AML Lab 3

Harshavardhan Subramanian

08/10/2020

2.1

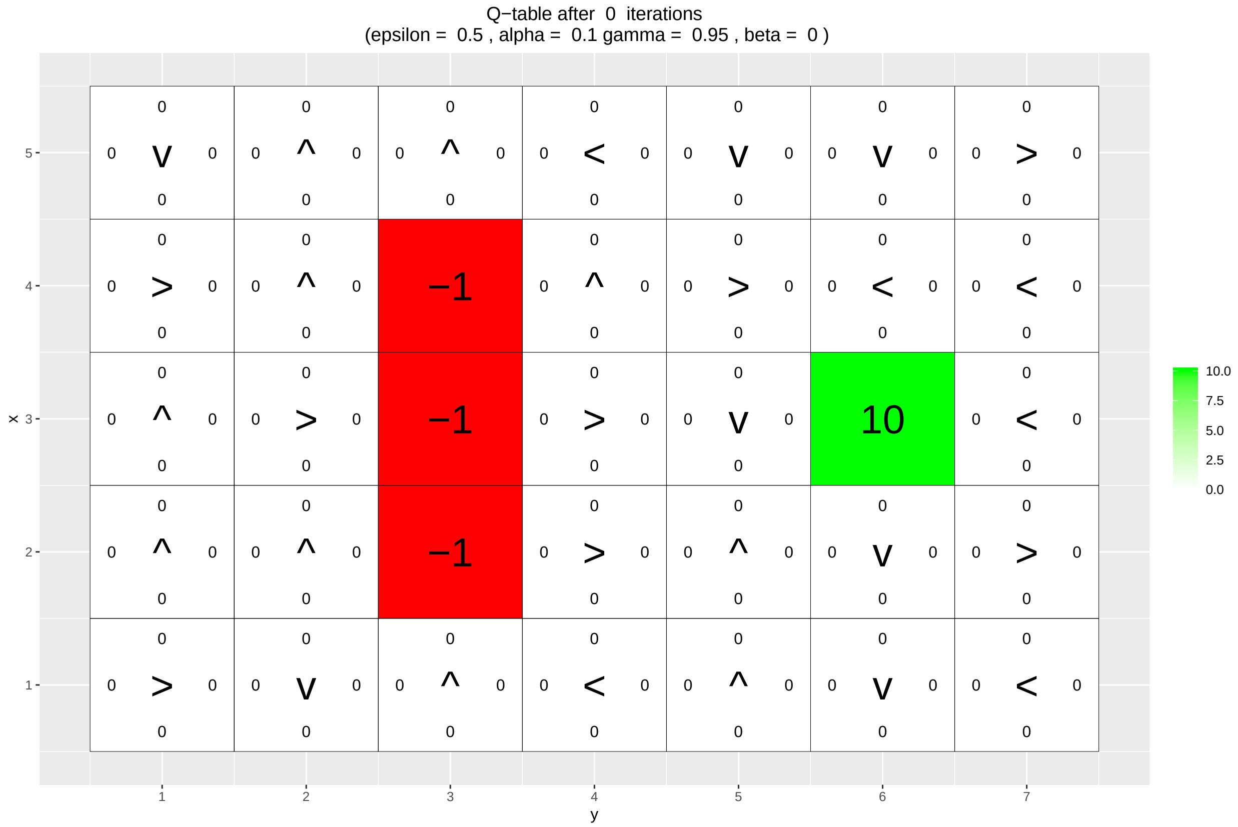
```r
library(ggplot2)
#library(vctrs)


#tempdir()
# [1] "C:\Users\XYZ~1\AppData\Local\Temp\Rtmp86bEoJ\Rtxt32dcef24de2"
#dir.create(tempdir())

# If you do not see four arrows in line 16, then do the following:
# File/Reopen with Encoding/UTF-8

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)))

}

GreedyPolicy <- function(x, y)
{
  ind <- q_table[x,y,]
  max_ind <- which(ind == max(ind))
  #print(max_ind)
  #print(length(max_ind))
  if(length(max_ind) == 1)
  {
    return(max_ind)
  }
  else
  {
  return(sample(max_ind,1))
  }
}

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.
  ind <- q_table[x,y,]
  ind[which(is.na(ind))] <- 0
  max_ind <- which(ind == max(ind))
  r <- runif(1)
  if(r > epsilon)
  {
    if(length(max_ind) == 1)
    {
```

```r
      return(max_ind)
    }
    else
    {
      return(sample(max_ind,1))
    }

  }
  else
  {
    return(sample(c(1:4),1))
  }
}



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)
}

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 greedily.
  #   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.
  reward <- 0
  episode_correction <- 0
  cur_st_x <- start_state[1]
  cur_st_y <- start_state[2]

  repeat{
    # Follow policy, execute action, get reward.
    act <- EpsilonGreedyPolicy(cur_st_x,cur_st_y,epsilon)
    new_state <- transition_model(cur_st_x,cur_st_y, act, beta)
    #print(new_state)
    next_st_x <- new_state[1]
    next_st_y <- new_state[2]

    greedy_max <- GreedyPolicy(next_st_x,next_st_y)
    #print(greedy_max)
    reward <- reward_map[next_st_x,next_st_y]
    temp_correction <- alpha * (reward + (gamma * q_table[next_st_x,next_st_y,greedy_max]) - q_table[cu
    q_table[cur_st_x,cur_st_y,act] <<- q_table[cur_st_x,cur_st_y,act] + temp_correction
    cur_st_x <- next_st_x
    cur_st_y <- next_st_y
    #reward <- reward + temp_reward
    episode_correction <- episode_correction + temp_correction


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

}
```

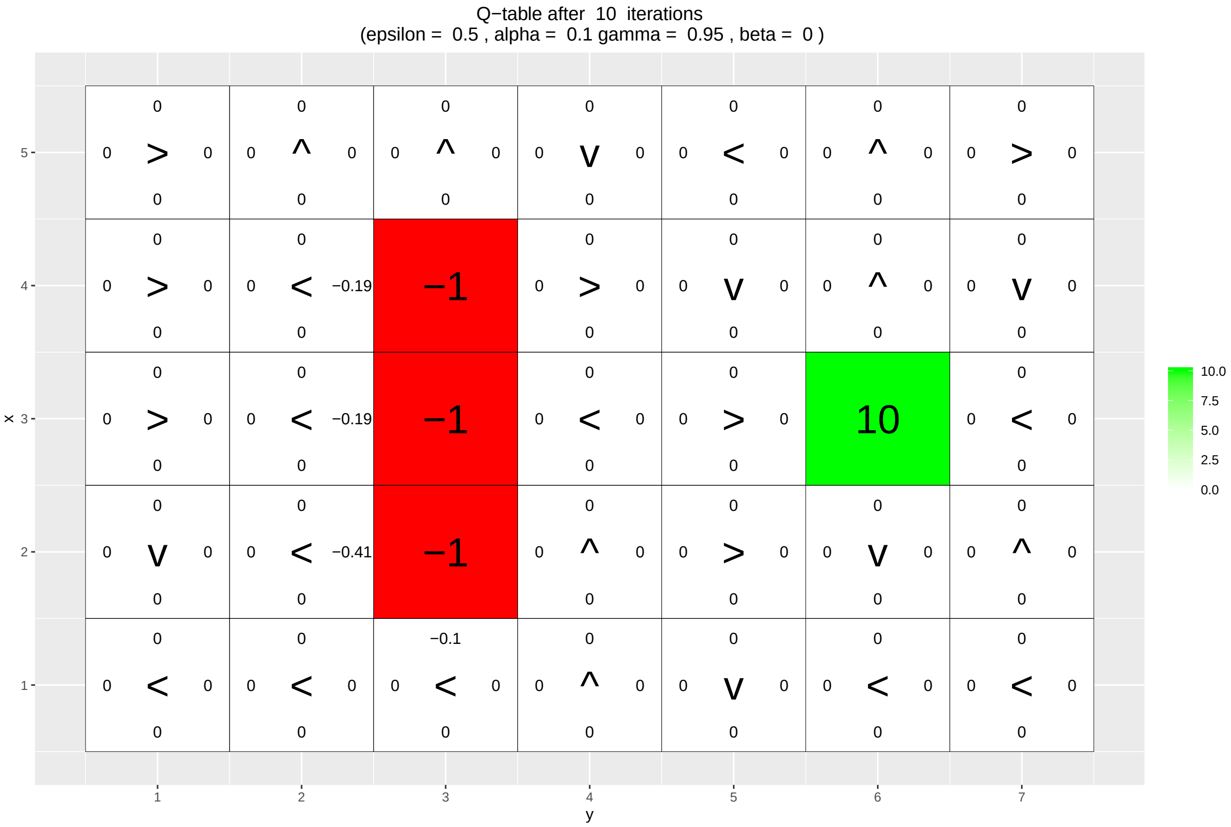
ENV A

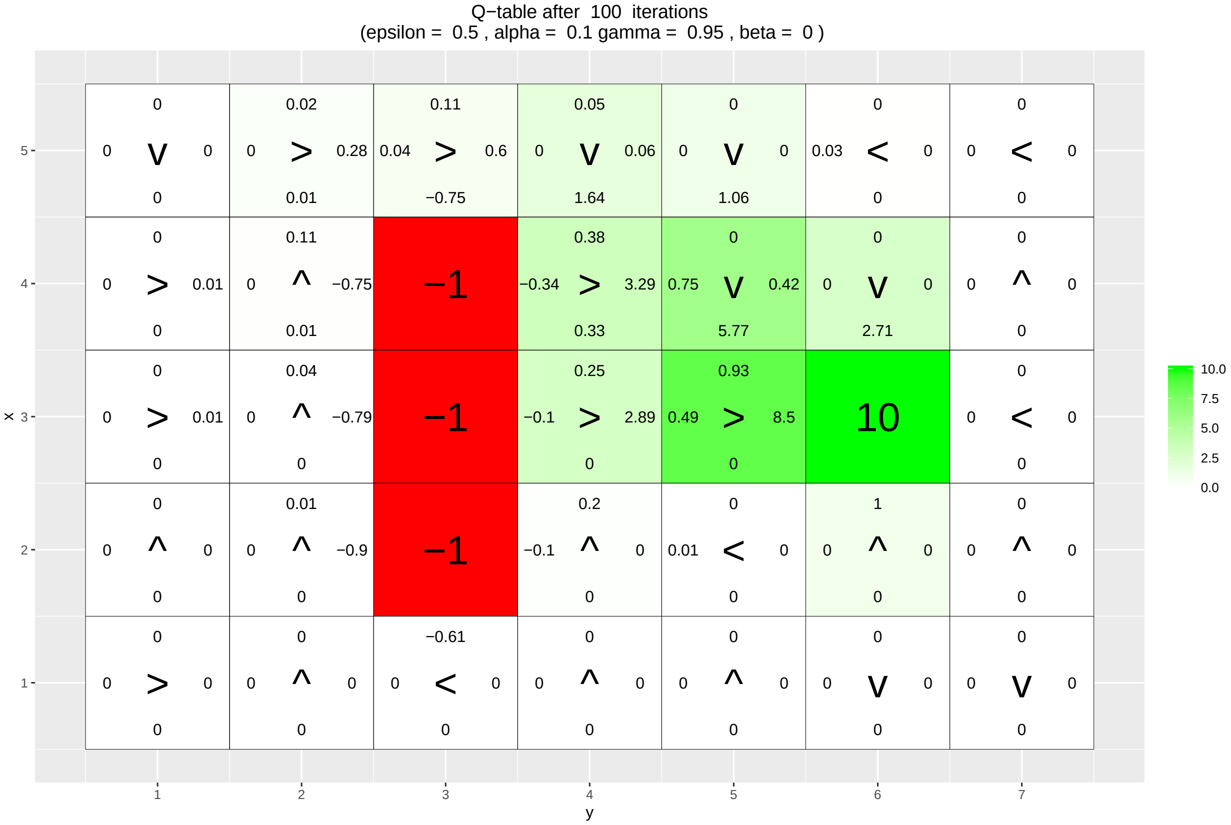
```r
# Environment A (learning)

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))
```

