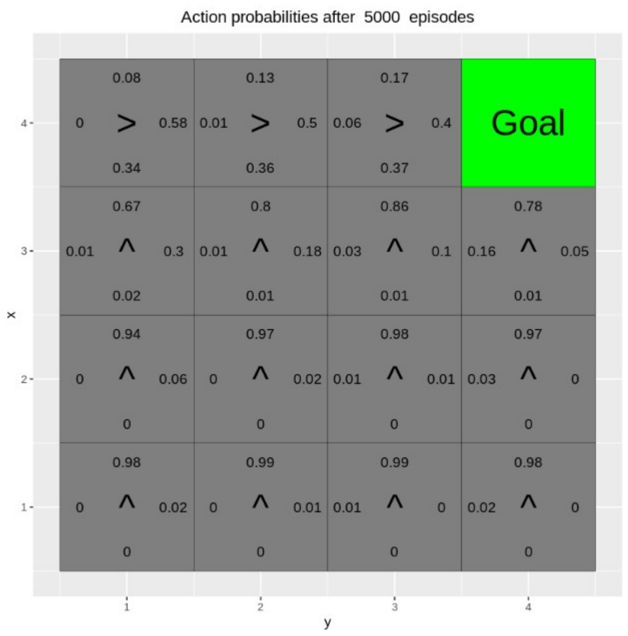
However, as the slipping factor increases, the agent finds it difficult to learn an optimal policy. In the case of $\beta = 0.66$ the agent is not able to learn an optimal policy because 66% of all its actions are sub-optimal.

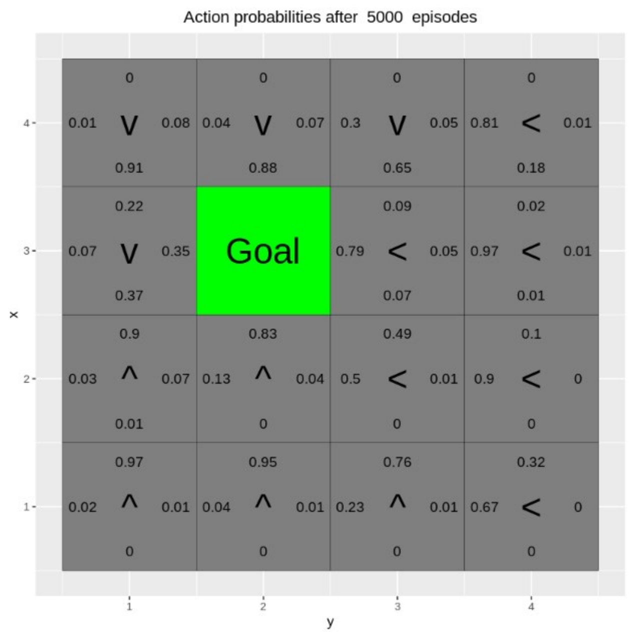
From the environment plot, we can observe that the algorithm assigns low expected returns to actions which are actually optimal and this is due to the agent "slipping" into the negative reward blocks.

