Lab3 Individual Report

aswma317

2023-10-18

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

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

```
## Warning: package 'ggplot2' was built under R version 4.2.3
```

```
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)) +
          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}.
```

```
  #Get the max q-value from the q_table for the state (x,y)
  max_q = max(q_table[x,y,])

  #Get the indexes which has maximum values
  max_index = which(q_table[x,y,] == max_q)#There can be more than 1 index

  #Question: can the agent stay in the same location?

  #In case of ties, sample randomly from those ties
  if (length(max_index) > 1) {
    action <- sample(max_index, size = 1)
  }else{
    action <- max_index
  }
  return(action)
}

# greedy_action <- GreedyPolicy(1,1)
# greedy_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}.

  #Explore with epsilon probability or exploit
  explore <- ifelse(epsilon > runif(1), 1, 0)

  if(explore){
    action <- sample(1:4, size = 1)#Exploration
  }else{
    action <- GreedyPolicy(x,y)#Exploitation
  }

  return(action)
}

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
  #Question: why is action_deltas(y,x)
  foo <- c(x,y) + unlist(action_deltas[final_action])
  foo <- pmax(c(1,1),pmin(foo,c(H,W)))

  return (foo)
}
#alpha(Learning Rate)
#0:emphasizes on already learned experience. May find the path, but Q table
#may not converge.
#1:overwrites the previous experience with new information. Q table will converge.

#gamma(Discount factor)
#0:Maximizes short term reward
#1:Maximizes long term reward

#epsilon(Exploration factor)
#0:Exploitation
#1:Exploration

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 <<-.

  #RL_L1_QLearning, slide 14

  #Initialize S
  old_x <- start_state[1]
  old_y <- start_state[2]
```

```
  episode_correction <- 0
  reward <- 0
  repeat{
    #print(paste0('State - (', old_x, ',', old_y,')'))
    #Choose A using policy
    action <- EpsilonGreedyPolicy(old_x,old_y,epsilon)

    #Take A and get the next state(S')
    s_prime <- transition_model(old_x, old_y, action, beta)
    new_x <- s_prime[1]
    new_y <- s_prime[2]

    #Observe the reward for moving into new state
    current_reward <- reward_map[new_x, new_y]

    #Calculate the Q-value temporal difference
    old_q <- q_table[old_x, old_y, action]
    new_q <- max(q_table[new_x, new_y, ])
    temporal_diff <- current_reward + (gamma * new_q) - old_q

    #Update Q-table
    q_table[old_x, old_y, action] <<- old_q + alpha * temporal_diff

    #Update S <- S'
    old_x <- new_x
    old_y <- new_y

    #Compute the episode correction - sum of temporal difference
    #Question: is it absolute or w/o
    episode_correction <- episode_correction + temporal_diff

    #Update the reward for the episode
    reward <- reward + current_reward

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

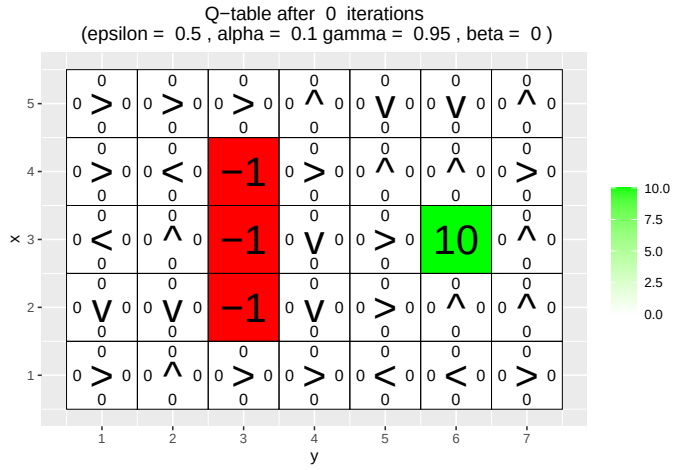
Environment A

Environment A (learning)

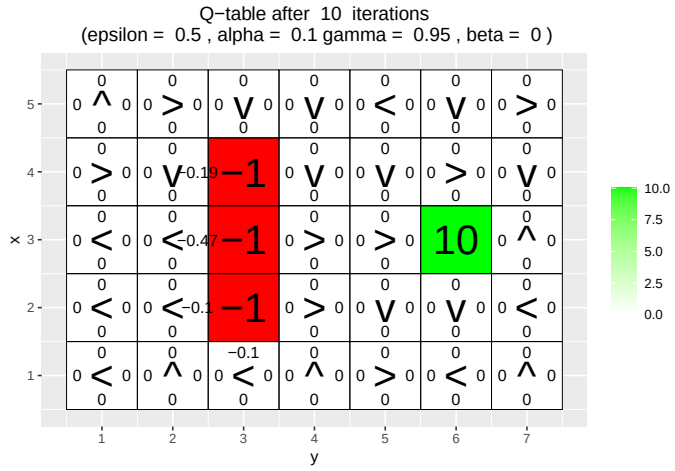
Sub Question 1 : What has the agent learned after the first 10 episodes ? We see below that after 10 iterations, the agent has only learned to avoid the states with a reward of -1 by assigning negative discounted expected reward if an action is leading to that state.

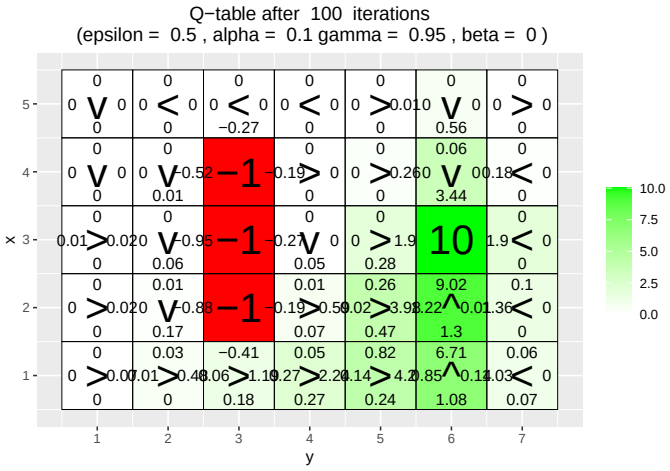
```
H <- 5
W <- 7

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[3,6] <- 10
```

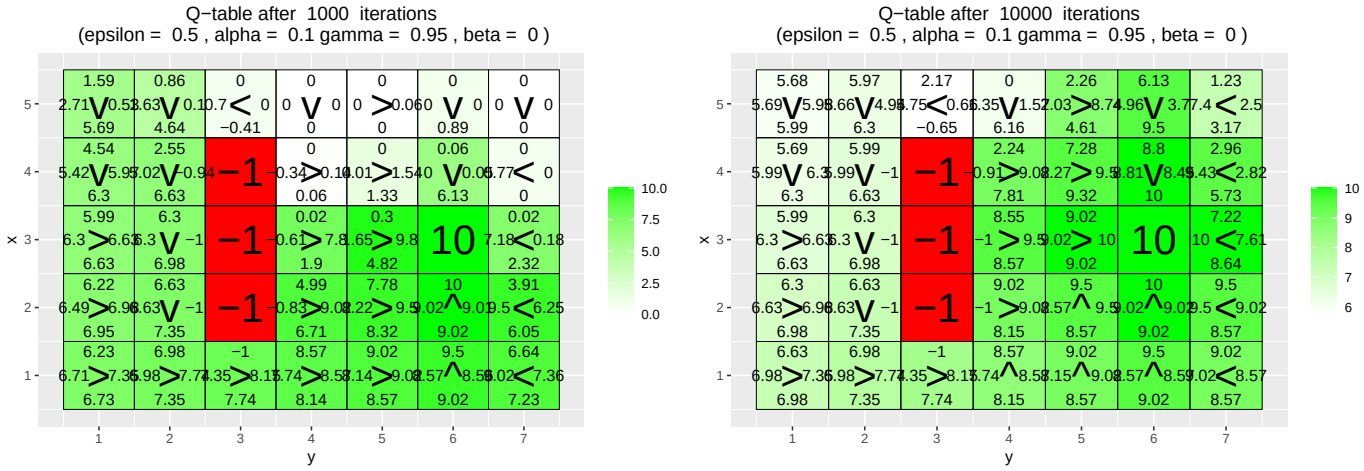
```
reward_map[2:4,3] <- -1