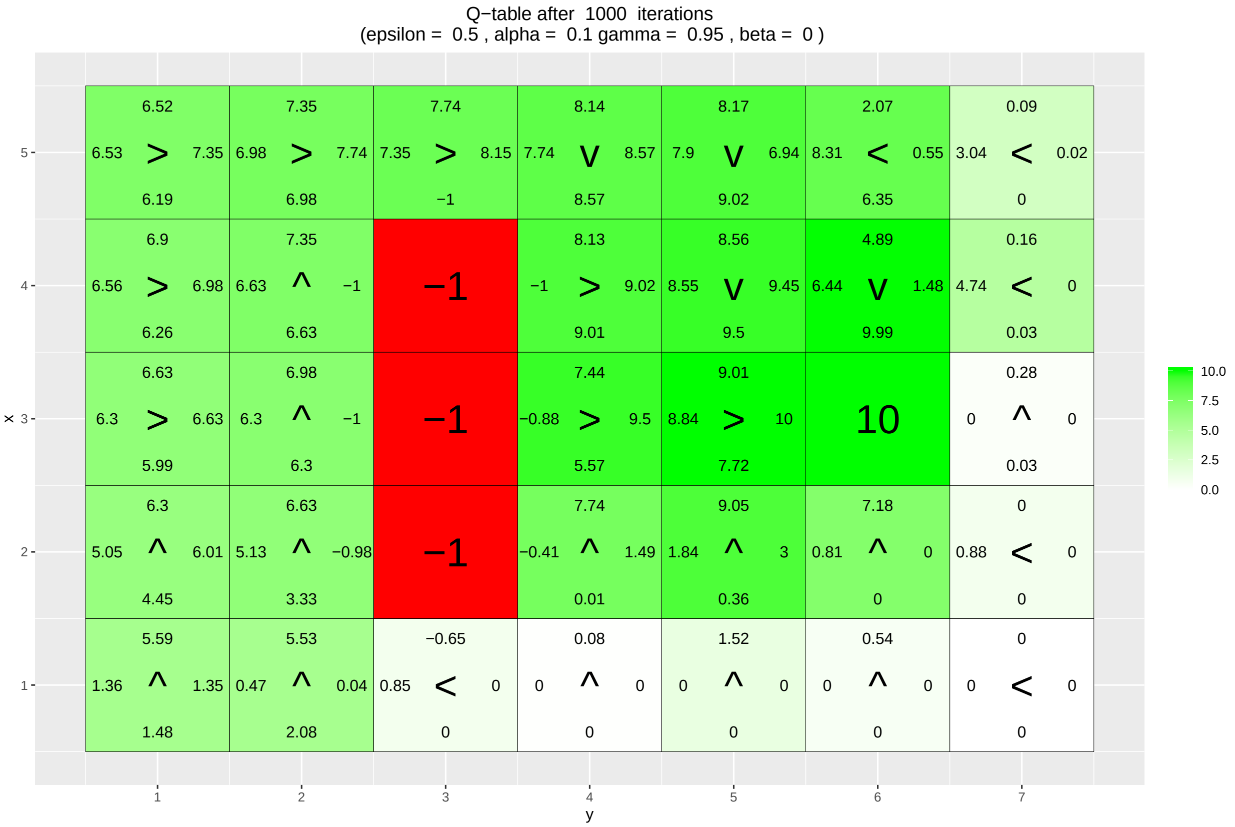
```r
vis_environment()
```

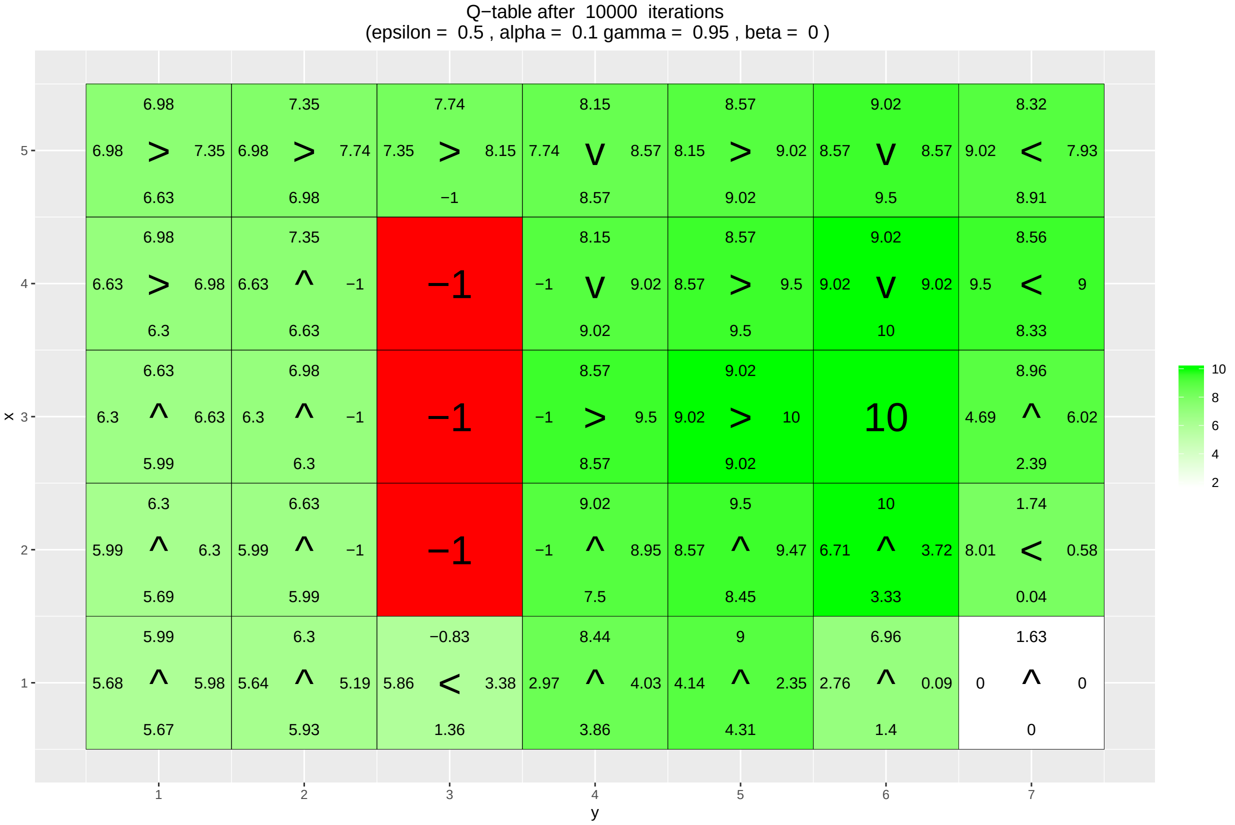


```r
set.seed(1234)
for(i in 1:10000){
  #i = 10
  foo <- q_learning(start_state = c(3,1))

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


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





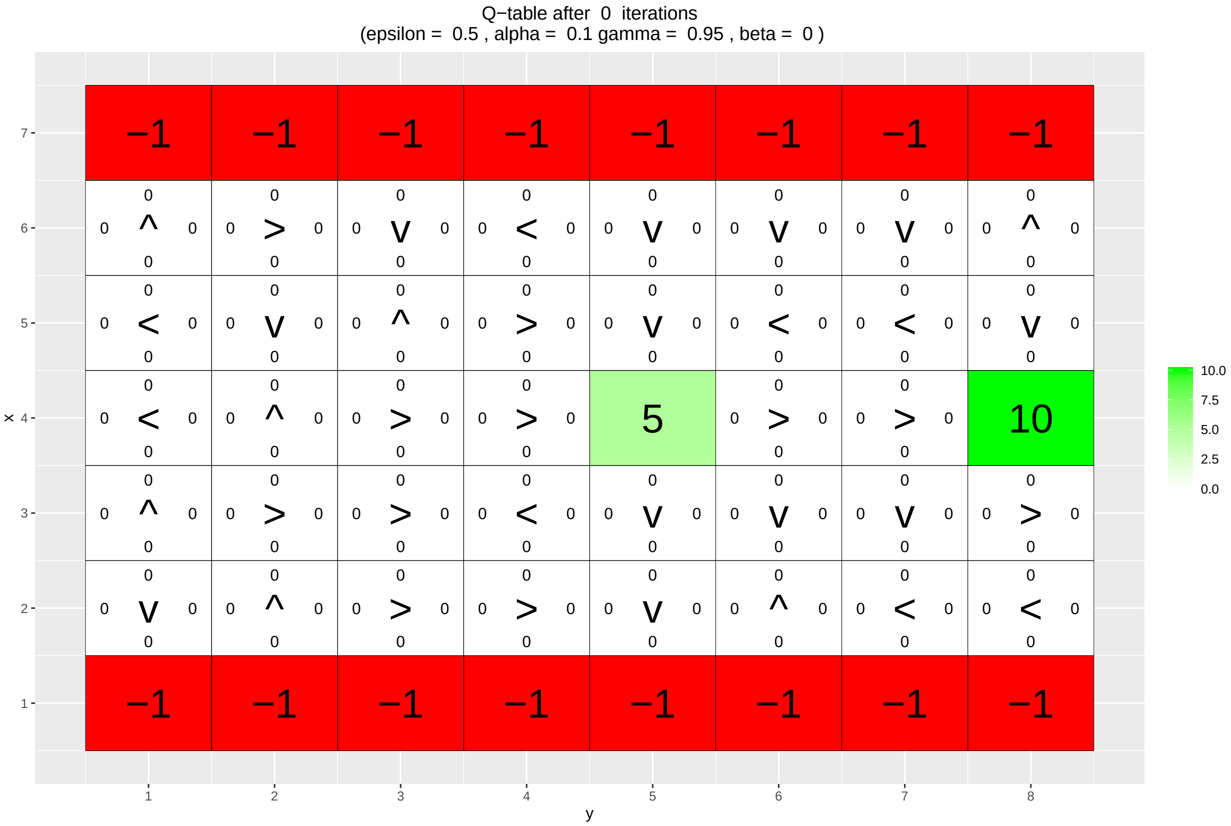
After 10 iterations, the agent is moving in the direction of maximum q value which in this case is 0 for all 4 direction initially and later makes an attempt in reaching the terminal state as early as possible without exploring much of non terminal states.

After 10000 iteration, the agent has learned to take the direction of maximal action according to Epsilon greedy policy and also with the discount factor close to 1, looks for long term reward and moves in direction of terminal with maximal reward. Hence greedy policy can be called optimal.

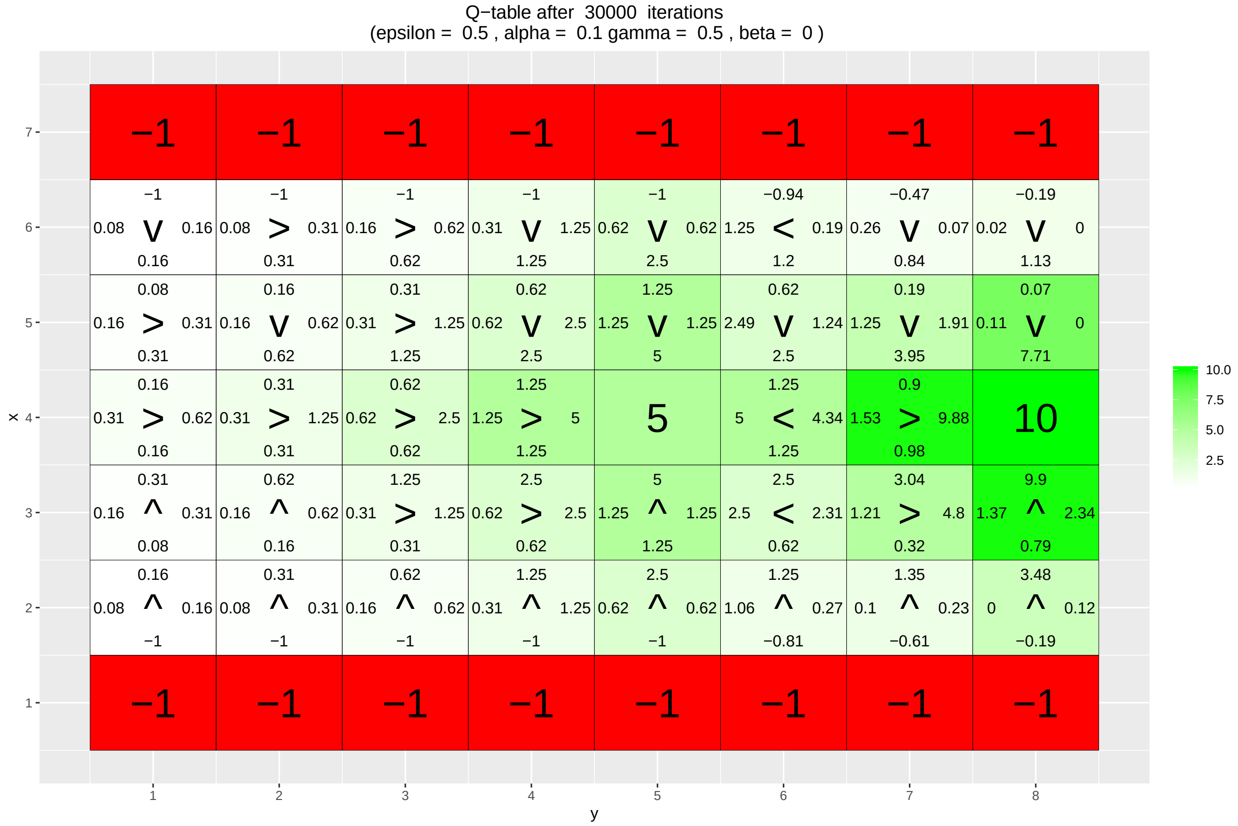
The agent has also tried exploring other alternatives directions leading to terminal state. Here in the Q table plot we can see that the agent has tried alternative(green shades) in reaching the terminal states.

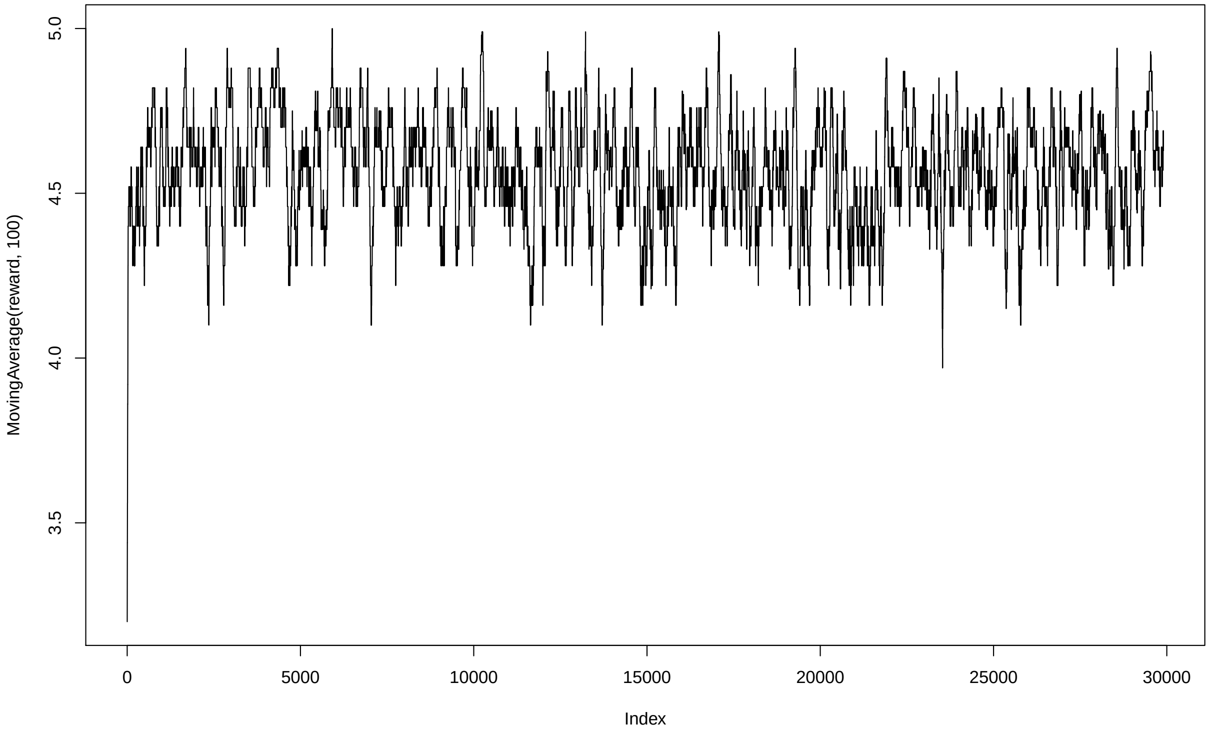
ENV B

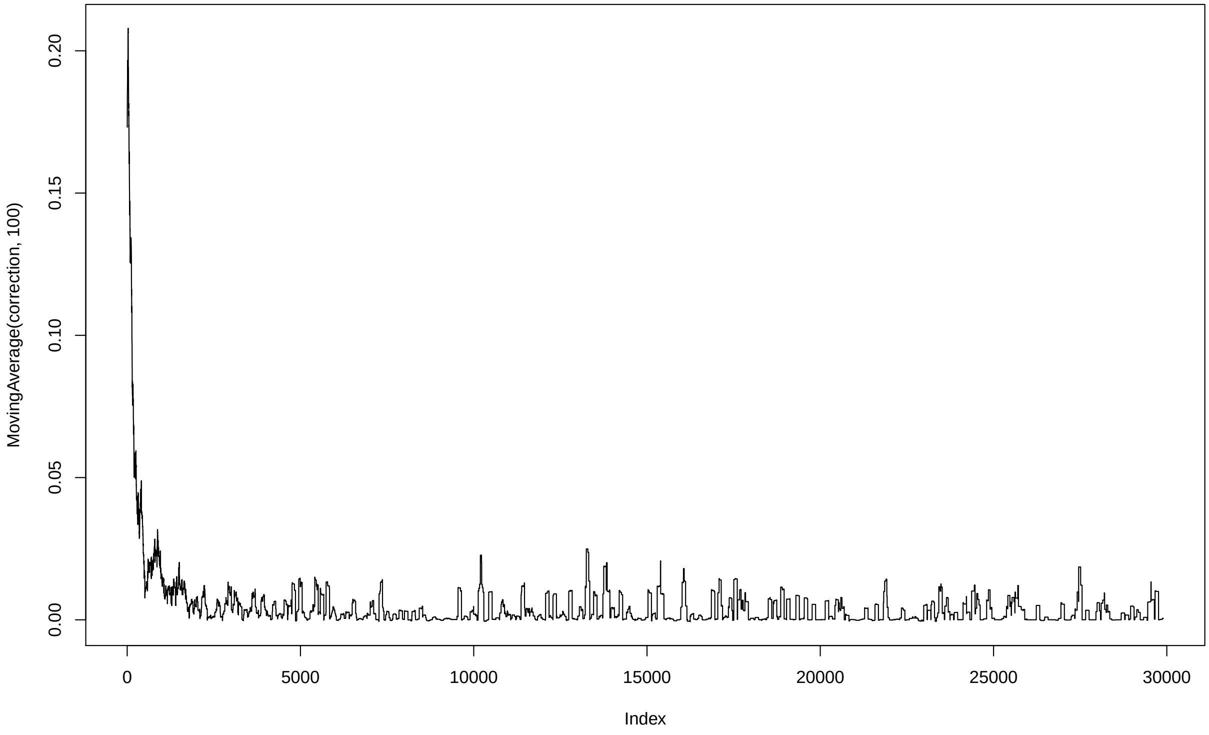
```
# Environment B (the effect of epsilon and gamma)