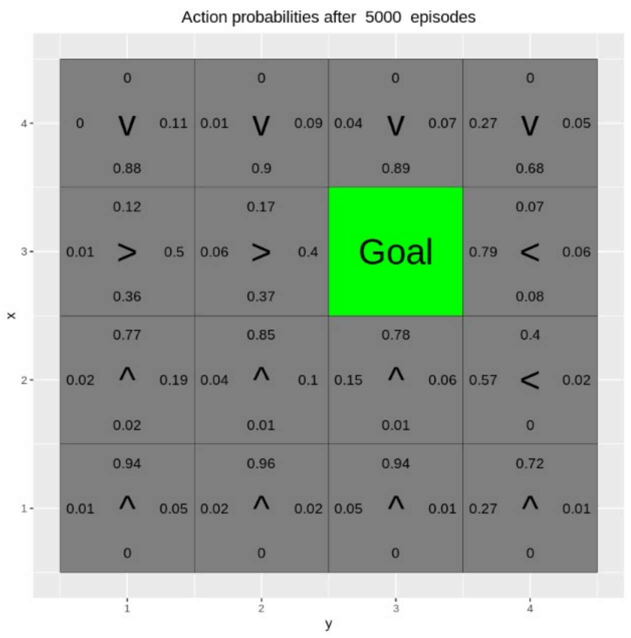
Environment D

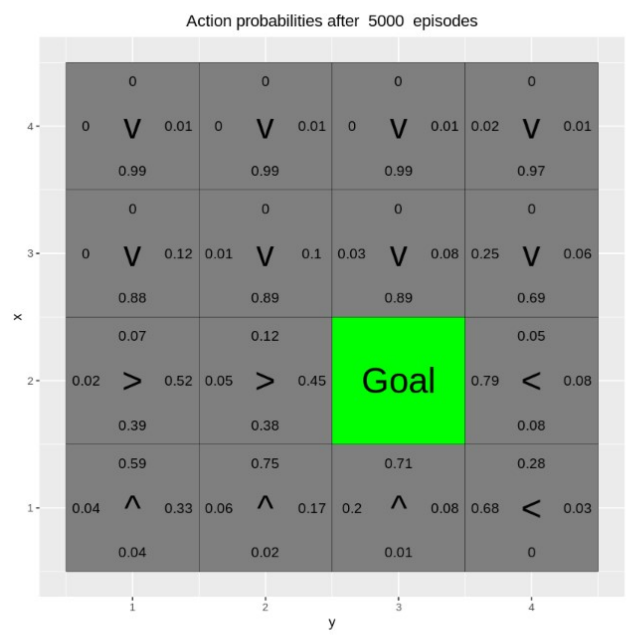
Sub Question 1: Has the agent learned a good policy? Why / Why not ?