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)
}
```







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.

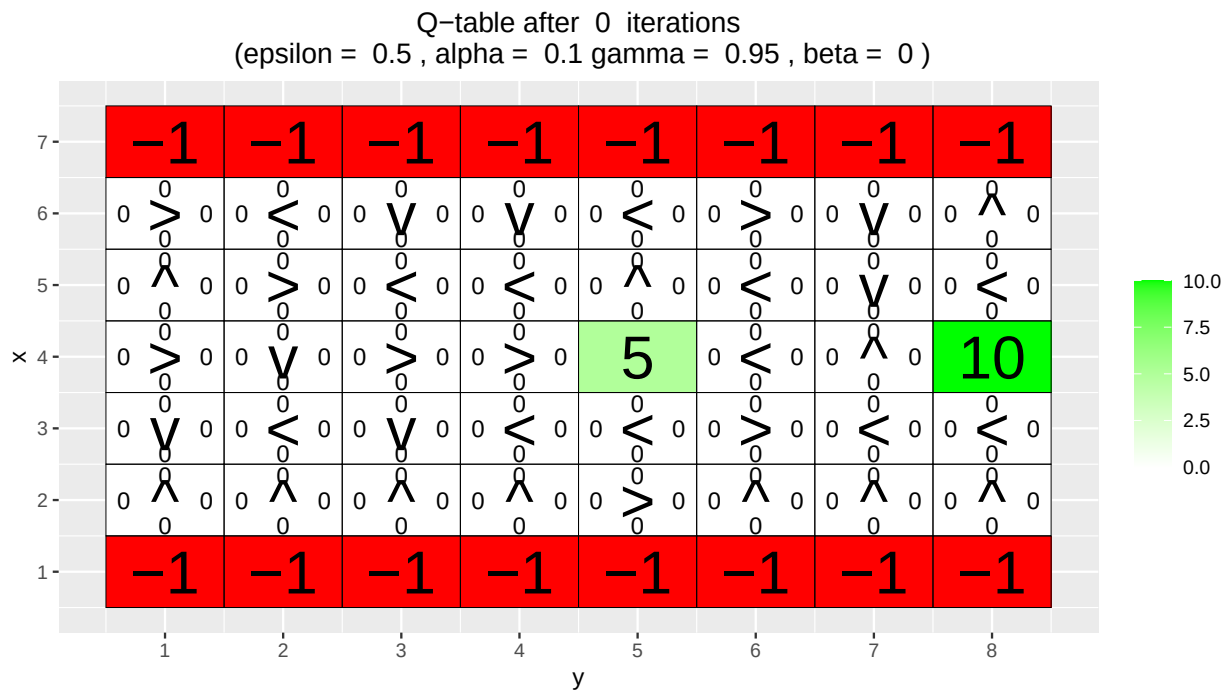
If we do want multiple paths, above and below the negative reward block, we would have to make the agent to be exploratory i.e., by setting a higher ϵ.

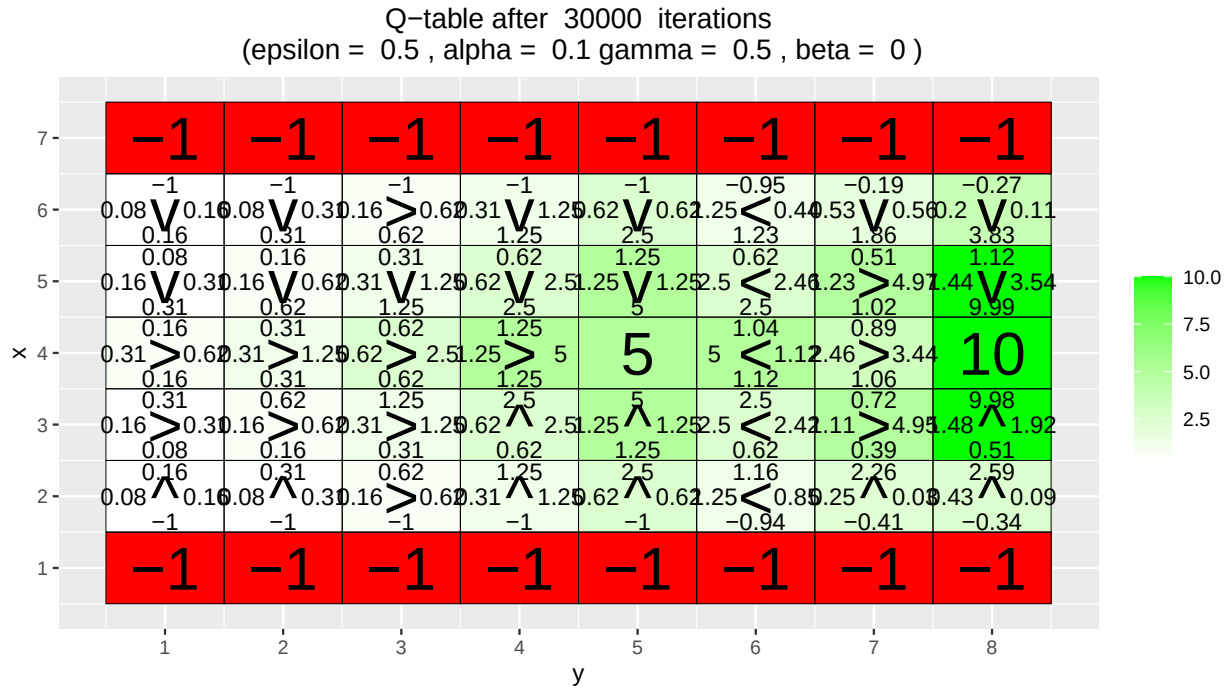
Environment B

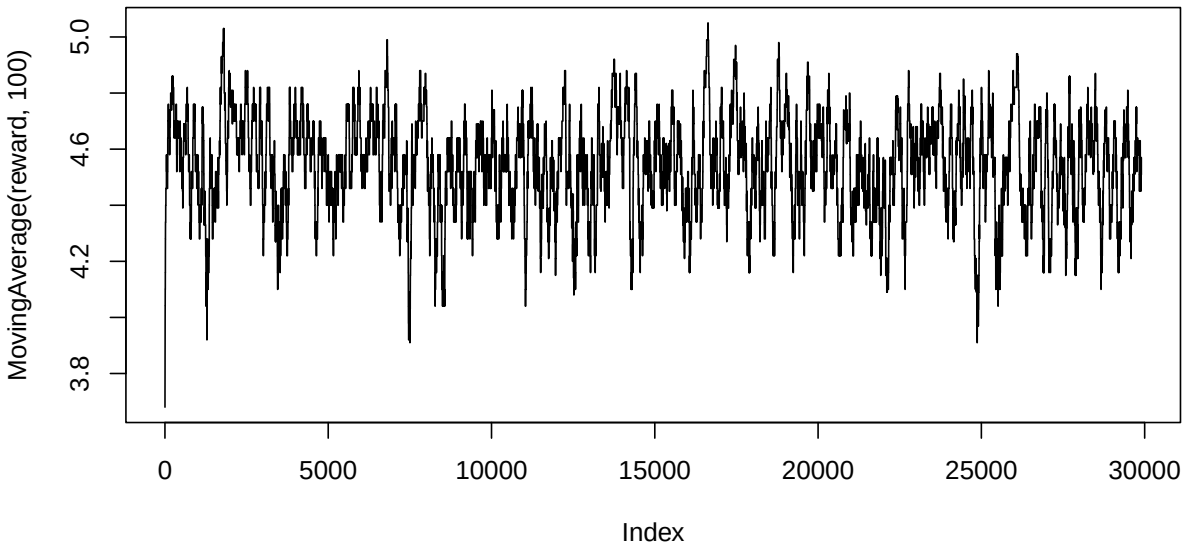
For $\epsilon = 0.5$

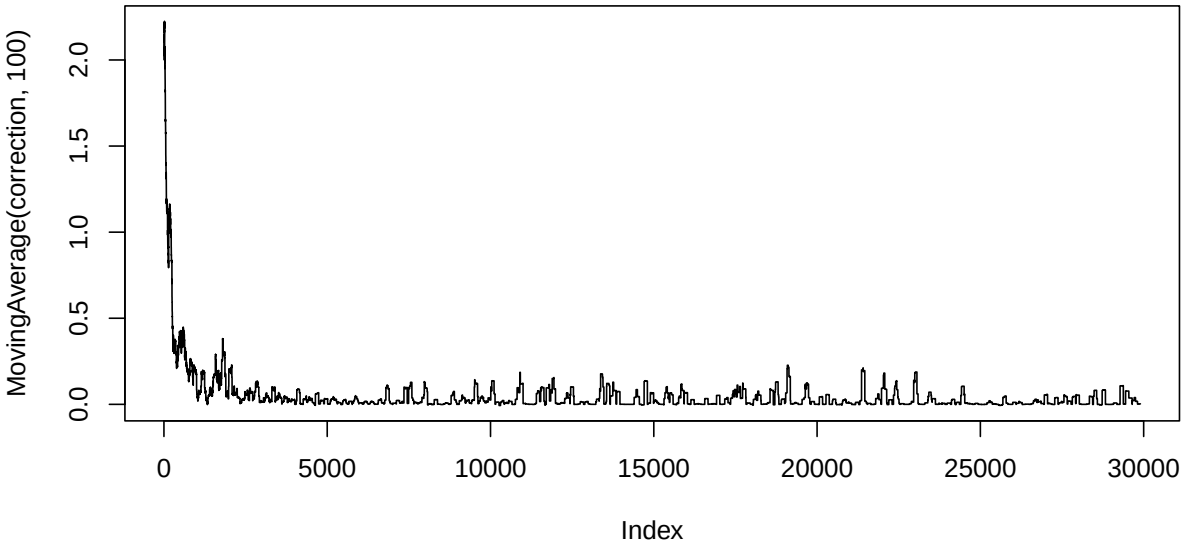
```
# Environment B (the effect of epsilon and gamma)