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()
```



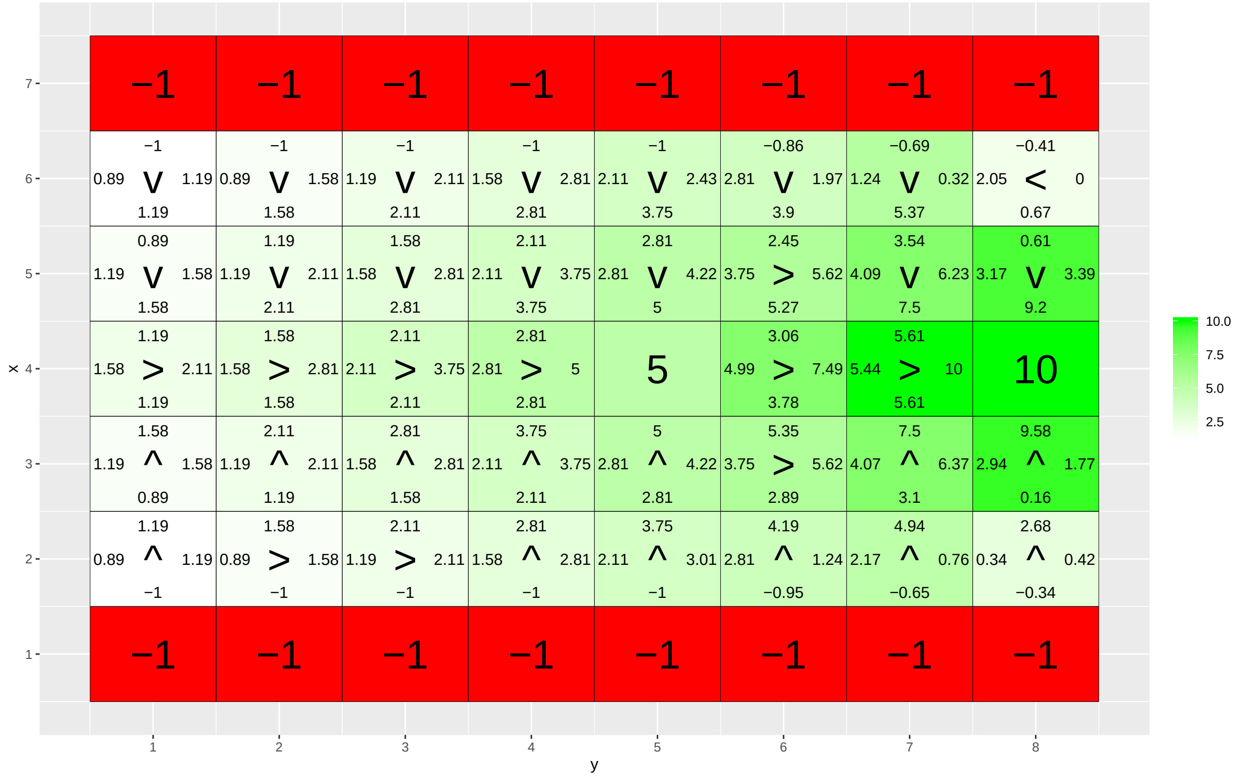
```r
MovingAverage <- function(x, n){