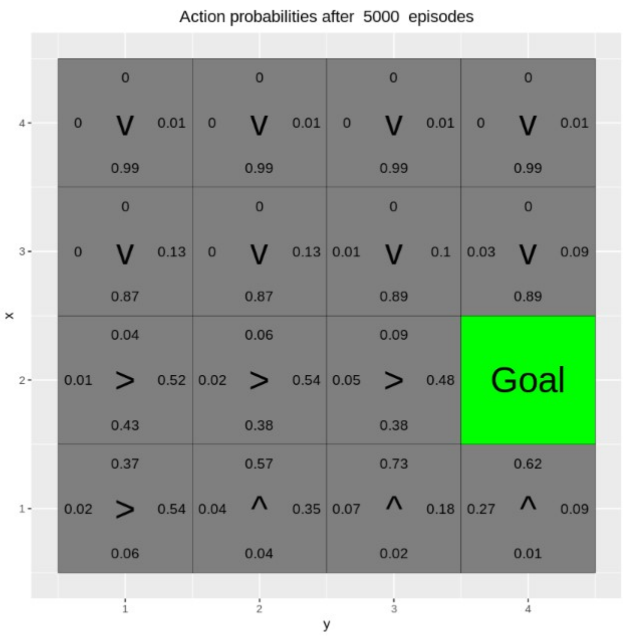


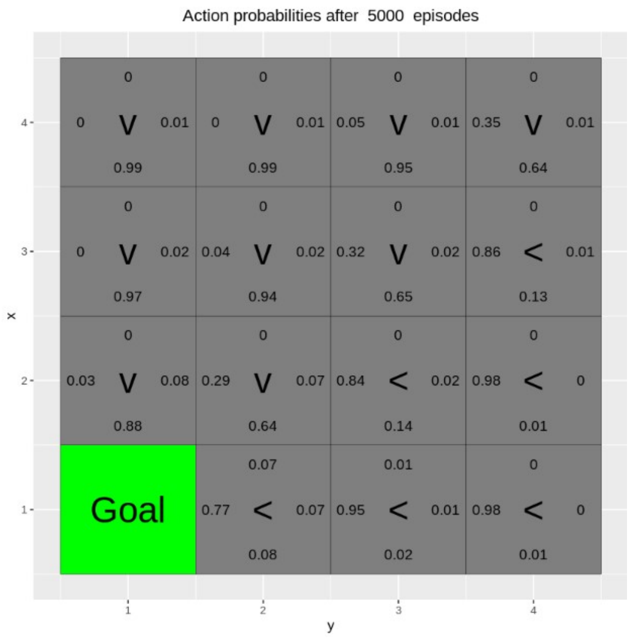


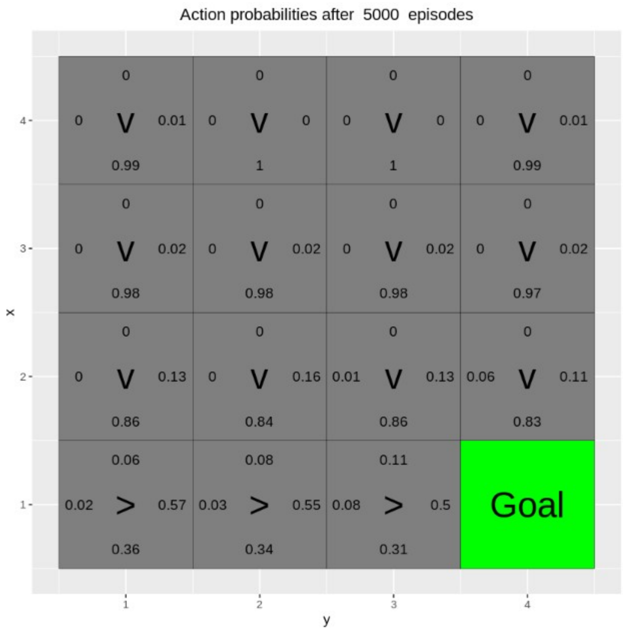












Yes, it has learnt the optimal policy based on the final probability distributions of actions for each state. That is, in almost all cases it takes the path that minimizes the categorical cross-entropy loss by optimizing the weights via Stochastic Gradient Descent.

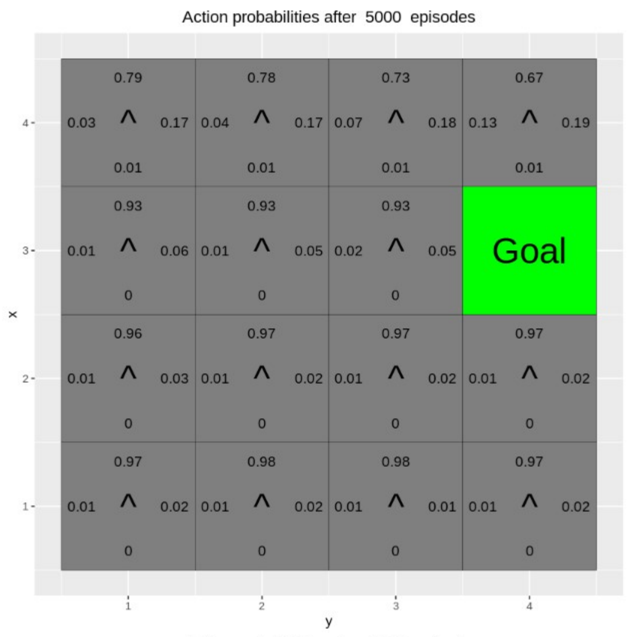
Another point to note is that the training goals span across the environment and the validation goals are in the vicinity of training goals. Because of this, the model is able to generalize well to the validation goals.