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

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

vis_environment()
```

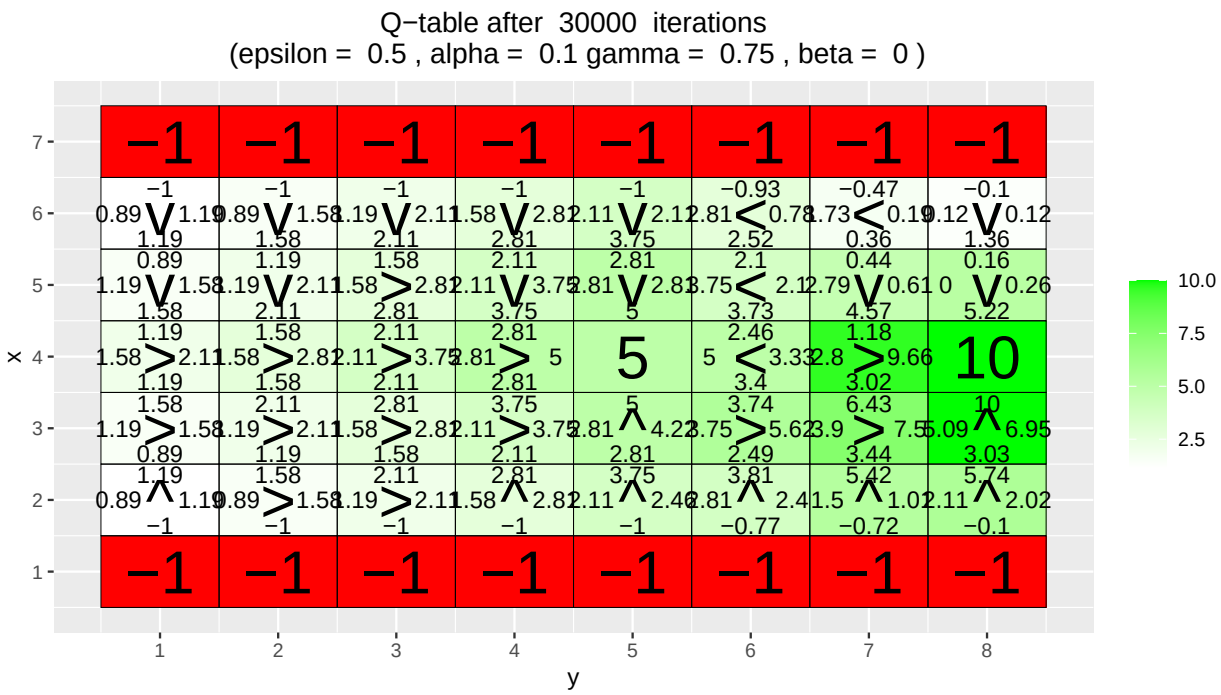
```
MovingAverage <- function(x, n){

  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)