  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)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}
```

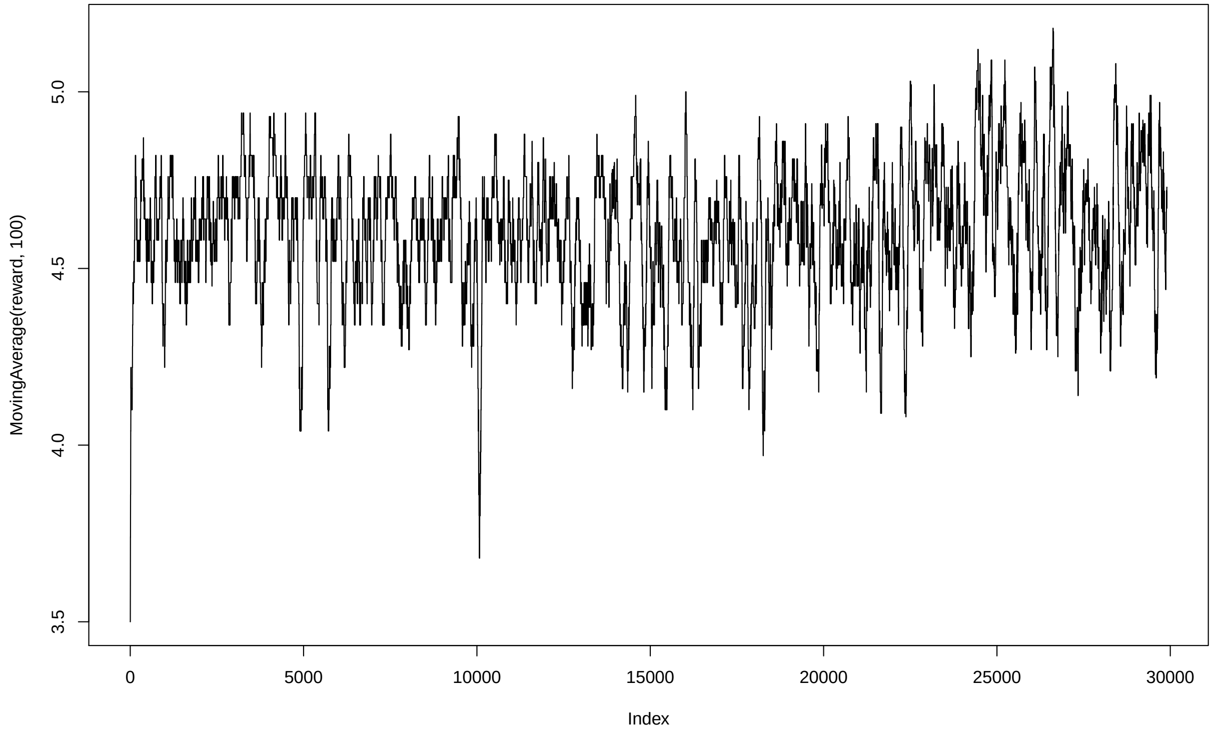


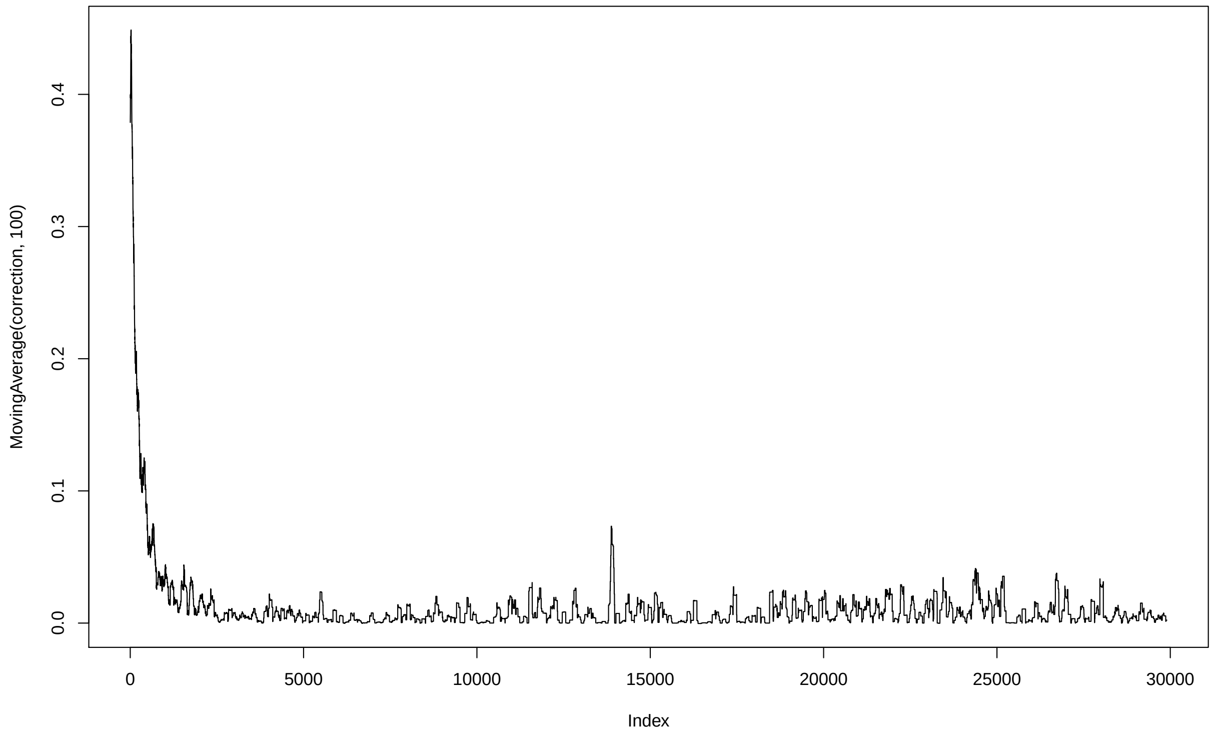

MovingAverage(reward, 100)
5.0
4.5
4.0
3.5
0
5000
10000
15000
20000
25000
30000
Index


MovingAverage(correction, 100)
0.20
0.15
0.10
0.05
0.00
0
5000
10000
15000
20000
25000
30000
Index

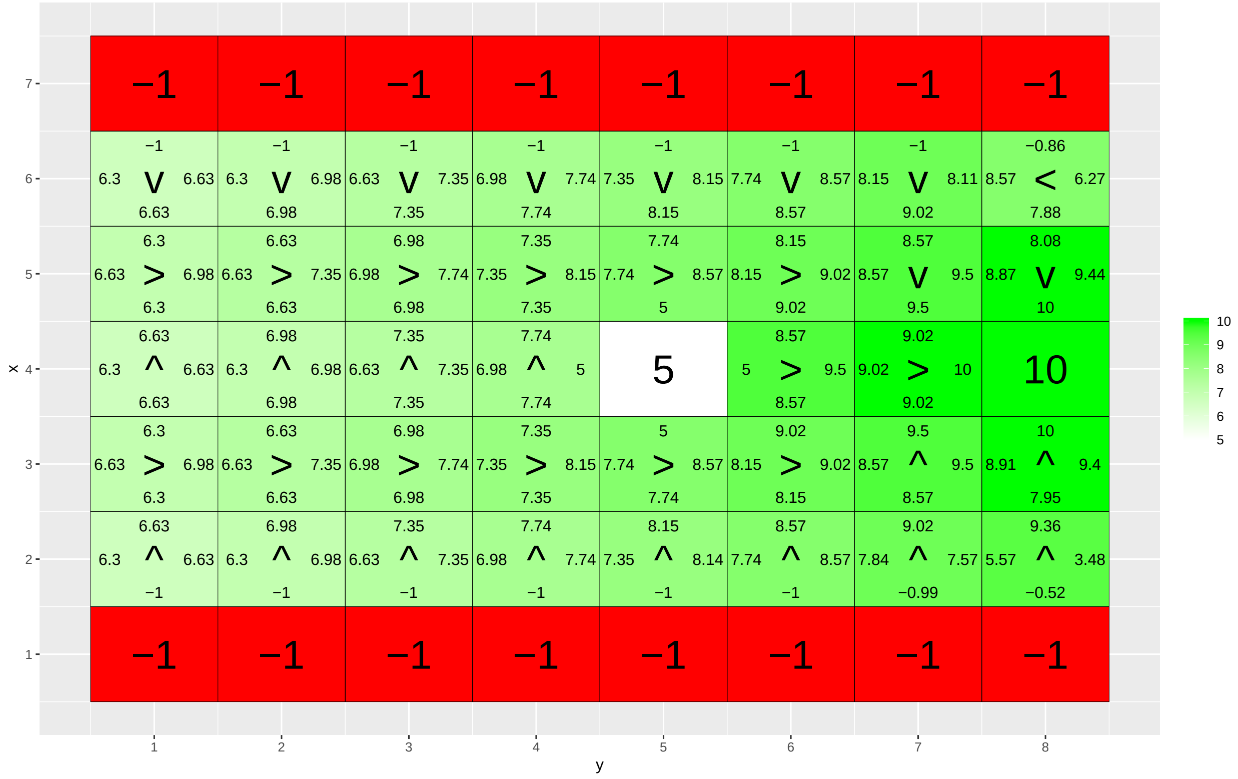
Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0)

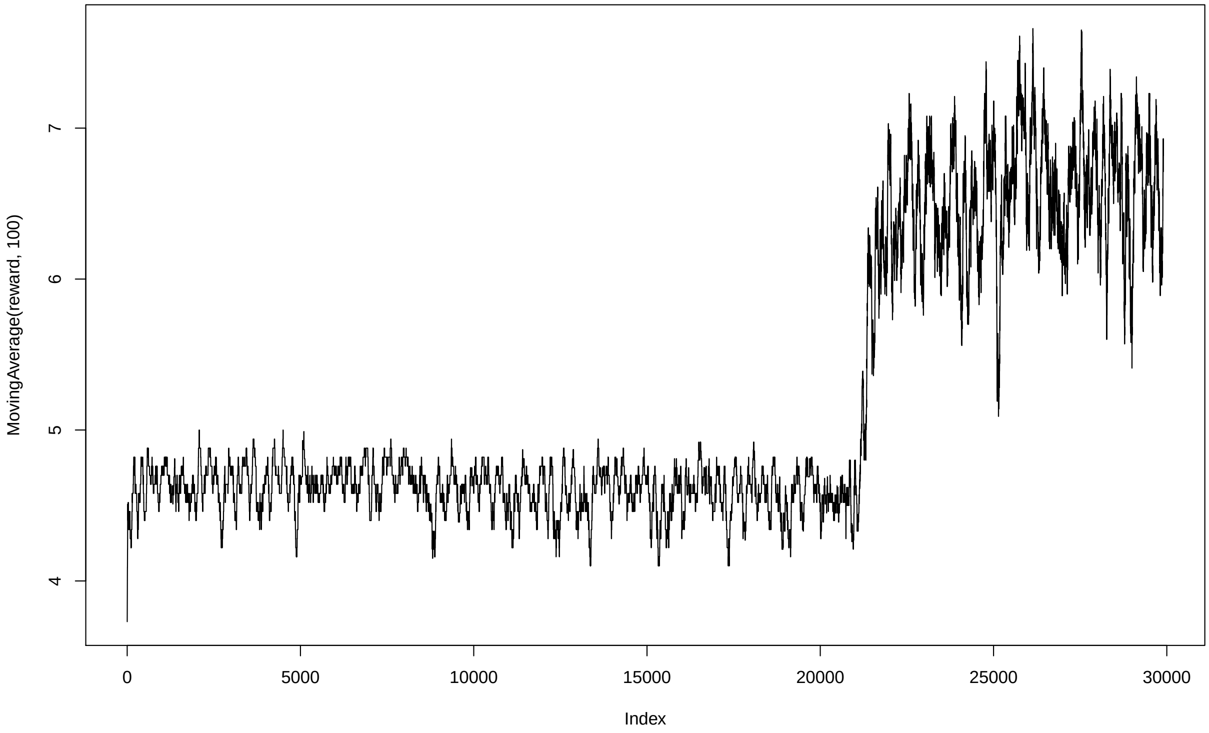


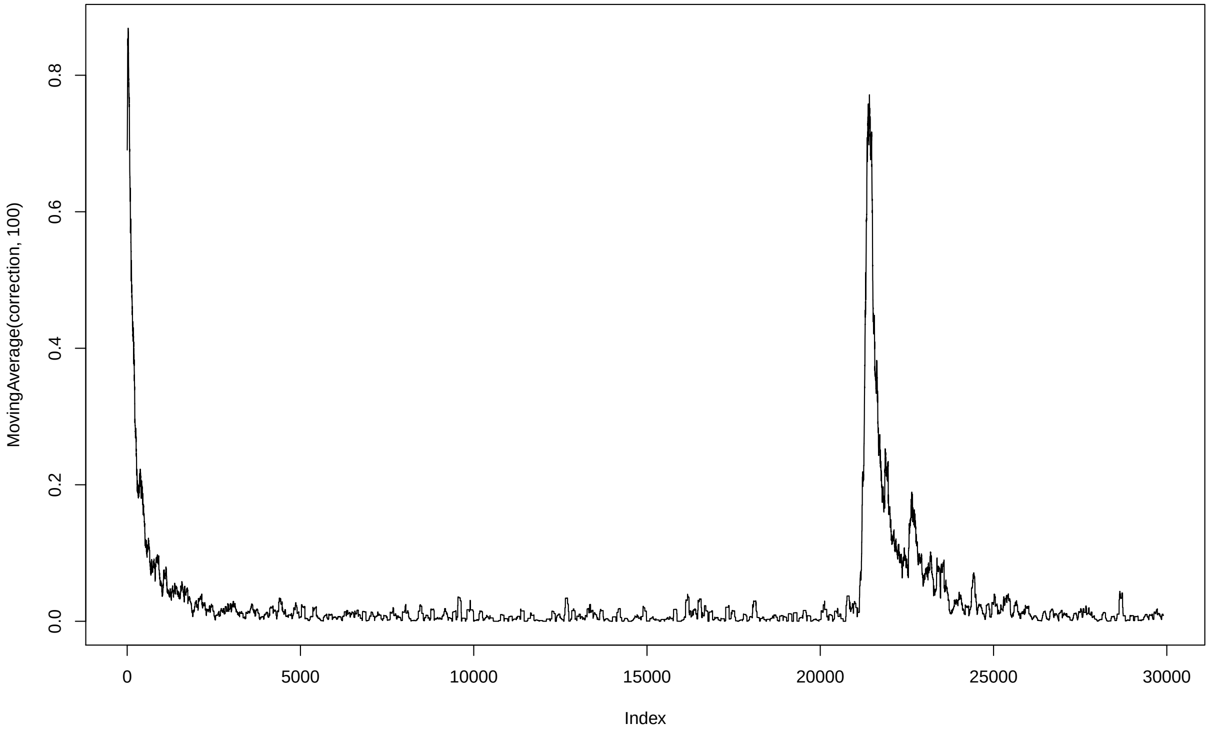




Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)







In the first case, Epsilon is set to 0.5 and Gamma variants of 0.5, 0.75, 0.95

Epsilon parameter deifnes how greedy the policy should be. Setting it 0.5 makes selecting the action with maximum qvalue and non max qvalue with equal probability. # Gamma parameter defines the discount factor, that is how much preference to be given for reward at each iteration. Gamma at low value defines the instant reward and as it increases return of the reward delays. When Gamma is 1, it means it takes long term reward at the end.

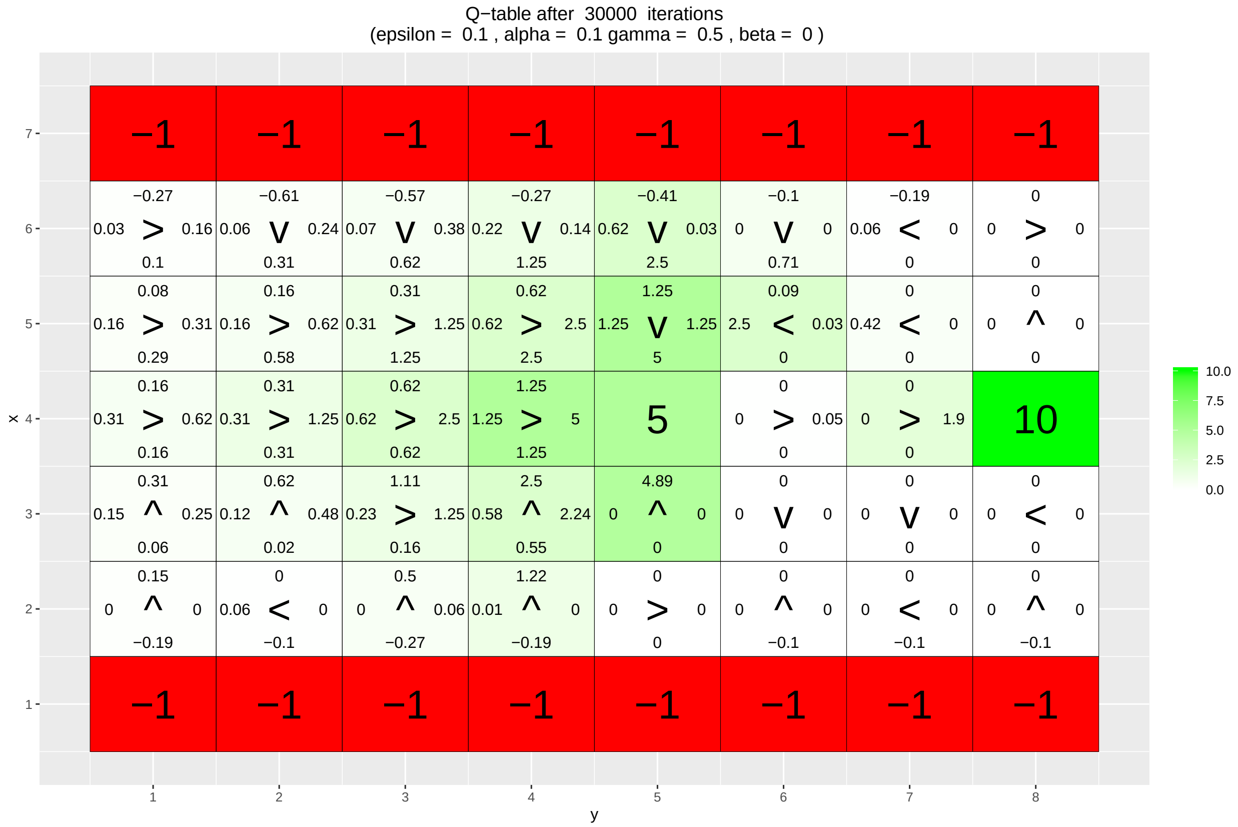
For Epsilon 0.5 and variants of Gamma to 0.5,0.75 and 0.95.

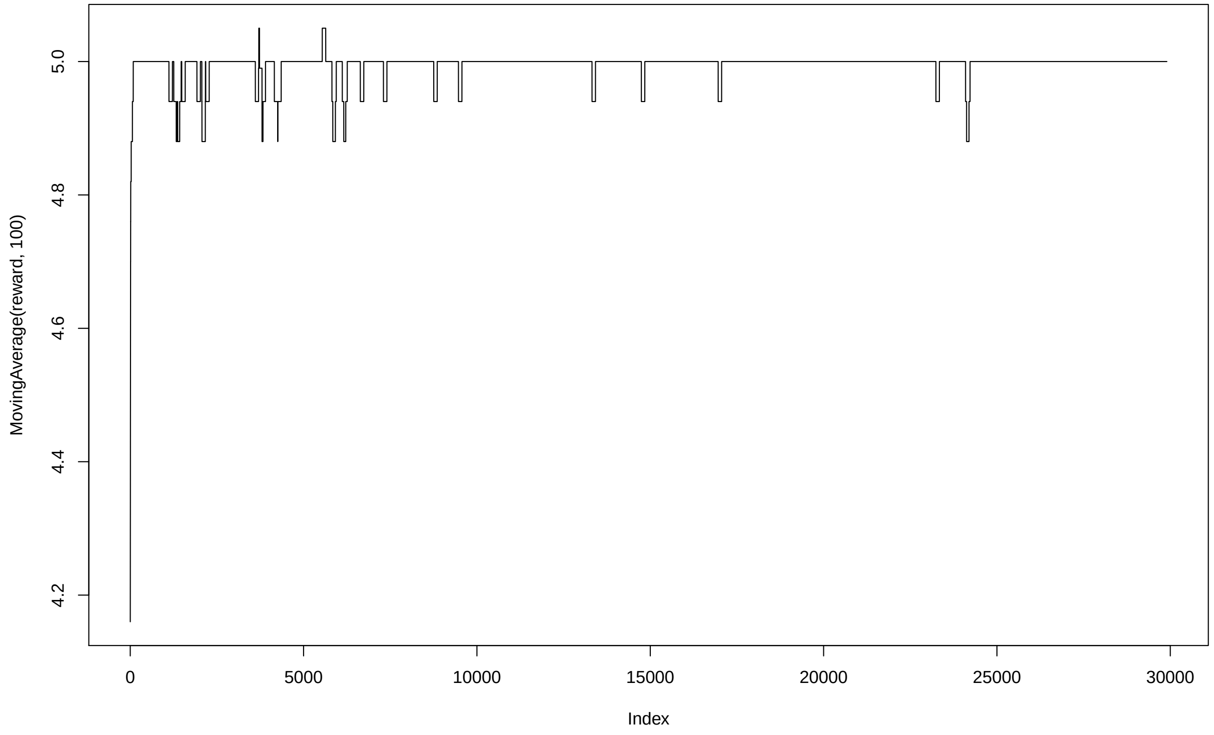
Epsilon = 0.5, Gamma = 0.5 => In this variant, the agent learns to take maximal action and also tries to find the nearby terminal state without exploring much. Since the low gamma value means instant reward. In this case it tries to settle at 5 terminal state since it has the closest reward as gamma is less than 10 terminal state.

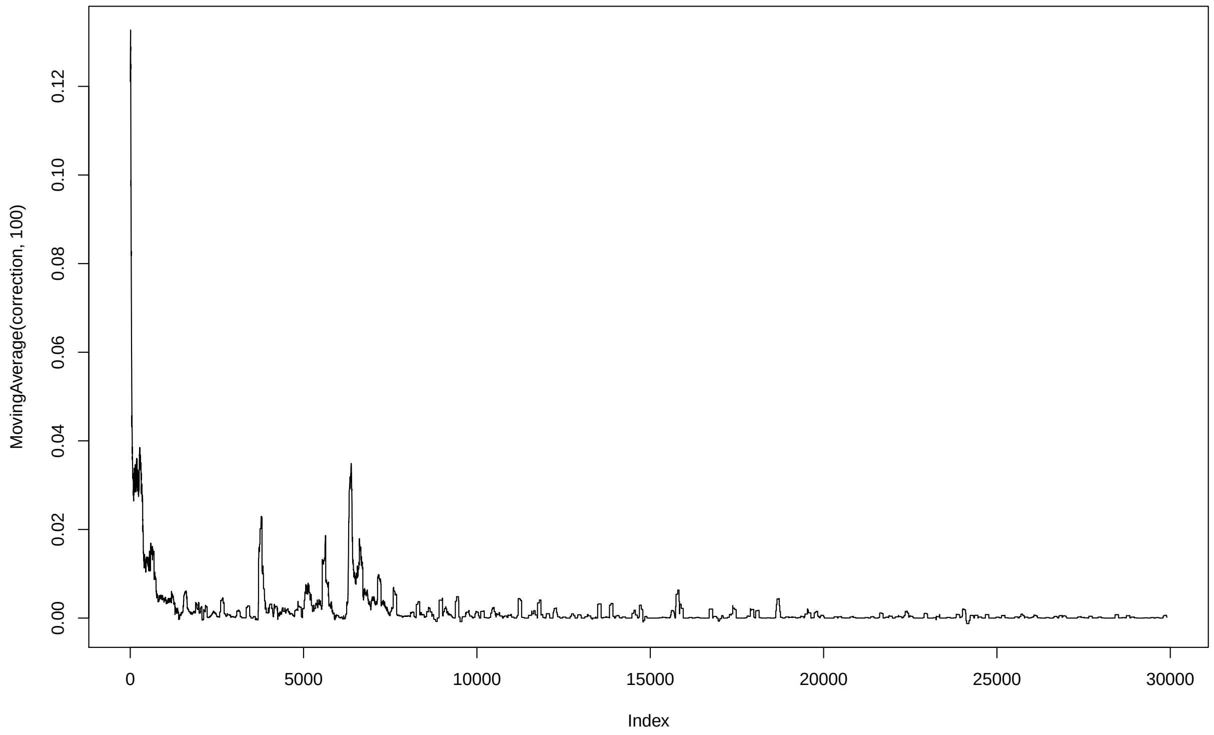
Epsilon = 0.5, Gamma = 0.75 => The second variant performed better than previous and was able to reach the minimum reward around 14000 iterations and learns to maximize the reward to 10 by moving in the direction of that terminal state.

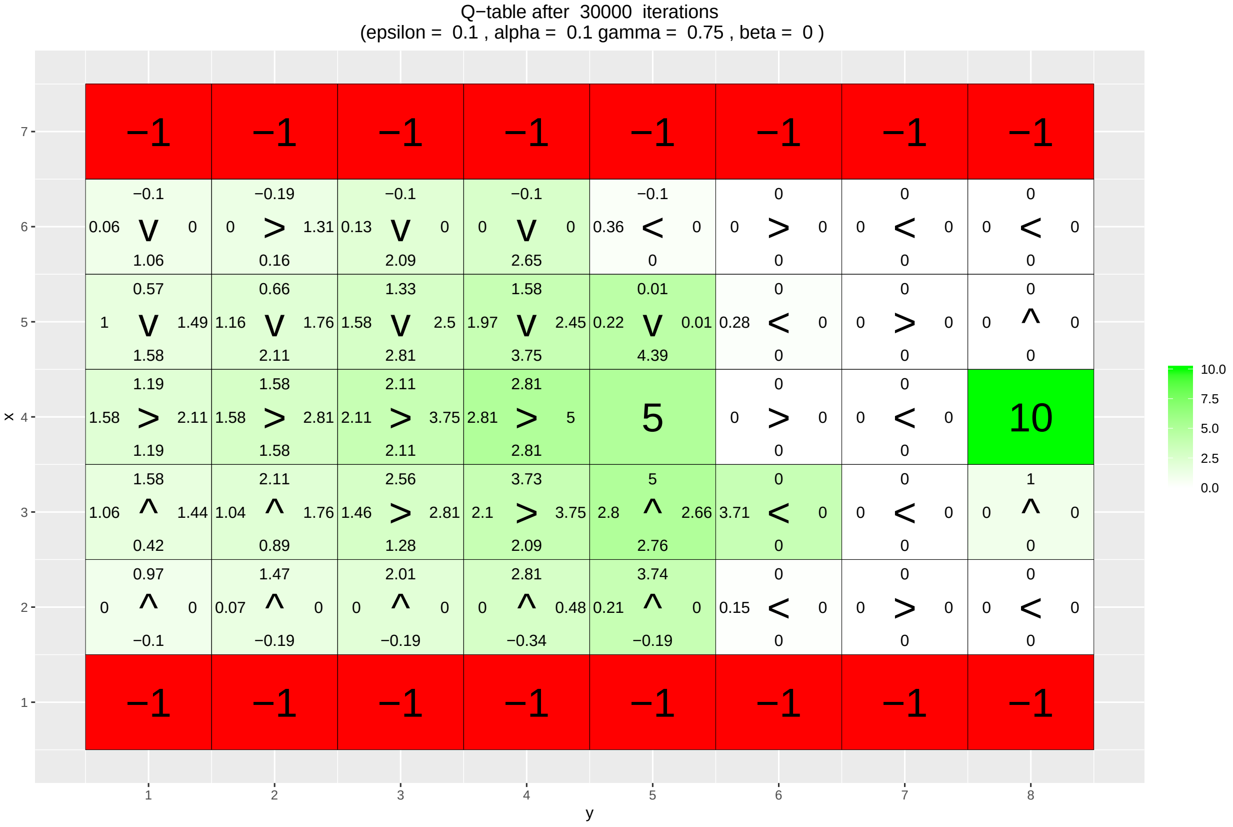
Epsilon = 0.5, Gamma = 0.95 => The third variant resulted in good policy by maximizing the action at each step and also exploring the alternative direction better compared to previous variants. The higher gamma means preferring the rewards at the long term. If we observe the Moving average graph for rewards, there is a sudden shift of rewards from 5 to towards 10 meaning as soon as it finds there is another max value apart from 5, it heads towards that direction.

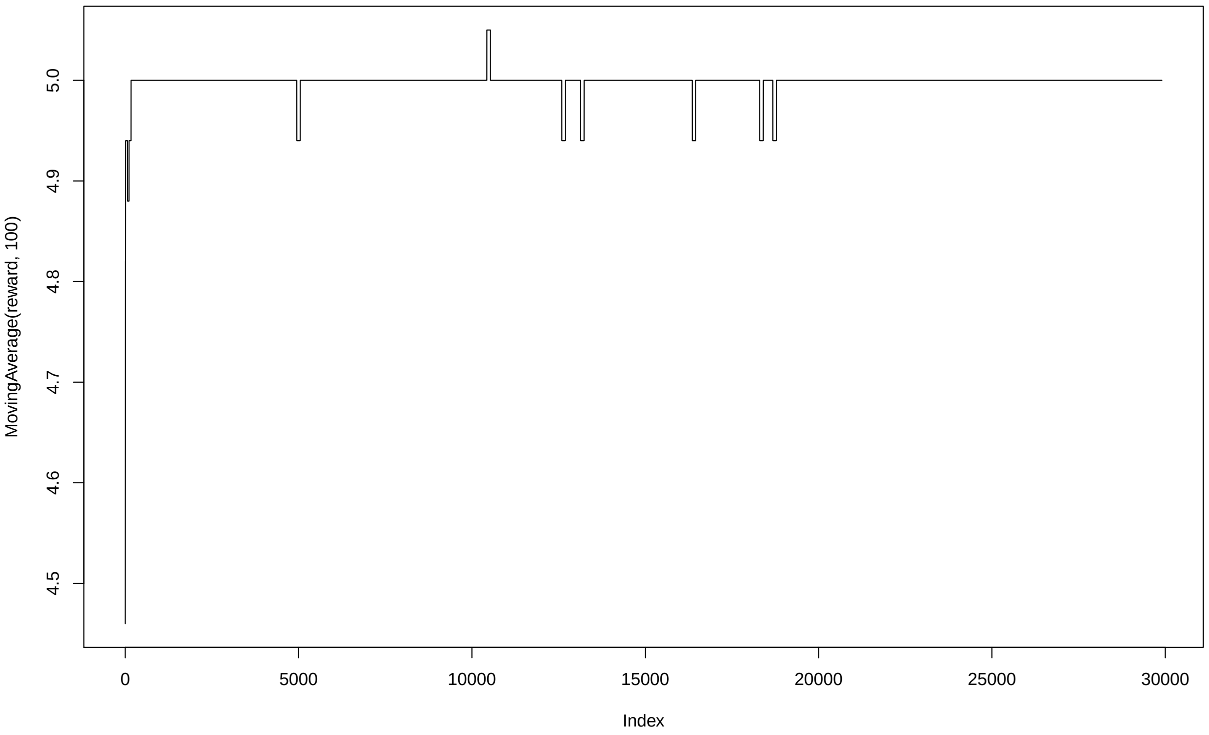
```r
for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  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)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}
```

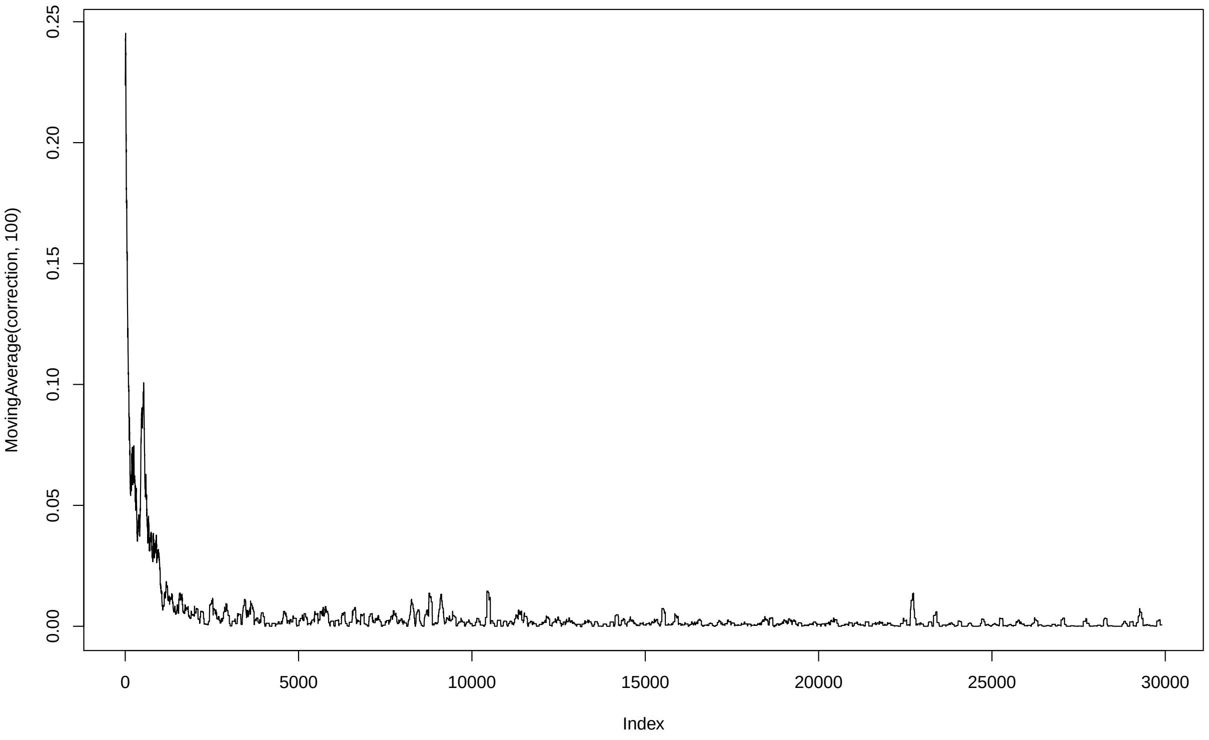

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.5 , beta = 0)
7
−1 −1 −1 −1 −1 −1 −1 −1
−0.27 −0.61 −0.57 −0.27 −0.41 −0.1 −0.19 0
6
0.03 > 0.16 0.06 V 0.24 0.07 V 0.38 0.22 V 0.14 0.62 V 0.03 0 V 0 0.06 < 0 0 > 0
0.1 0.31 0.62 1.25 2.5 0.71 0 0
0.08 0.16 0.31 0.62 1.25 0.09 0 0
5
0.16 > 0.31 0.16 > 0.62 0.31 > 1.25 0.62 > 2.5 1.25 V 1.25 2.5 < 0.03 0.42 < 0 0 ^ 0
0.29 0.58 1.25 2.5 5 0 0 0
0.16 0.31 0.62 1.25 0 0
x 4
0.31 > 0.62 0.31 > 1.25 0.62 > 2.5 1.25 > 5 5 0 > 0.05 0 > 1.9 10
0.16 0.31 0.62 1.25 0 0
0.31 0.62 1.11 2.5 4.89 0 0 0
3
0.15 ^ 0.25 0.12 ^ 0.48 0.23 > 1.25 0.58 ^ 2.24 0 ^ 0 0 V 0 0 V 0 0 < 0
0.06 0.02 0.16 0.55 0 0 0 0
0.15 0 0.5 1.22 0 0 0 0
2
0 ^ 0 0.06 < 0 0 ^ 0.06 0.01 ^ 0 0 > 0 0 ^ 0 0 < 0 0 ^ 0
−0.19 −0.1 −0.27 −0.19 0 −0.1 −0.1 −0.1
1