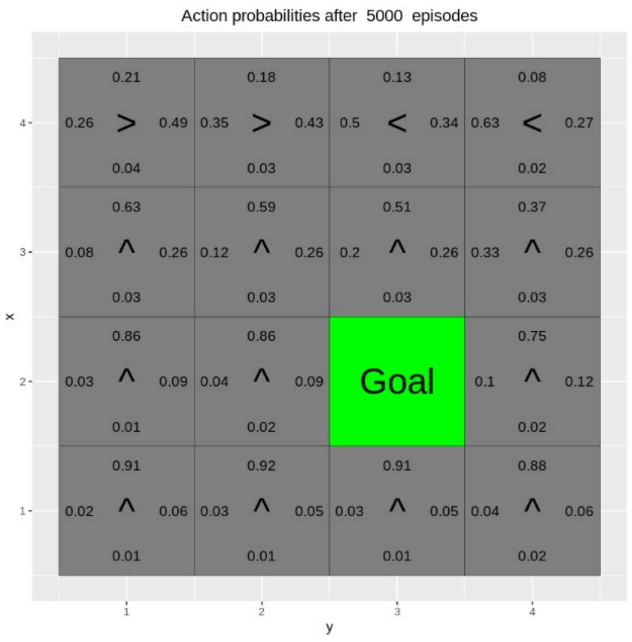
Sub Question 2: Could you have used the Q-learning algorithm to solve this task?

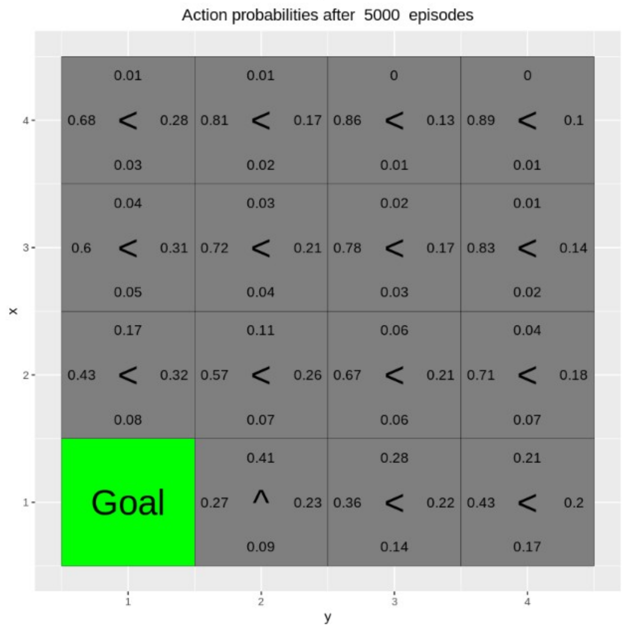
Q-learning(model-free solution) would have resulted in a bad policy learning since it is usually applied to environments where the end goal is fixed. In this case since the start and end goals are not fixed, we would need a model-based solution.

Environment E

Sub Question 1 & 2: Has the agent learned a good policy? Why / Why not ?







In this environment the agent has not learned a good policy because, its training was explicitly based on the top row. It cannot generalize well when the validation goals end up being anywhere other than the top row.

Whereas, as mentioned above, the agent performs/generalizes better in Environment D due to the spread in training goals across the environment.

Some notes on the parameters and their effects:

The value of alpha affects the convergence of the algorithm. When alpha=0.1, an optimal path is typically found. The q-values corresponding to the actions in the path are close to 10, which is the true value as gamma=1. For alpha=0.01, 0.001, an optimal path may still be found but the q-values for the actions in it have not converged.

Some notes on agent behavior during training and testing:

During training Q-learning performs worse because it takes the shortest route to the goal state, which means the agent falling off the cliff now and then due to epsilon. Q-learning prefers the shortest path because it assumes in the updating rule that the subsequent moves will be greedy and, thus, the agent will never fall off the cliff. The reality is that the moves are not greedy due to epsilon. During testing Q-learning performs better because epsilon is zero and thus the agent never falls off the cliff.

```
knitr::opts_chunk$set(echo = TRUE)
#####################################################################################################
# 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)
  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))
  foo <- mapply(function(x,y)
    ifelse(reward_map[x,y] == 0,arrows[GreedyPolicy(x,y)],reward_map[x,y]),df$x,df$y)
  df$val5 <- as.vector(foo)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,max(q_table[x,y,]),
                                     ifelse(reward_map[x,y]<0,NA,reward_map[x,y])),df$x,df$y)
  df$val6 <- as.vector(foo)

  print(ggplot(df,aes(x = y,y = x)) +
```

```r
          scale_fill_gradient(low = "white", high = "green", na.value = "red", name = "") +
          geom_tile(aes(fill=val6)) +
          geom_text(aes(label = val1),size = 4,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 4,nudge_x = .35,na.rm = TRUE) +
          geom_text(aes(label = val3),size = 4,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 4,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 10) +
          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)) +
          scale_x_continuous(breaks = c(1:W),labels = c(1:W)) +
          scale_y_continuous(breaks = c(1:H),labels = c(1:H)))

}

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))
  final_action <- ((action + delta + 3) %% 4) + 1
  foo <- c(x,y) + unlist(action_deltas[final_action])
  foo <- pmax(c(1,1),pmin(foo,c(H,W)))

  return (foo)
}
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}.

  # Your code here.

  # We first retrieve the q-value from the table for given state coordinates
  q_value <- q_table[x,y,]
  # Choose the maximum q-value
  max_qVal <- max(q_value)

  # To handle ties in the scenario
  # We first check which for maximum qvalues
  max_action <- which(q_value == max_qVal)

  # Sample to break tie by choosing an action randomly
  if (length(max_action) > 1) {
    chosen_action <- sample(max_action, 1)
  }else{
    chosen_action <- max_action
  }
  return(chosen_action)
}

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}.

  # Your code here.

  if (runif(1) < epsilon) {
    chosen_action <- sample(x = c(1:4), size = 1)
    return(chosen_action)
  }else{
    return(GreedyPolicy(x,y))
  }
}

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 <<-.

  # Your code here.
  # Initializing state S
  current_state <- start_state
  reward <- 0
  episode_correction <- 0
  repeat{
    # Follow policy, execute action, get reward.
    # determine an action via Epsilon-Greedy method
    action <- EpsilonGreedyPolicy(x = current_state[1],
                                  y = current_state[2], epsilon = epsilon)
    # After taking an action, observing next state and reward
    next_state <- transition_model(x = current_state[1],
                                   y = current_state[2], action = action,
                                   beta = beta)
    next_reward <- reward_map[next_state[1], next_state[2]]
    # Q-table update.
    q_current <- q_table[current_state[1],current_state[2],action]
    max_a_qnext <- max(q_table[next_state[1],next_state[2],])
    tempDiff_err <- next_reward + ((gamma*max_a_qnext) - q_current)
    # Combining terms above for Q-table update:
    q_table[current_state[1],
            current_state[2], action] <<- q_current + (alpha * tempDiff_err)

    # Accumulate the temporal difference correction for this episode
    episode_correction <- episode_correction + tempDiff_err

    # Move to the next state
    current_state <- next_state

    # Update the episode reward
    reward <- reward + next_reward

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

}
q_learning1 <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                        beta = 0, test = 0){