}
```

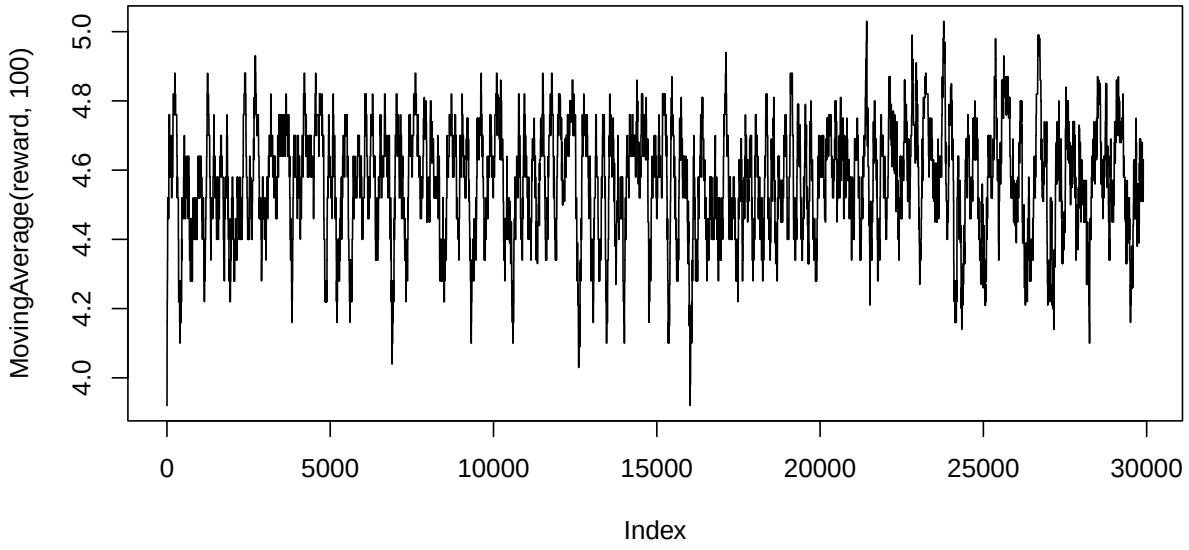


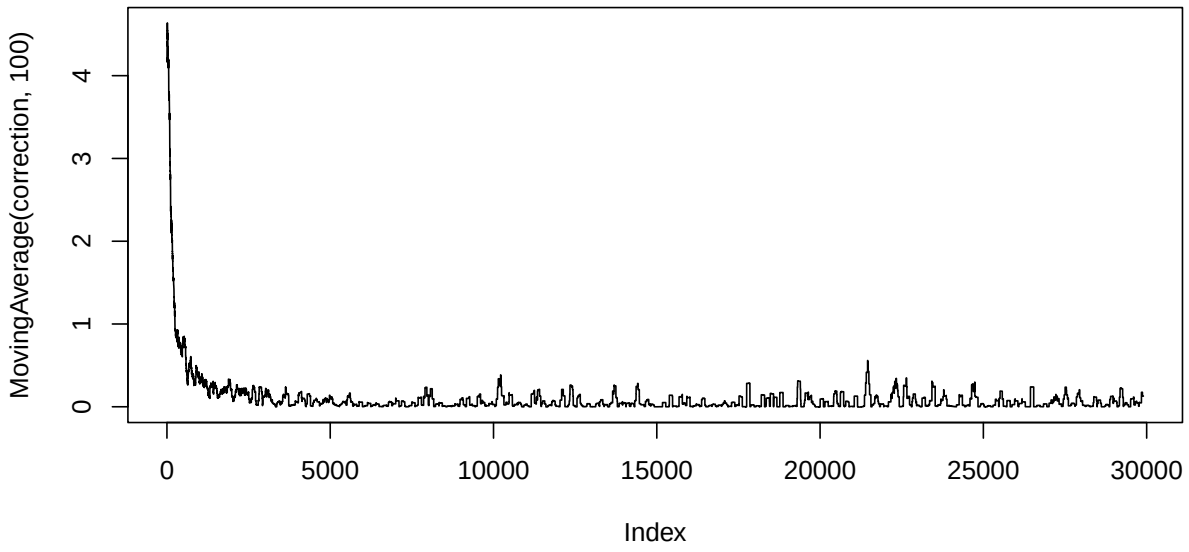

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.5 , beta = 0)
x
y
10.0
7.5
5.0
2.5

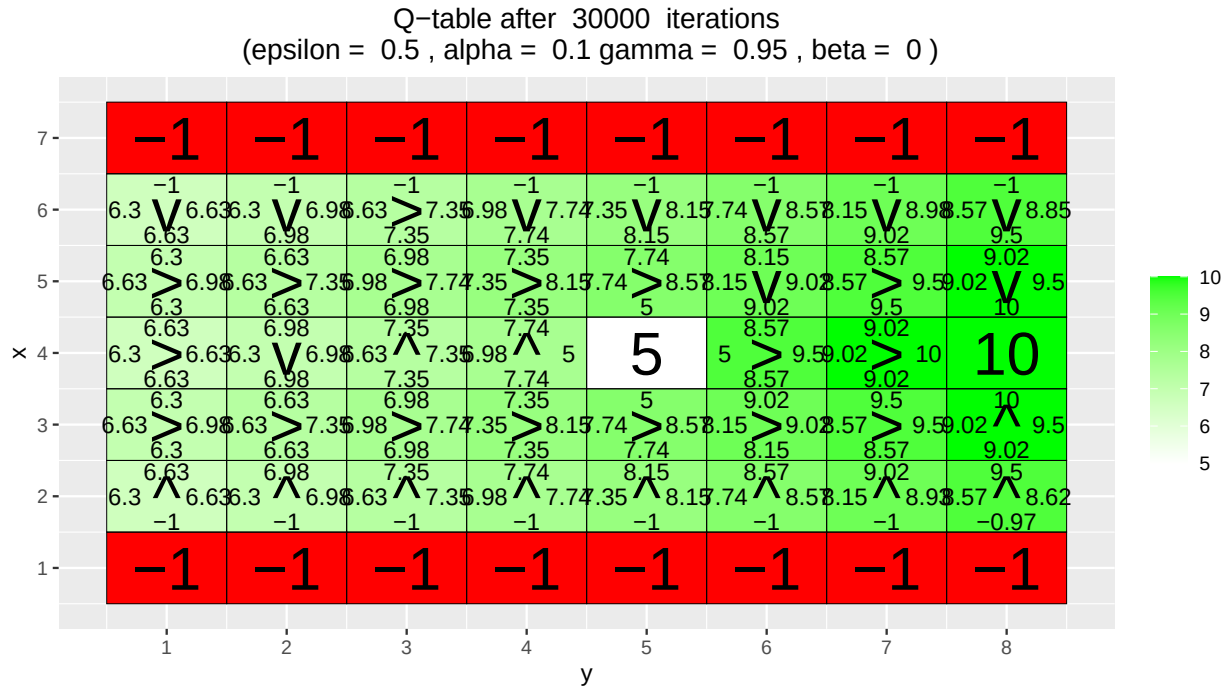

MovingAverage(reward, 100)
Index

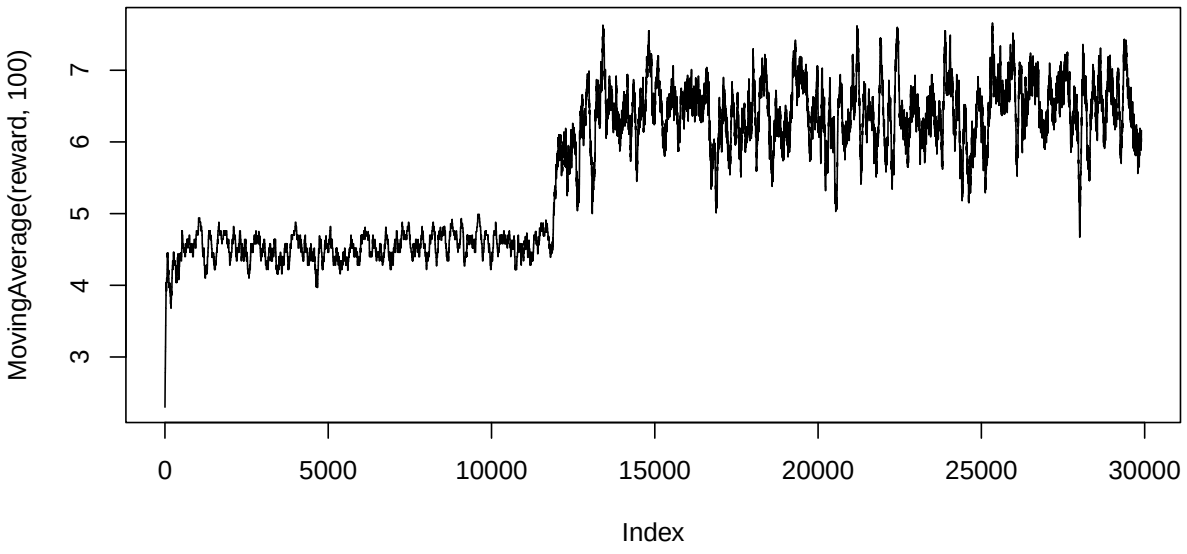

MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
2.0
0
5000
10000
15000
20000
25000
30000
Index

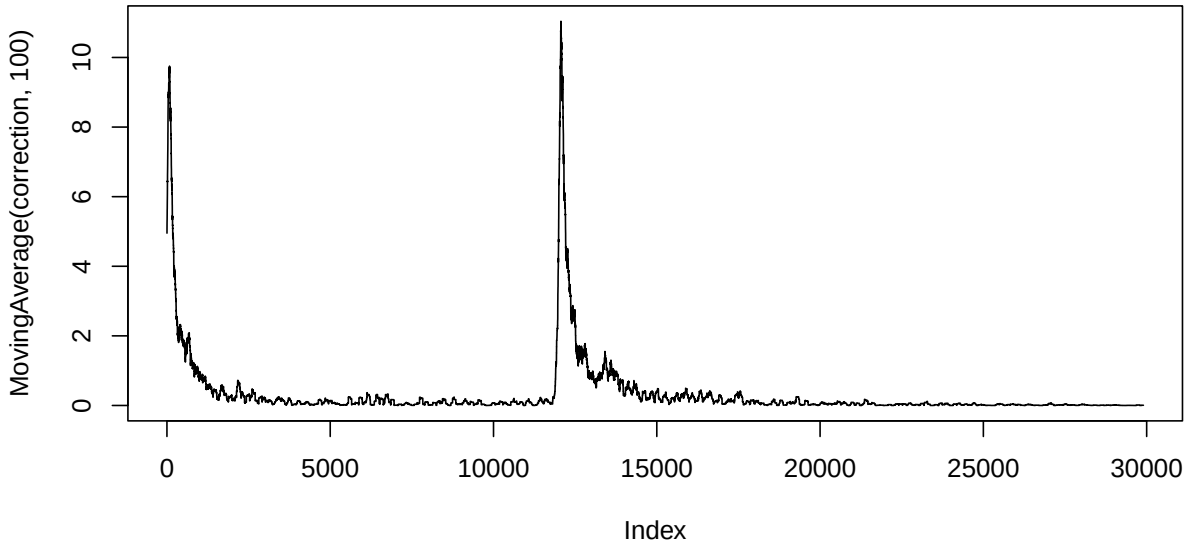

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5


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


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


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


MovingAverage(reward, 100)
Index
0
5000
10000
15000
20000
25000
30000



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.

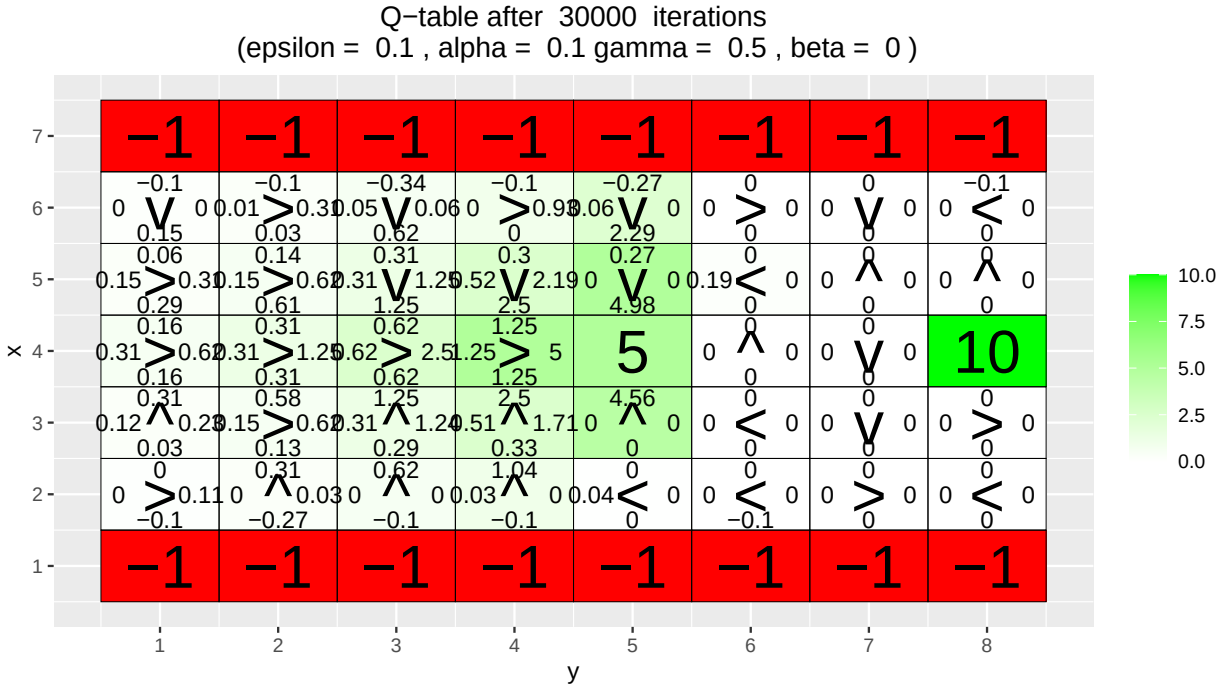