−1 −1 −1 −1 −1 −1 −1 −1
1 2 3 4 5 6 7 8
y
10.0
7.5
5.0
2.5
0.0

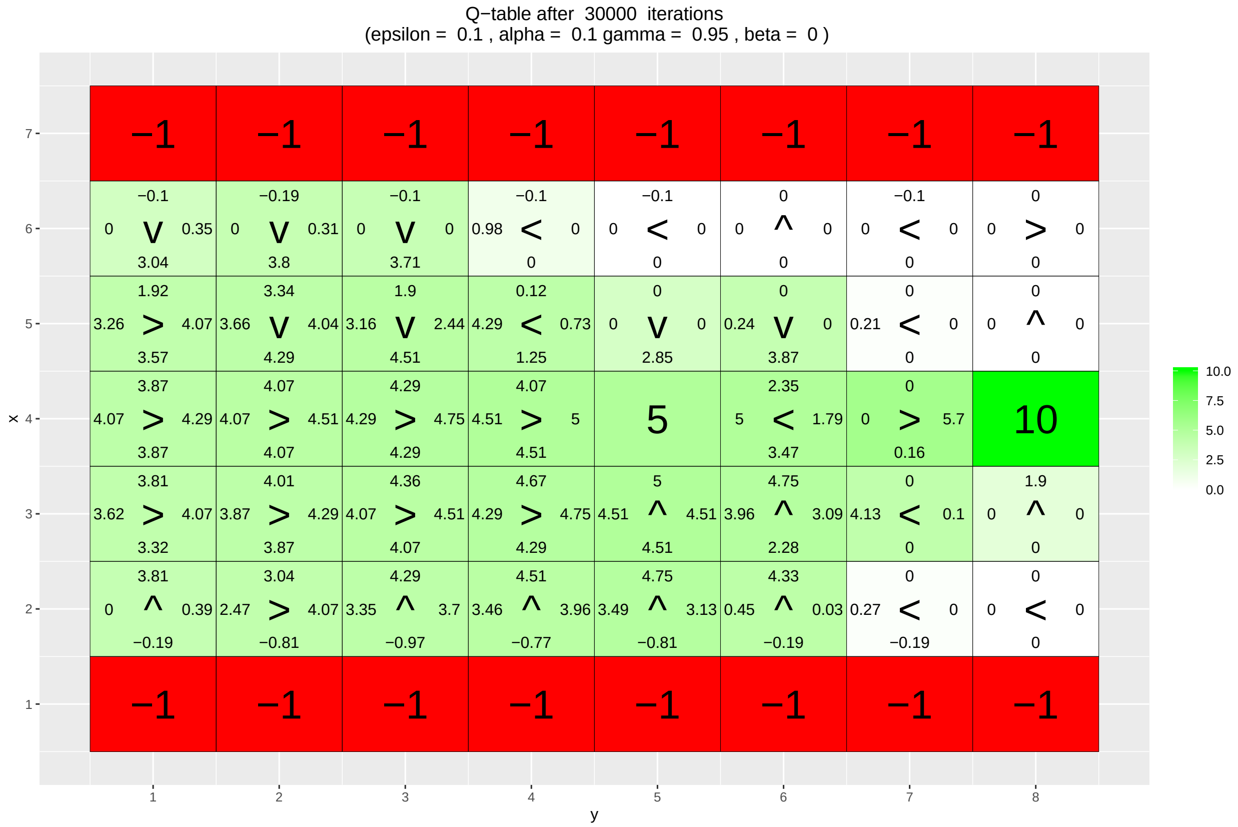

MovingAverage(reward, 100)
5.0
4.8
4.6
4.4
4.2
0
5000
10000
15000
20000
25000
30000
Index

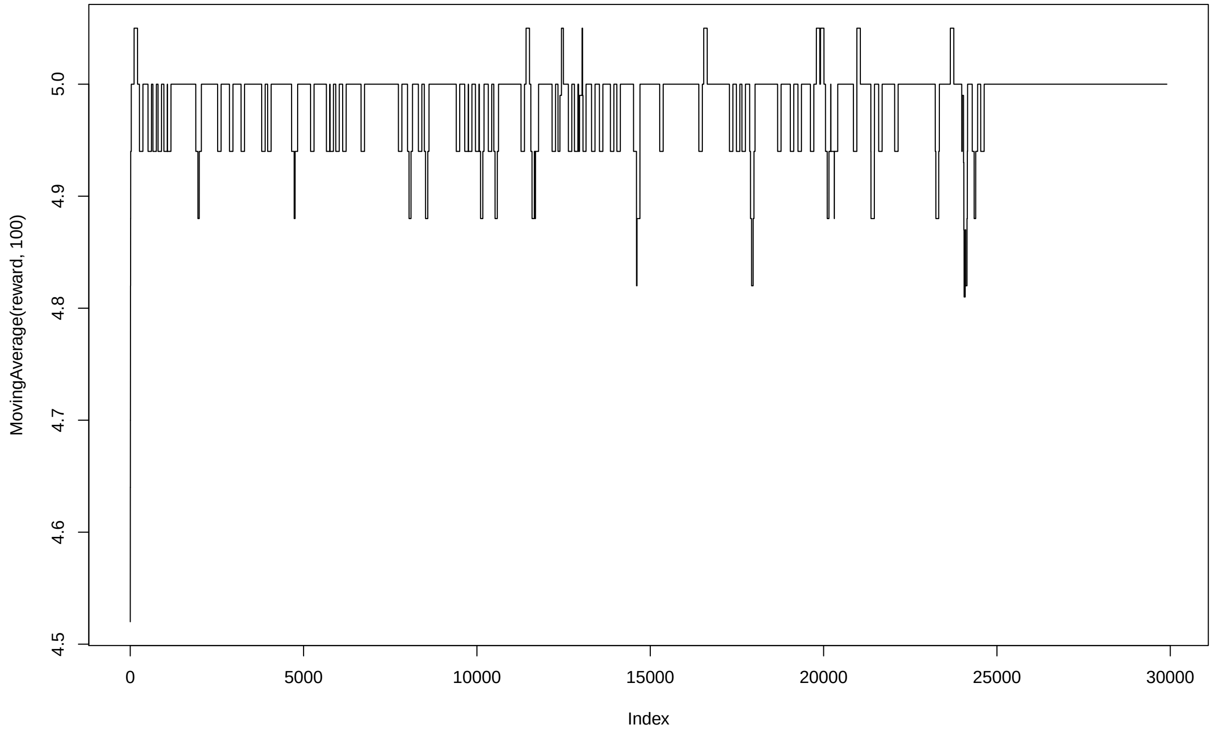

MovingAverage(correction, 100)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
5000
10000
15000
20000
25000
30000
Index

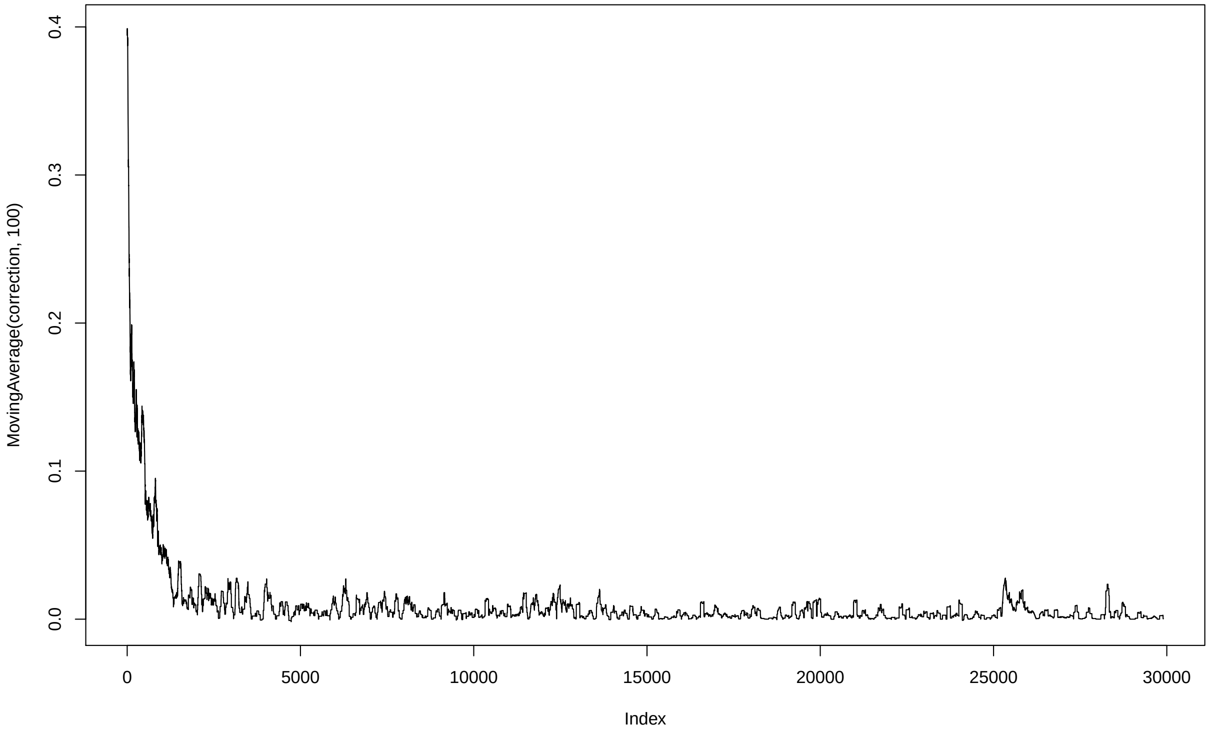

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.75 , beta = 0)
7
−1 −1 −1 −1 −1 −1 −1 −1
6
−0.1 0.06 V 0 1.06
−0.19 0 > 1.31 0.16
−0.1 0.13 V 0 2.09
−0.1 0 V 0 2.65
−0.1 0.36 < 0 0
0 0 > 0 0
0 0 < 0 0
0 0 < 0 0
5
0.57 1 V 1.49 1.58
0.66 1.16 V 1.76 2.11
1.33 1.58 V 2.5 2.81
1.58 1.97 V 2.45 3.75
0.01 0.22 V 0.01 4.39
0 0.28 < 0 0
0 0 > 0 0
0 0 ^ 0 0
x 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 0 > 0 0
0 0 < 0 0
10
3
1.58 1.06 ^ 1.44 0.42
2.11 1.04 ^ 1.76 0.89
2.56 1.46 > 2.81 1.28
3.73 2.1 > 3.75 2.09
5 2.8 ^ 2.66 2.76
0 3.71 < 0 0
0 0 < 0 0
1 0 ^ 0 0
2
0.97 0 ^ 0 −0.1
1.47 0.07 ^ 0 −0.19
2.01 0 ^ 0 −0.19
2.81 0 ^ 0.48 −0.34
3.74 0.21 ^ 0 −0.19
0 0.15 < 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
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
4.5
4.6
4.7
4.8
4.9
5.0
0
5000
10000
15000
20000
25000
30000
Index


MovingAverage(correction, 100)
0.00
0.05
0.10
0.15
0.20
0.25
0
5000
10000
15000
20000
25000
30000
Index


Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.95 , beta = 0)
−1 −1 −1 −1 −1 −1 −1 −1
−0.1 0 V 0.35 3.04
−0.19 0 V 0.31 3.8
−0.1 0 V 0 3.71
−0.1 0.98 < 0 0
−0.1 0 < 0 0
0 0 ^ 0 0
−0.1 0 < 0 0
0 0 > 0 0
1.92 3.26 > 4.07 3.57
3.34 3.66 V 4.04 4.29
1.9 3.16 V 2.44 4.51
0.12 4.29 < 0.73 1.25
0 0 V 0 2.85
0 0.24 V 0 3.87
0 0.21 < 0 0
0 0 ^ 0 0
3.87 4.07 > 4.29 3.87
4.07 4.07 > 4.51 4.07
4.29 4.29 > 4.75 4.29
4.07 4.51 > 5 4.51
5
2.35 5 < 1.79 3.47
0 0 > 5.7 0.16
10
3.81 3.62 > 4.07 3.32
4.01 3.87 > 4.29 3.87
4.36 4.07 > 4.51 4.07
4.67 4.29 > 4.75 4.29
5 4.51 ^ 4.51 4.51
4.75 3.96 ^ 3.09 2.28
0 4.13 < 0.1 0
1.9 0 ^ 0 0
3.81 0 ^ 0.39 −0.19
3.04 2.47 > 4.07 −0.81
4.29 3.35 ^ 3.7 −0.97
4.51 3.46 ^ 3.96 −0.77
4.75 3.49 ^ 3.13 −0.81
4.33 0.45 ^ 0.03 −0.19
0 0.27 < 0 −0.19
0 0 < 0 0
−1 −1 −1 −1 −1 −1 −1 −1
x
7
6
5
4
3
2
1
y
1 2 3 4 5 6 7 8
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
4.5
4.6
4.7
4.8
4.9
5.0
0
5000
10000
15000
20000
25000
30000
Index



For Epsilon 0.5 and variants of Gamma to 0.5,0.75 and 0.95.

Epsilon = 0.1, Gamma = 0.5 => In this variant, with such less epsilon value, the agent just moves along the maximized action towards the nearby terminal state and settles there by not exploring other non terminal states.

Epsilon = 0.1, Gamma = 0.75 => In this variant, since there is slight improvement in gamma, it looks to get as much as instant rewards by trying alternative route but again settles at 5 terminal state.

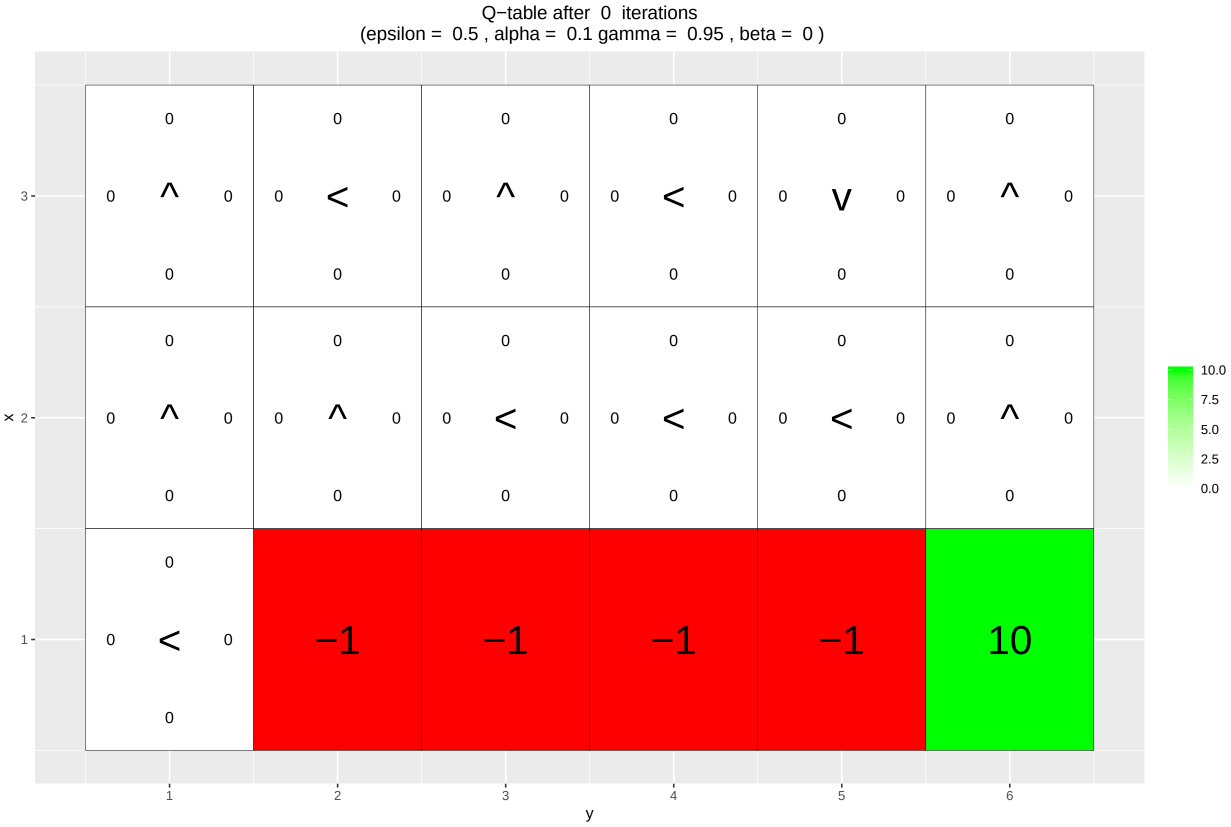
Epsilon = 0.1, Gamma = 0.95 => The agent is looking for long term reward since gamma is close to 1. In this case, since the epsilon is very low meaning does not look for alternative routes, hardly it finds a route to maximize the long term reward to 10 terminal state.

With more epsilon value more directions are explored and with less epsilon the alternatives routes are not explored much. The gamma variants lower to higher looks for maximizing the intial rewards and settling at nearest terminal state until it finds other terminal state with higher reward.

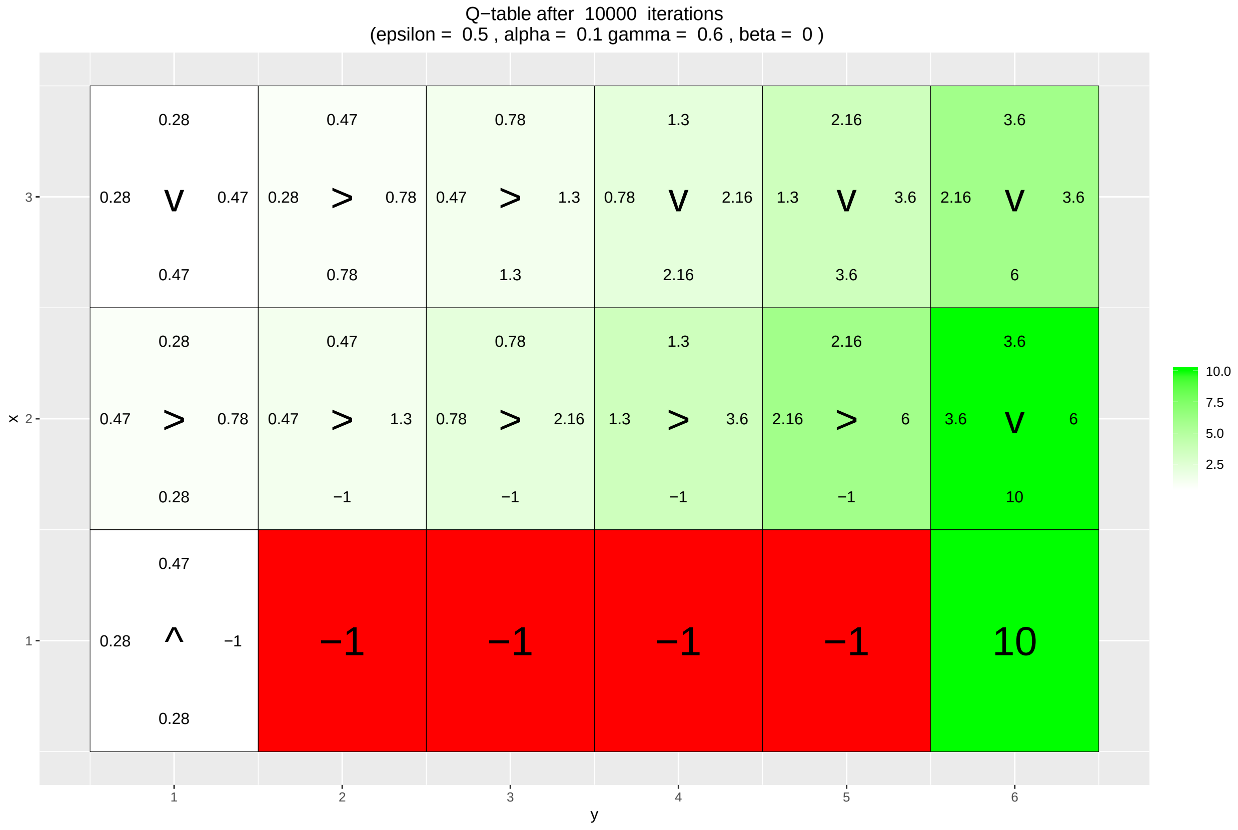
ENV C