  # Just setting epsilon=0 instead of using the test argument is also OK.
  # But in that case the agent acts greedily while still updating the q-table.

  cur_pos <- start_state
  action <- EpsilonGreedyPolicy(cur_pos[1], cur_pos[2], epsilon*(1-test))
  episode_correction <- 0
  ite <- 0
  repeat{
    # Follow policy, execute action, get reward.
    # action <- EpsilonGreedyPolicy(cur_pos[1], cur_pos[2], epsilon*(1-test))
    new_pos <- transition_model(cur_pos[1], cur_pos[2], action, beta)
    reward <- reward_map[new_pos[1], new_pos[2]]
    new_action <- EpsilonGreedyPolicy(new_pos[1], new_pos[2],epsilon*(1-test))
    # Q-table update.
    old_q <- q_table[cur_pos[1], cur_pos[2], action]
    correction <- ifelse(reward==0,-1,reward) + (gamma*q_table[new_pos[1],
                                                           new_pos[2],
                                                           new_action]) - old_q
    q_table[cur_pos[1], cur_pos[2], action] <<- old_q + alpha*correction*(1-test)

    cur_pos <- new_pos
    episode_correction <- episode_correction + correction*(1-test)
    action <- new_action

    if(reward!=0)
      # End episode.
      return (c(reward-ite,episode_correction))
    else
      ite <- ite+1
  }

}

q_learning2 <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                  beta = 0){

  cur_pos <- start_state
  cur_action <- EpsilonGreedyPolicy(cur_pos[1], cur_pos[2], epsilon)

  new_pos <- transition_model(cur_pos[1], cur_pos[2], cur_action, beta)
  new_action <- EpsilonGreedyPolicy(new_pos[1], new_pos[2], epsilon)
  reward <- reward_map[new_pos[1], new_pos[2]]
  episode_correction <- 0
  repeat{
    if (reward != 0) {
      old_q <- q_table[cur_pos[1], cur_pos[2], cur_action]
      q_table[cur_pos[1],
              cur_pos[2],
              cur_action] <<- old_q + alpha*(reward - old_q)
      break
    }else{
      new_pos_2step <- transition_model(new_pos[1], new_pos[2], new_action, beta)
      reward_2step <- reward_map[new_pos_2step[1], new_pos_2step[2]]
      new_action_2step <- EpsilonGreedyPolicy(new_pos_2step[1], new_pos_2step[2], epsilon)

      if (reward_2step != 0) {
        old_q <- q_table[cur_pos[1], cur_pos[2], cur_action]
        q_table[cur_pos[1],
                cur_pos[2],
                cur_action] <<- old_q + alpha*(-1 + (gamma*reward_2step) - old_q)

        old_q <- q_table[new_pos[1], new_pos[2], new_action]
        q_table[new_pos[1],
                new_pos[2],
                new_action] <<- old_q + alpha*(reward_2step - old_q)
        break
      }else{
        old_q <- q_table[cur_pos[1], cur_pos[2], cur_action]
        q_table[cur_pos[1],
                cur_pos[2],
                cur_action] <<- old_q + alpha*(-1 + (gamma*-1) +
                                                ((gamma^2)*
                                                   (q_table[new_pos_2step[1],
                                                            new_pos_2step[2],
                                                            new_action_2step])) - old_q)

        cur_pos <- new_pos
        cur_action <- new_action

        new_pos <- new_pos_2step
        new_action <- new_action_2step

        reward <- reward_2step
      }
    }
  }

}

# Code for Value Iteration:
value_iteration <- function(epsilon = 0.0001, gamma = 0.95) {
  # Initialize the value function arbitrarily (V0)
  V <- matrix(0, nrow = H, ncol = W)  # H and W are the height and width of the grid world

  while (TRUE) {
    delta <- 0  # Initialize the change in value function

    for (x in 1:H) {
      for (y in 1:W) {
        if (is_terminal(x, y)) {
          # Skip terminal states
          next
        }

        # Calculate the new value using the Bellman equation
        max_action_value <- -Inf
        for (action in 1:4) {  # Assuming there are four possible actions
          s_prime <- transition_model(x, y, action, beta)
          new_x <- s_prime[1]
          new_y <- s_prime[2]
          current_reward <- reward_map[new_x, new_y]
          new_value <- current_reward + gamma * V[new_x, new_y]
          max_action_value <- max(max_action_value, new_value)
        }

        # Update the change in value function
        delta <- max(delta, abs(max_action_value - V[x, y]))

        # Update the value function
        V[x, y] <- max_action_value
      }
    }