When $\gamma = 0.75$ the behavior of the agent with respect to moving average of rewards and corrections are similar to the previous case. However, now the agent assigns higher q-values to actions leading to reward block 10 in the vicinity of reward block 5. This signifies that the agent is now starting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

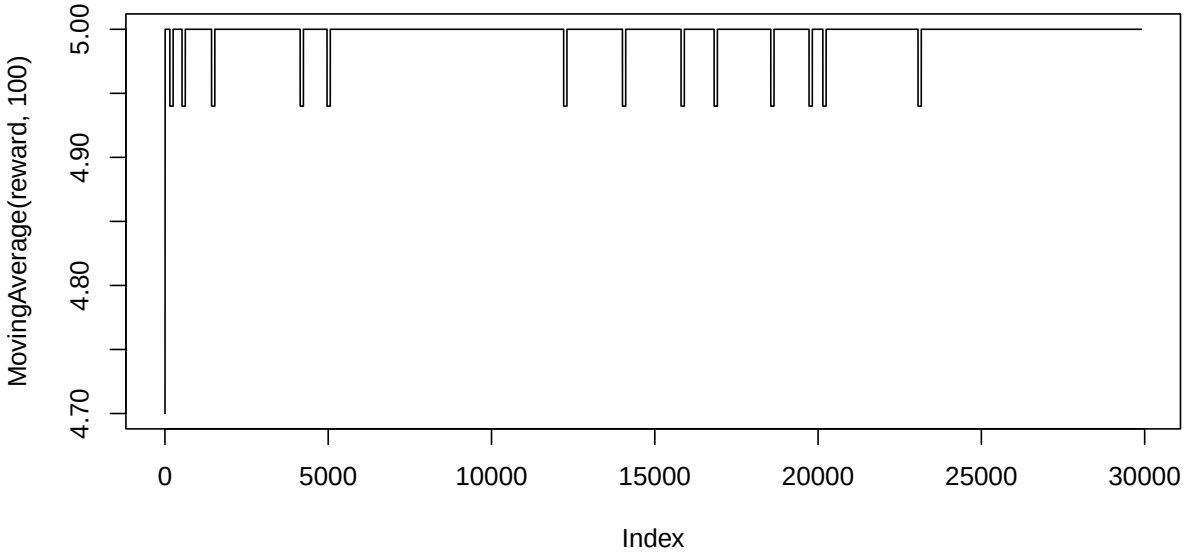
When $\gamma = 0.95$, we can see from the moving average plots for rewards and corrections, that after 2500 episodes the agent starts to acknowledge the terminal state of reward 10. Since, the discount factor is higher the agent does not prioritize immediate rewards i.e., reward block 5 and hence always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

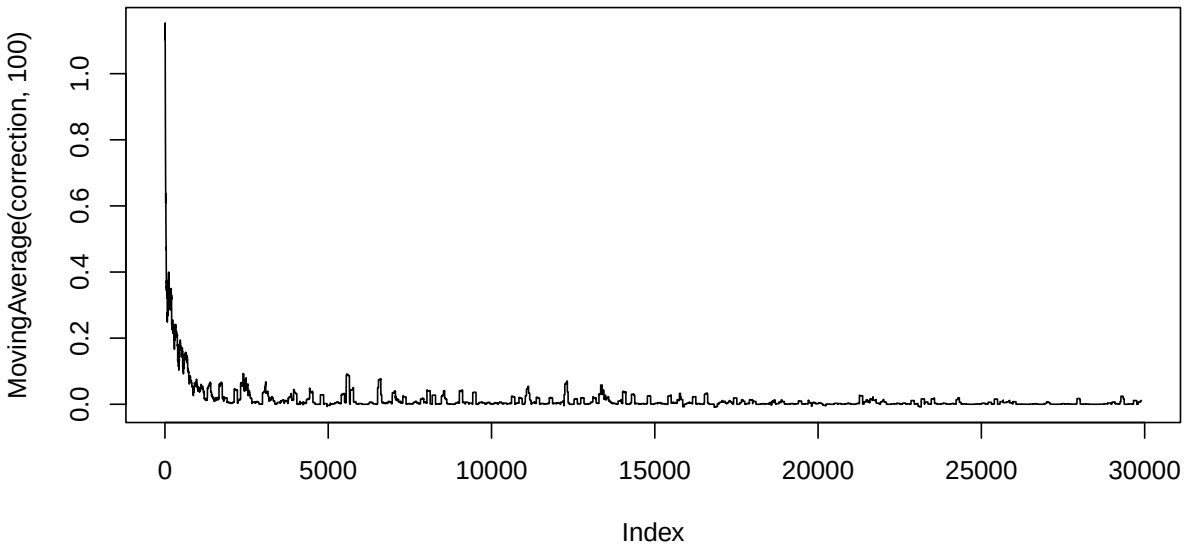
For $\epsilon = 0.1$

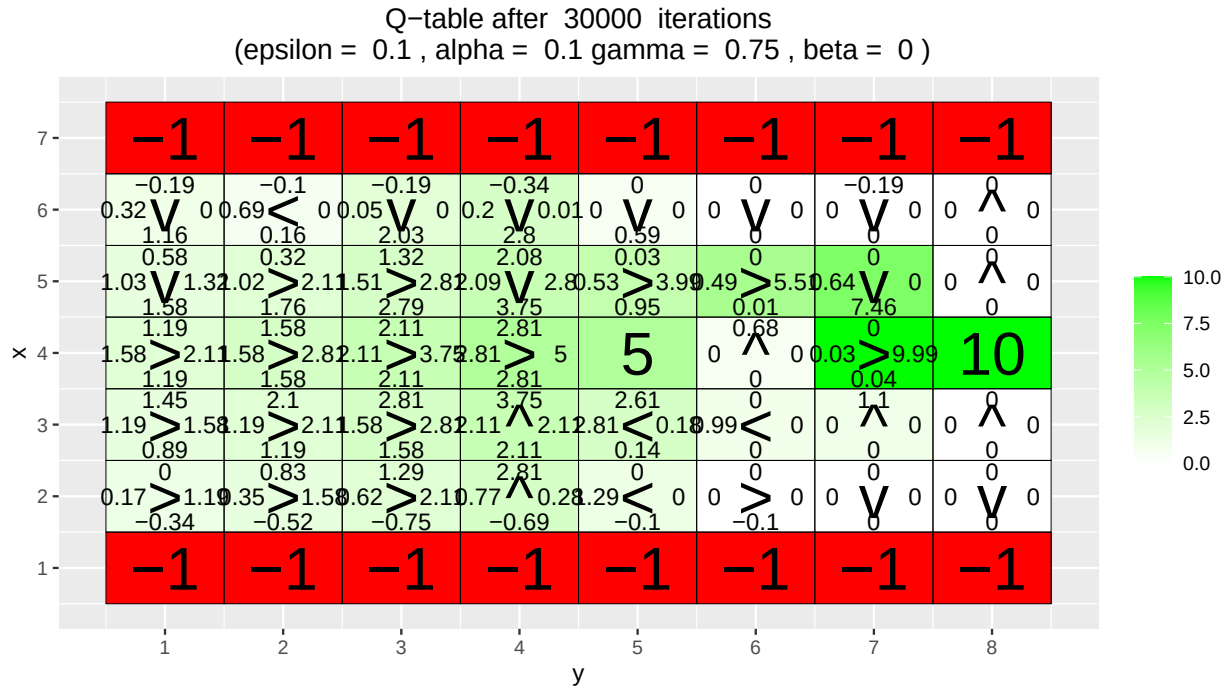
```
for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL
```

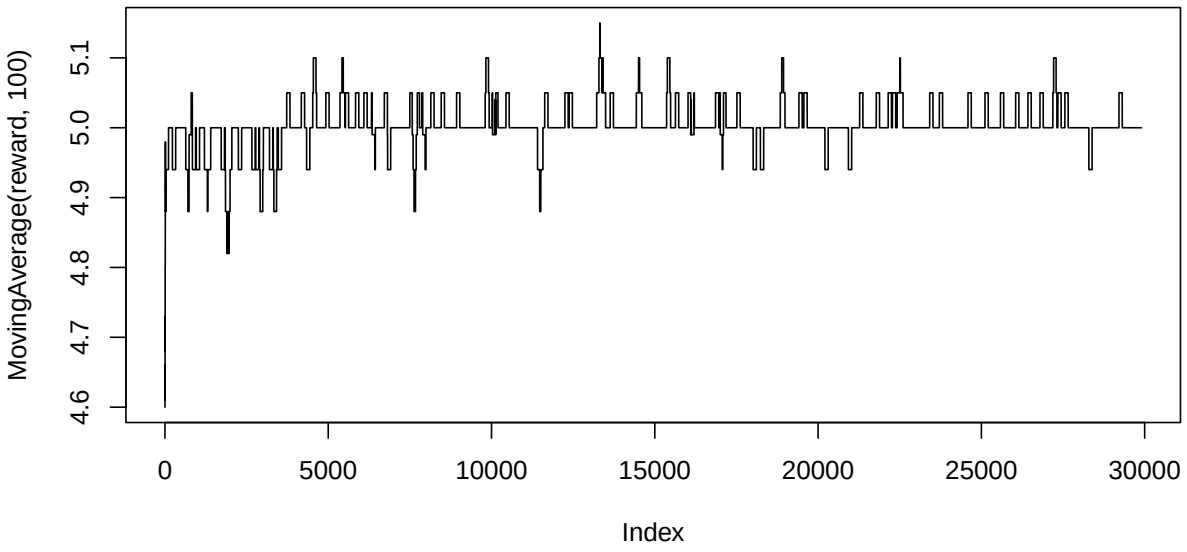
```
  for(i in 1:30000){
    foo <- q_learning(epsilon = 0.1, gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }

  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)
}
```

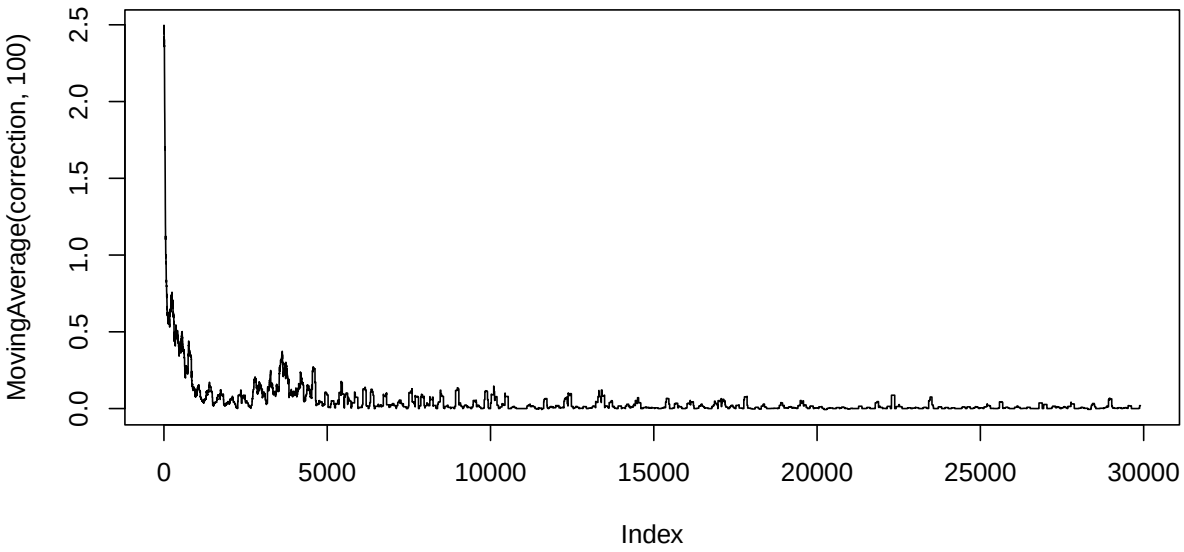


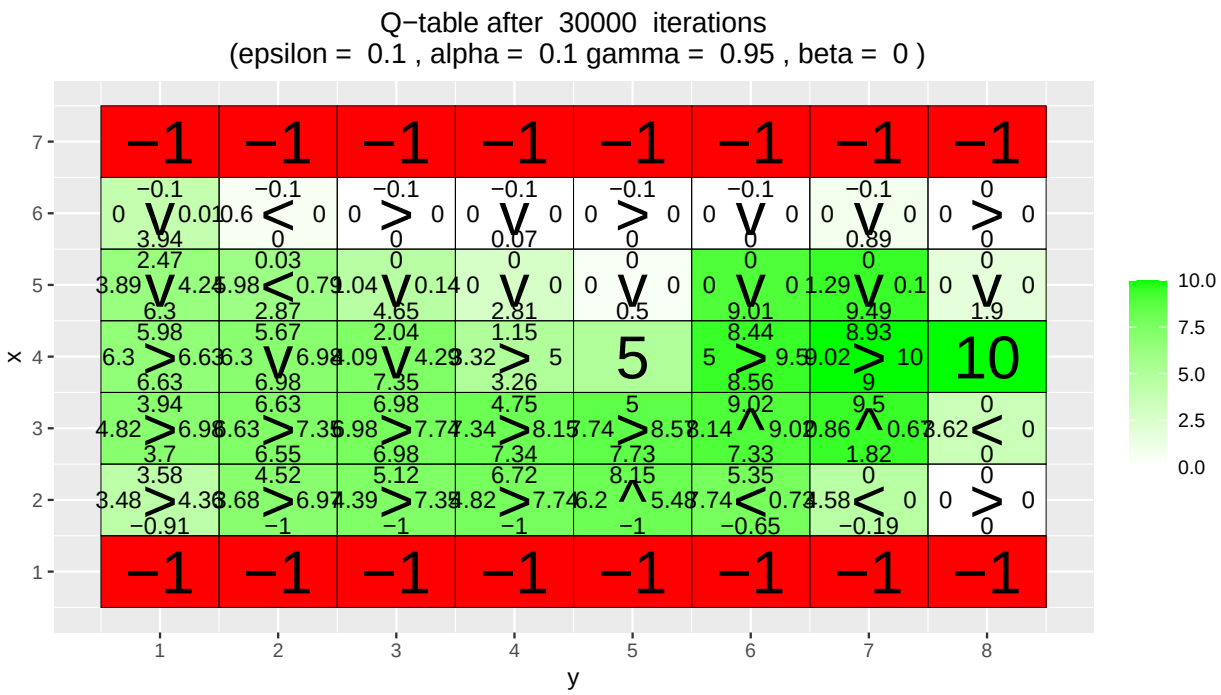

MovingAverage(reward, 100)
5.00
4.90
4.80
4.70
0
5000
10000
15000
20000
25000
30000
Index

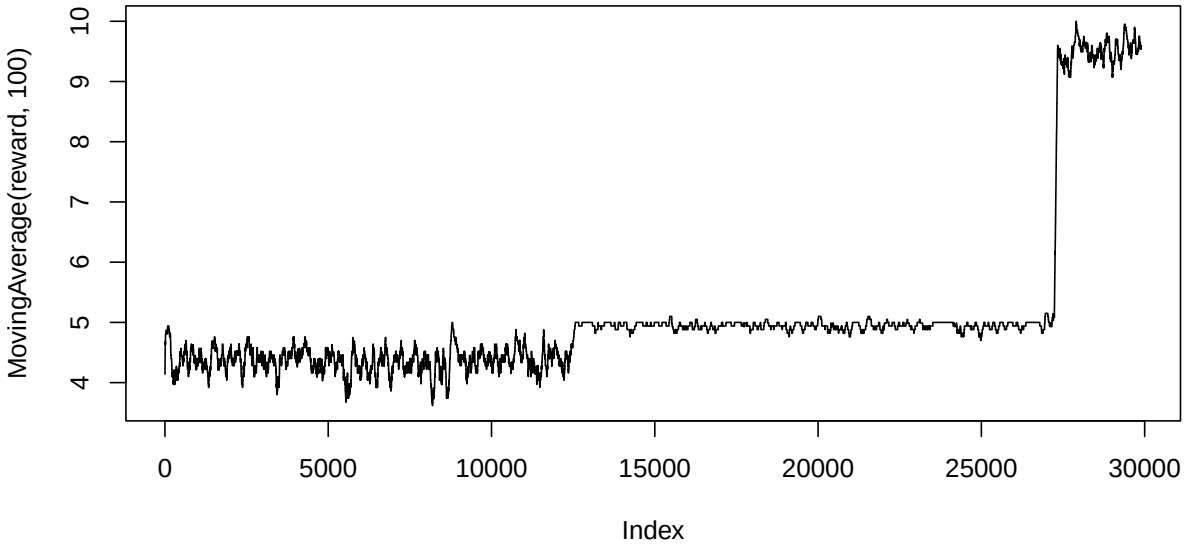

MovingAverage(correction, 100)
1.0
0.8
0.6
0.4
0.2
0.0
0
5000
10000
15000
20000
25000
30000
Index

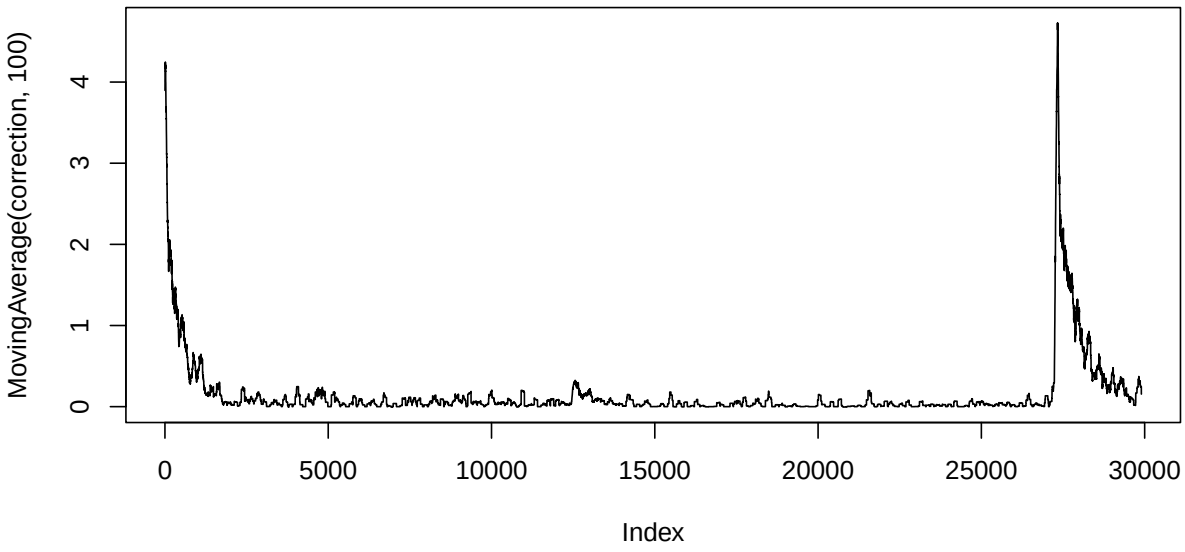

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
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 = 0.1 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0



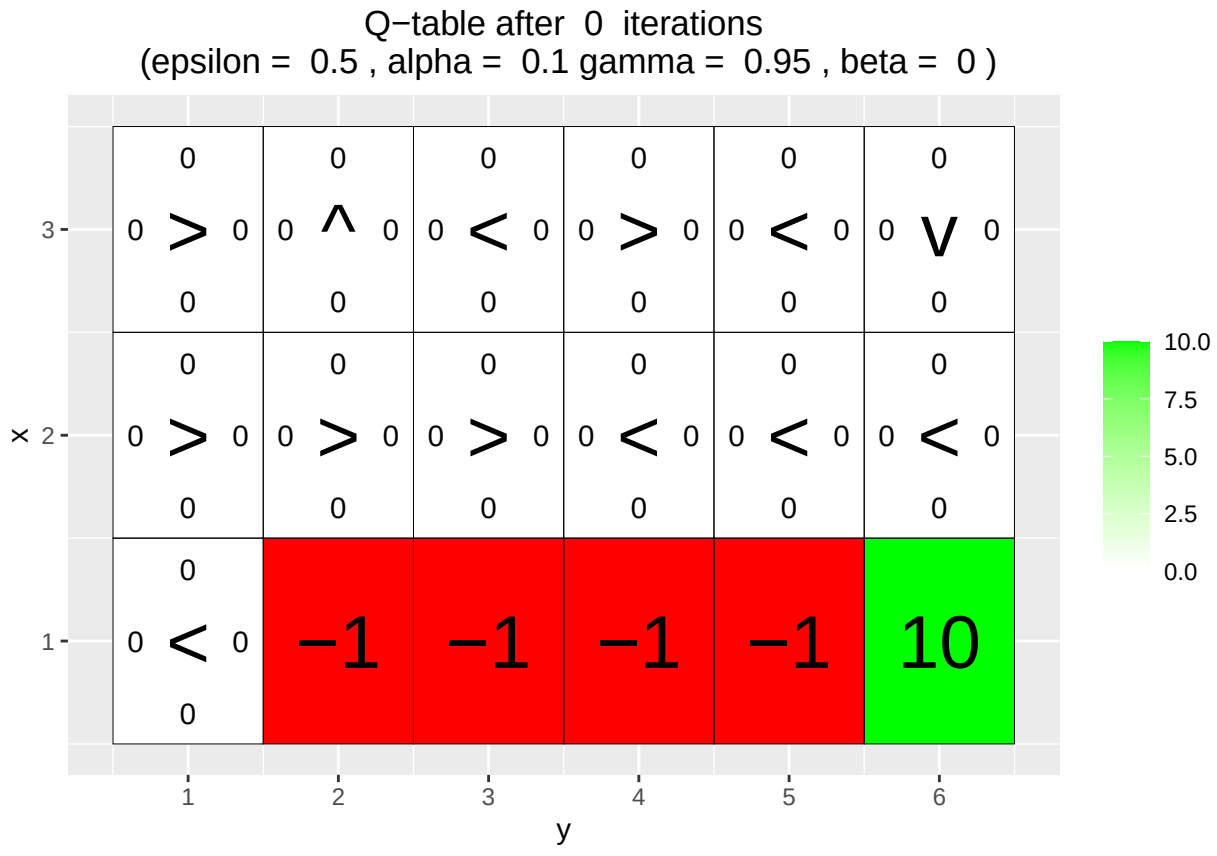


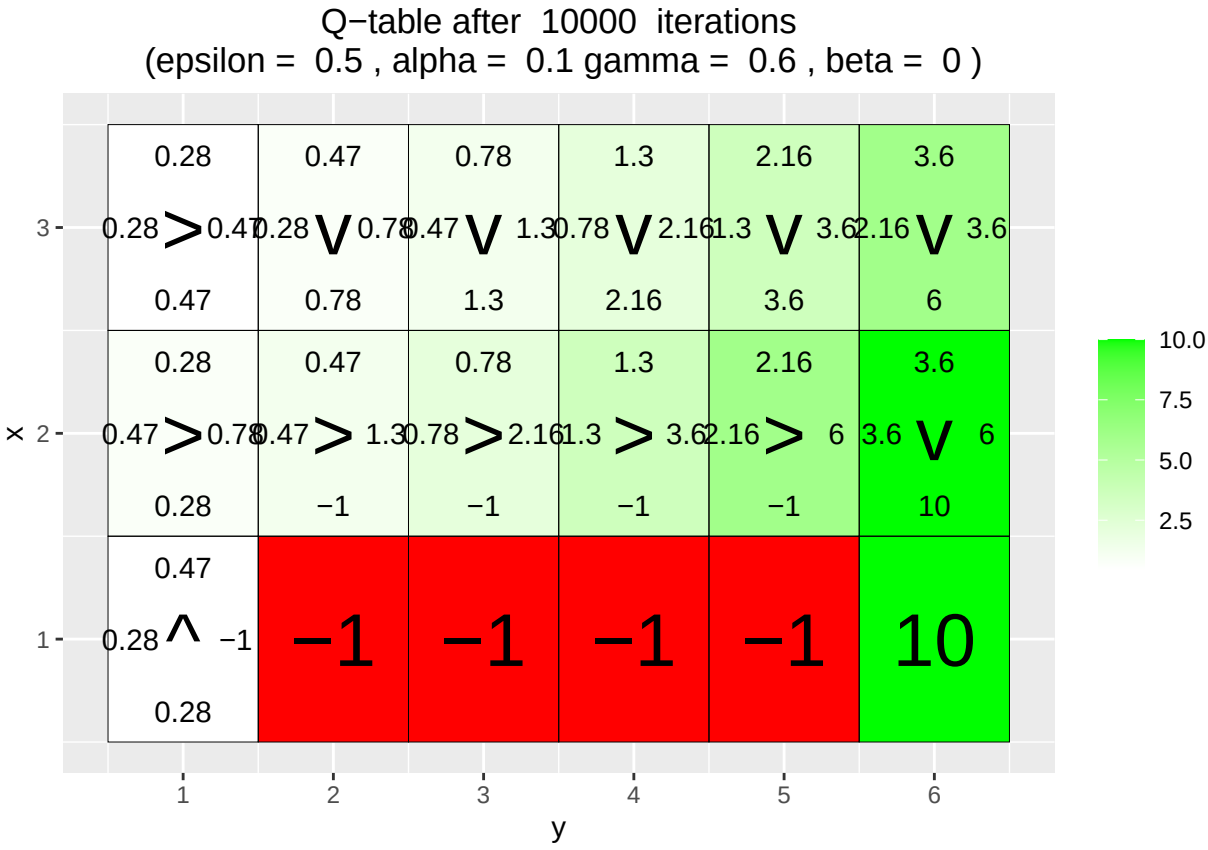