```
# 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))
```

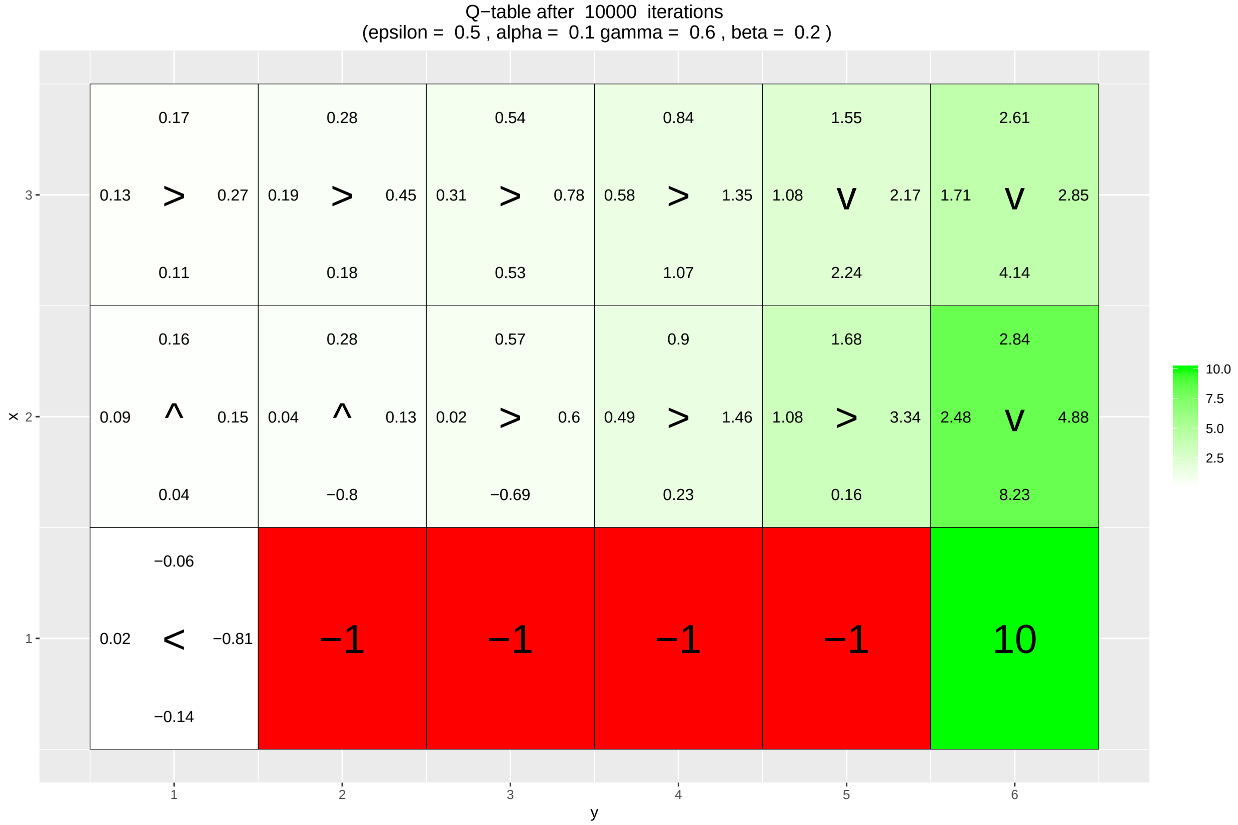
```
vis_environment()
```



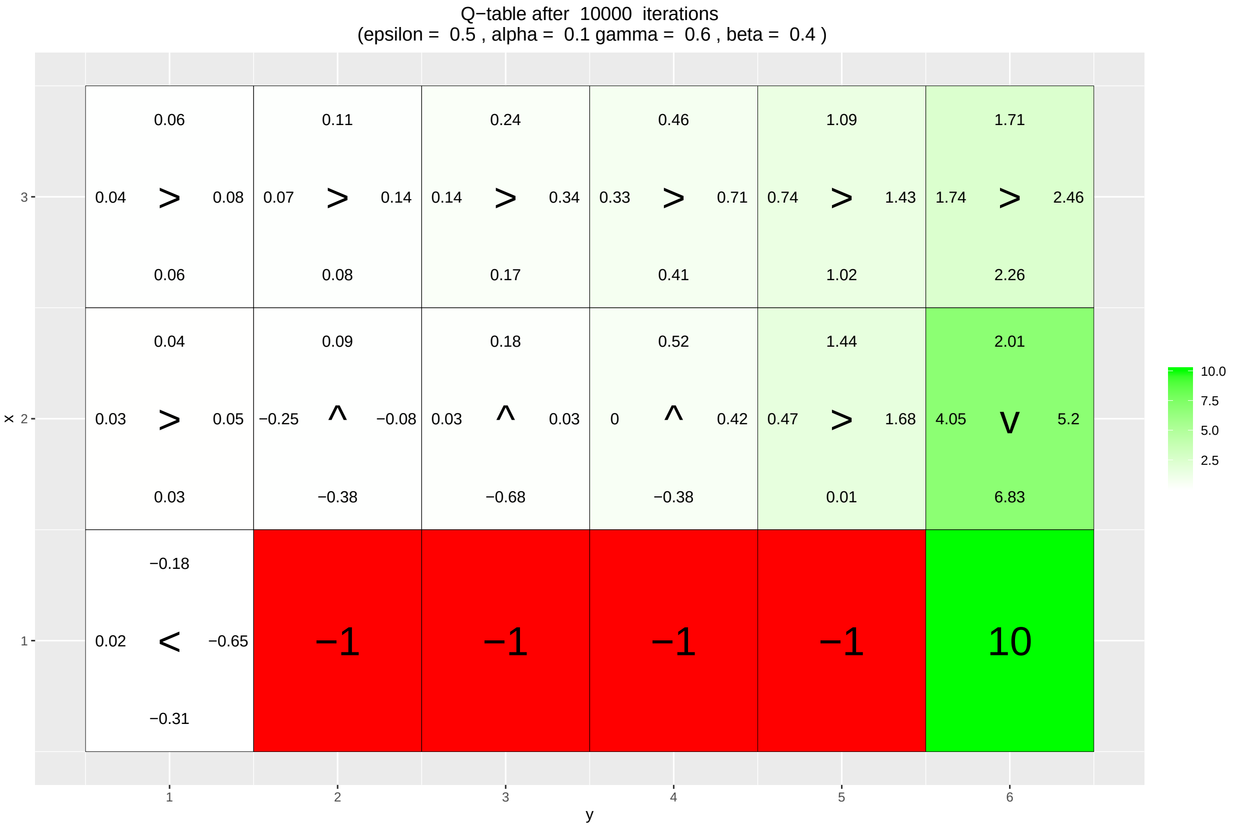
```
for(j in c(0,0.2,0.4,0.66)){
  q_table <- array(0,dim = c(H,W,4))