    # Check for convergence
    if (delta < epsilon) {
      break
    }
  }

  # Now, V contains the optimal value function V*.

  # To extract the optimal policy, you can use the value function:
  policy <- matrix(0, nrow = H, ncol = W)  # Initialize the policy
  for (x in 1:H) {
    for (y in 1:W) {
      if (is_terminal(x, y)) {
        # Skip terminal states
        next
      }

      # Find the action that maximizes the expected value
      max_action_value <- -Inf
      max_action <- 0
      for (action in 1:4) {  # Assuming there are four possible actions
        s_prime <- transition_model(x, y, action, beta)
        new_x <- s_prime[1]
        new_y <- s_prime[2]
        current_reward <- reward_map[new_x, new_y]
        new_value <- current_reward + gamma * V[new_x, new_y]
        if (new_value > max_action_value) {
          max_action_value <- new_value
          max_action <- action
        }
      }

      # Set the optimal action in the policy
      policy[x, y] <- max_action
    }
  }

  return(list(ValueFunction = V, Policy = policy))
}

policy_iteration <- function(gamma = 0.95) {
  # Initialize the policy arbitrarily
```

```r
policy <- matrix(sample(1:4, H * W, replace = TRUE), nrow = H, ncol = W)

while (TRUE) {
  # Policy Evaluation: Evaluate the current policy
  V <- matrix(0, nrow = H, ncol = W)  # Initialize the value function arbitrarily

  while (TRUE) {
    delta <- 0  # Initialize the change in value function

    for (x in 1:H) {
      for (y in 1:W) {
        if (is_terminal(x, y)) {
          # Skip terminal states
          next
        }

        action <- policy[x, y]
        s_prime <- transition_model(x, y, action, beta)
        new_x <- s_prime[1]
        new_y <- s_prime[2]
        current_reward <- reward_map[new_x, new_y]
        new_value <- current_reward + gamma * V[new_x, new_y]

        delta <- max(delta, abs(new_value - V[x, y]))
        V[x, y] <- new_value
      }
    }

    # Check for convergence
    if (delta < epsilon) {
      break
    }
  }

  # Policy Improvement: Improve the policy using the current value function
  policy_stable <- TRUE

  for (x in 1:H) {
    for (y in 1:W) {
      if (is_terminal(x, y)) {
        # Skip terminal states
        next
      }

      old_action <- policy[x, y]
      max_action_value <- -Inf
      best_action <- 0

      for (action in 1:4) {  # Assuming there are four possible actions
        s_prime <- transition_model(x, y, action, beta)
        new_x <- s_prime[1]
        new_y <- s_prime[2]
        current_reward <- reward_map[new_x, new_y]
```

```r
          new_value <- current_reward + gamma * V[new_x, new_y]

          if (new_value > max_action_value) {
            max_action_value <- new_value
            best_action <- action
          }
        }

        # Update the policy if a better action is found
        if (best_action != old_action) {
          policy_stable <- FALSE
          policy[x, y] <- best_action
        }
      }
    }

    # If the policy is stable (no change), we have found the optimal policy
    if (policy_stable) {
      break
    }
  }

  return(list(Policy = policy))
}
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))

vis_environment()

for(i in 1:10000){
  foo <- q_learning(start_state = c(3,1))

  if(any(i==c(10,100,1000,10000)))
    vis_environment(i)
}


# Environment B (the effect of epsilon and gamma)

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
```

```r
q_table <- array(0,dim = c(H,W,4))

vis_environment()

MovingAverage <- function(x, n){

  cx <- c(0,cumsum(x))
  rsum <- (cx[(n+1):length(cx)] - cx[1:(length(cx) - n)]) / n

  return (rsum)
}

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)
  gridExtra::grid.arrange(plot(MovingAverage(reward,100),type = "l"),
  plot(MovingAverage(correction,100),type = "l"), nrow = 2)

}

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)
  gridExtra::grid.arrange(plot(MovingAverage(reward,100),type = "l"),
  plot(MovingAverage(correction,100),type = "l"), nrow = 2)
}
# Environment C (the effect of beta).

H <- 3
W <- 6

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,2:5] <- -1
reward_map[1,6] <- 10
```

```r
q_table <- array(0,dim = c(H,W,4))

vis_environment()

for(j in c(0,0.2,0.4,0.66)){
  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)
}
```