When $\epsilon = 0.1$, the agent becomes non exploratory in the sense that once it reaches the terminal state of reward block 5, it starts to exploit the path with the highest q-values leading to the same reward block and doesn't explore for another terminal state i.e., in this case reward block 10.

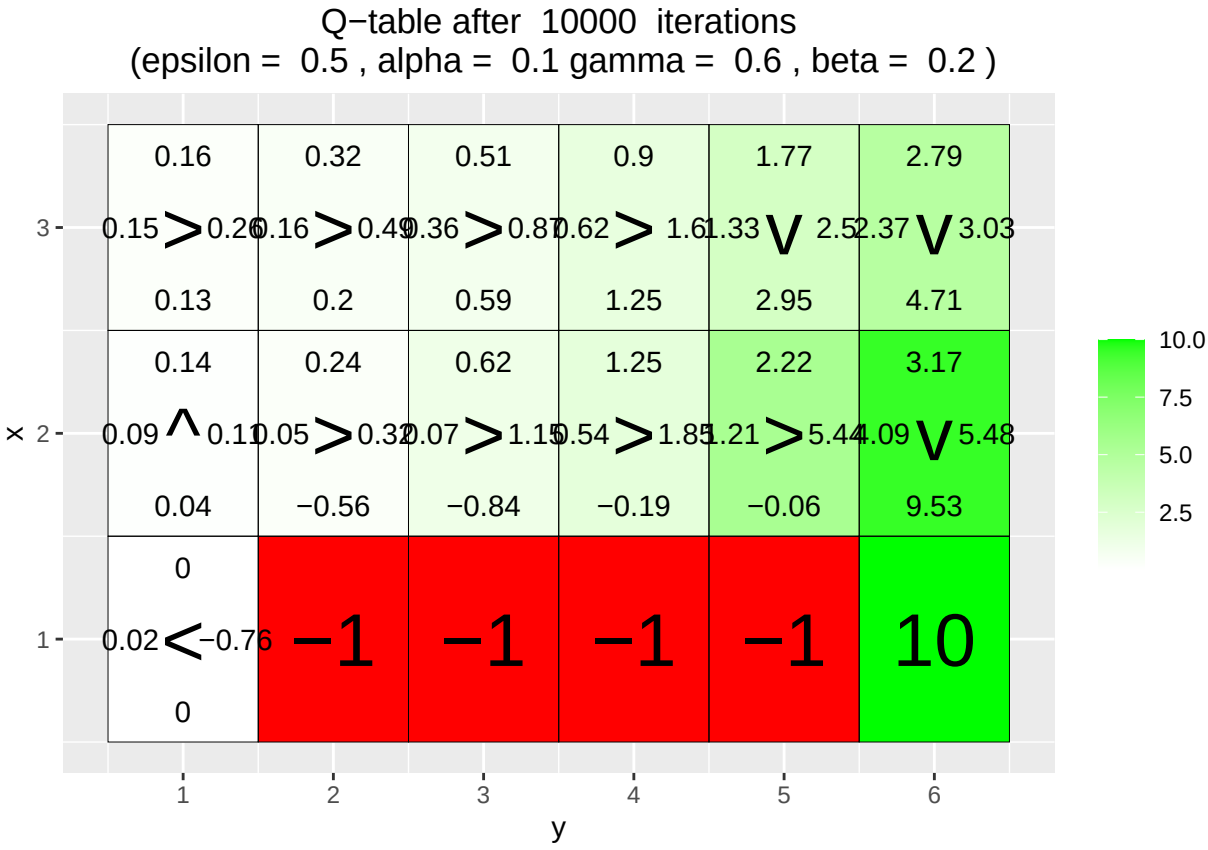
Because of this exploitative nature of the agent, increasing the discount factor doesn't change its behavior as it almost always reaches the terminal state of reward block 5. The same can confirmed from the moving average plots for rewards.

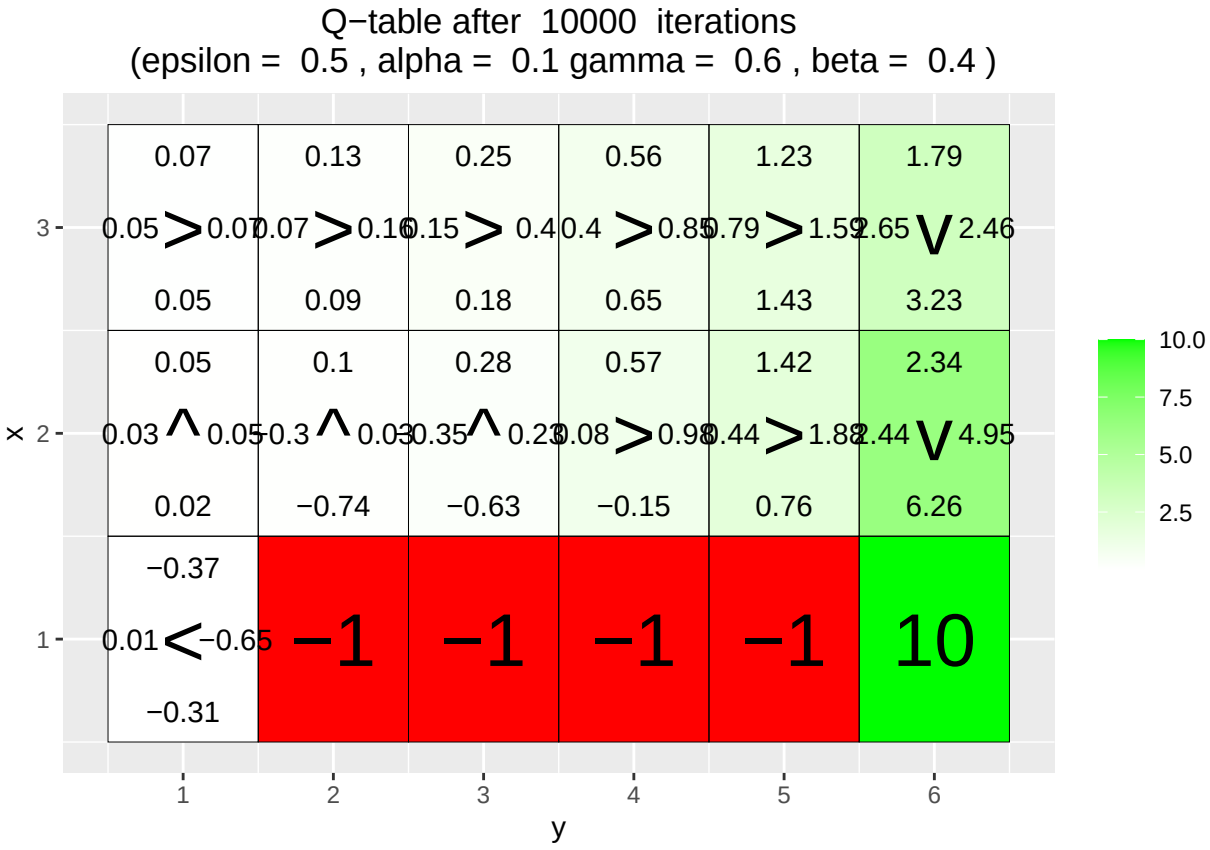
Environment C

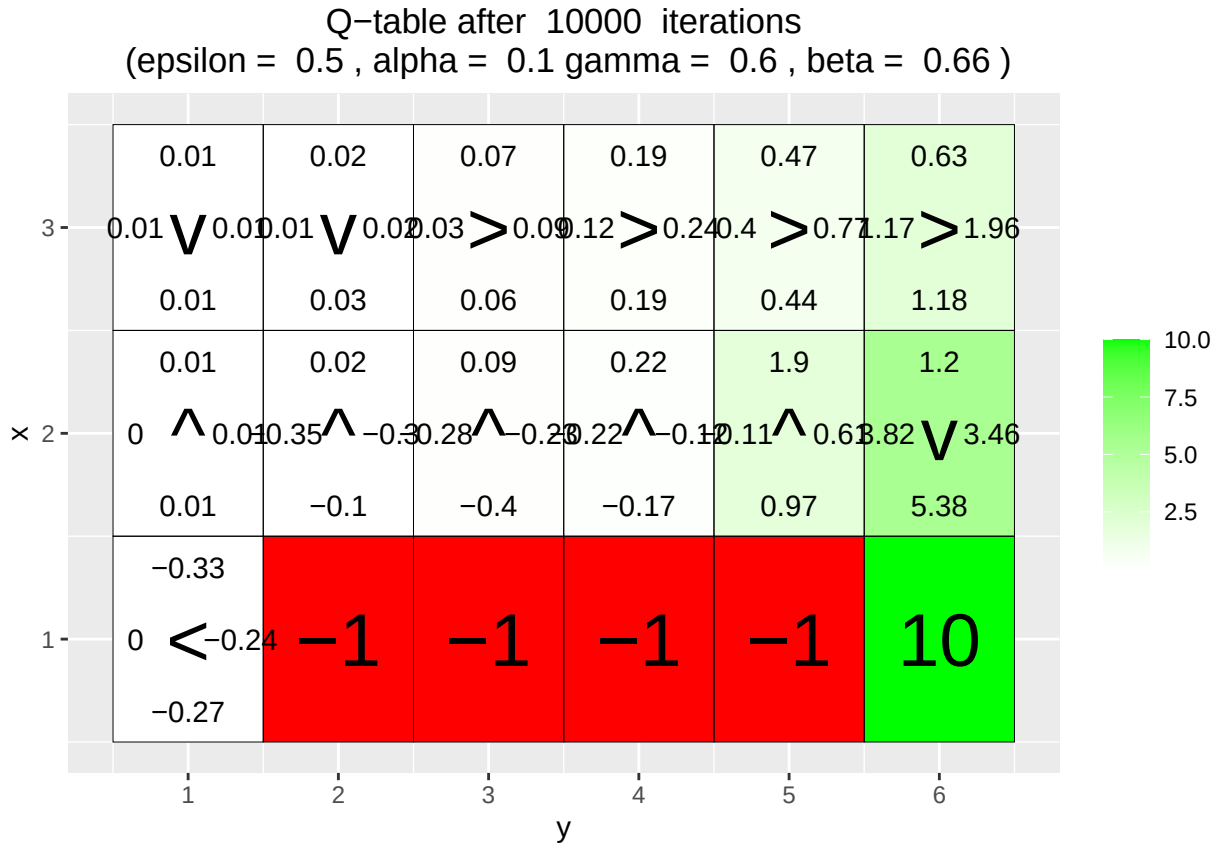
```
# Environment C (the effect of beta).