  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)
}
```



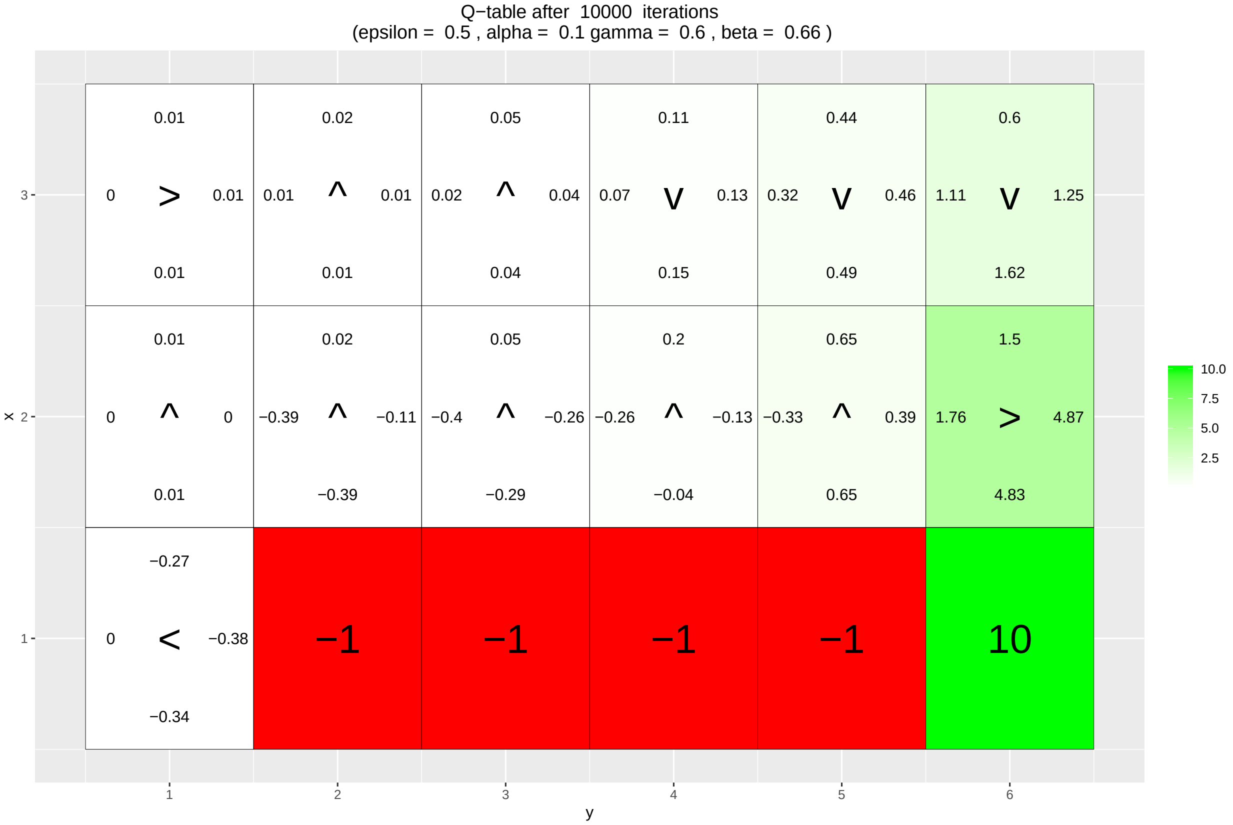
Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0)



Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.2)



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)

For Epsilon = 0.5, Gamma = 0.6 and alpha = 0.1 and different variants of beta = (0,0.2,0.4,0.66).

Beta the probability of agent moving in intended direction. If beta = 0 then robot is moving in intended direction 100% # Amongst all 4 variants of beta, beta with 0.4 resulted in descent result since the agent tried exploring other directions with almost equal probability of intended direction. The beta variants moved either with too less probability in intended direction or too more.

ENV D