H <- 3
W <- 6

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

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)
}
```




Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0)
0.28
0.47
0.78
1.3
2.16
3.6
0.28 > 0.47
0.28 V 0.78
0.47 V 1.3
0.78 V 2.16
1.3 V 3.6
2.16 V 3.6
0.47
0.78
1.3
2.16
3.6
6
0.28
0.47
0.78
1.3
2.16
3.6
0.47 > 0.78
0.47 > 1.3
0.78 > 2.16
1.3 > 3.6
2.16 > 6
3.6 V 6
0.28
−1
−1
−1
−1
10
0.47
0.28 ^ −1
−1
−1
−1
−1
10
0.28
3
2
1
x
1
2
3
4
5
6
y
10.0
7.5
5.0
2.5


Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.2)
0.16
0.15 > 0.26
0.13
0.32
0.16 > 0.49
0.2
0.51
0.36 > 0.87
0.59
0.9
0.62 > 1.61
1.25
1.77
1.33 V 2.52
2.95
2.79
2.37 V 3.03
4.71
0.14
0.09 ^ 0.11
0.04
0.24
0.05 > 0.32
−0.56
0.62
0.07 > 1.15
−0.84
1.25
0.54 > 1.85
−0.19
2.22
1.21 > 5.44
−0.06
3.17
4.09 V 5.48
9.53
0
0.02 < −0.76
0
−1
−1
−1
−1
10
3
2
1
x
1
2
3
4
5
6
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)
0.07
0.05 > 0.07
0.05
0.13
0.07 > 0.16
0.09
0.25
0.15 > 0.4
0.18
0.56
0.4 > 0.85
0.65
1.23
0.79 > 1.59
1.43
1.79
2.65 V 2.46
3.23
0.05
0.03 ^ 0.05
0.02
0.1
0.3 ^ 0.03
−0.74
0.28
0.35 ^ 0.23
−0.63
0.57
0.08 > 0.98
−0.15
1.42
0.44 > 1.88
0.76
2.34
2.44 V 4.95
6.26
−0.37
0.01 < −0.65
−0.31
−1
−1
−1
−1
10
x
y
1
2
3
4
5
6
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.