```
library(keras)
library(tensorflow) # You may also need to run this if you don't have an existing tensorflow installati
#install_tensorflow()
# library(keras)

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

# If you do not see four arrows in line 19, then do the following:
# File/Reopen with Encoding/UTF-8

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

vis_prob <- function(goal, episodes = 0){

  # Visualize an environment with rewards.
  # Probabilities for all actions are displayed on the edges of each tile.
  # The (greedy) policy for each state is also displayed.
  #
  # Args:
  #   goal: goal coordinates, array with 2 entries.
  #   episodes, epsilon, alpha, gamma, beta (optional): for the figure title.
  #   H, W (global variables): environment dimensions.

  df <- expand.grid(x=1:H,y=1:W)
  dist <- array(data = NA, dim = c(H,W,4))
  class <- array(data = NA, dim = c(H,W))
  for(i in 1:H)
    for(j in 1:W){
      dist[i,j,] <- DeepPolicy_dist(i,j,goal[1],goal[2])
      foo <- which(dist[i,j,]==max(dist[i,j,]))
      class[i,j] <- ifelse(length(foo)>1,sample(foo, size = 1),foo)
    }

  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,1]),df$x,df$y)
  df$val1 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,2]),df$x,df$y)
  df$val2 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,3]),df$x,df$y)
  df$val3 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,4]),df$x,df$y)
  df$val4 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,class[x,y]),df$x,df$y)
  df$val5 <- as.vector(arrows[foo])
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),"Goal",NA),df$x,df$y)
  df$val6 <- as.vector(foo)

  print(ggplot(df,aes(x = y,y = x)) +
          geom_tile(fill = 'white', colour = 'black') +
          scale_fill_manual(values = c('green')) +
          geom_tile(aes(fill=val6), show.legend = FALSE, colour = 'black') +
          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,na.rm = TRUE) +
          geom_text(aes(label = val6),size = 10,na.rm = TRUE) +
          ggtitle(paste("Action probabilities after ",episodes," episodes")) +
          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)
}

DeepPolicy_dist <- function(x, y, goal_x, goal_y){

  # Get distribution over actions for state (x,y) and goal (goal_x,goal_y) from the deep policy.
  #
  # Args:
  #   x, y: state coordinates.
  #   goal_x, goal_y: goal coordinates.
  #   model (global variable): NN encoding the policy.
  #
  # Returns:
  #   A distribution over actions.

  foo <- matrix(data = c(x,y,goal_x,goal_y), nrow = 1)

  # return (predict_proba(model, x = foo))
  return (predict_on_batch(model, x = foo)) # Faster.

}

DeepPolicy <- function(x, y, goal_x, goal_y){

  # Get an action for state (x,y) and goal (goal_x,goal_y) from the deep policy.
  #
  # Args:
  #   x, y: state coordinates.
  #   goal_x, goal_y: goal coordinates.
  #   model (global variable): NN encoding the policy.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  foo <- DeepPolicy_dist(x,y,goal_x,goal_y)

  return (sample(1:4, size = 1, prob = foo))

}

DeepPolicy_train <- function(states, actions, goal, gamma){

  # Train the policy network on a rolled out trajectory.
  #
  # Args:
  #   states: array of states visited throughout the trajectory.
  #   actions: array of actions taken throughout the trajectory.
  #   goal: goal coordinates, array with 2 entries.
  #   gamma: discount factor.

  # Construct batch for training.
  inputs <- matrix(data = states, ncol = 2, byrow = TRUE)
  inputs <- cbind(inputs,rep(goal[1],nrow(inputs)))
  inputs <- cbind(inputs,rep(goal[2],nrow(inputs)))

  targets <- array(data = actions, dim = nrow(inputs))
  targets <- to_categorical(targets-1, num_classes = 4)

  # Sample weights. Reward of 5 for reaching the goal.
  weights <- array(data = 5*(gamma^(nrow(inputs)-1)), dim = nrow(inputs))

  # Train on batch. Note that this runs a SINGLE gradient update.
  train_on_batch(model, x = inputs, y = targets, sample_weight = weights)

}
```

```
reinforce_episode <- function(goal, gamma = 0.95, beta = 0){

  # Rolls out a trajectory in the environment until the goal is reached.
  # Then trains the policy using the collected states, actions and rewards.
  #
  # Args:
  #   goal: goal coordinates, array with 2 entries.
  #   gamma (optional): discount factor.
  #   beta (optional): probability of slipping in the transition model.

  # Randomize starting position.
  cur_pos <- goal
  while(all(cur_pos == goal))
    cur_pos <- c(sample(1:H, size = 1),sample(1:W, size = 1))

  states <- NULL
  actions <- NULL

  steps <- 0 # To avoid getting stuck and/or training on unnecessarily long episodes.
  while(steps < 20){
    steps <- steps+1

    # Follow policy and execute action.
    action <- DeepPolicy(cur_pos[1], cur_pos[2], goal[1], goal[2])
    new_pos <- transition_model(cur_pos[1], cur_pos[2], action, beta)

    # Store states and actions.
    states <- c(states,cur_pos)
    actions <- c(actions,action)
    cur_pos <- new_pos

    if(all(new_pos == goal)){
      # Train network.
      DeepPolicy_train(states,actions,goal,gamma)
      break
    }
  }

}
```

Yes, the agent has learned a policy, irrespective of change in the Goal positions at every episode, at validation time, the agent sucessfully reaches the goal state by maximizing the action at each step and also trying alternative routes to the goal state. # If Qlearning algorithm was used to run this task, it would have been expensive in terms of storage and also end up in poor performace. The reason is Qlearning cannot handle continous state and action spaces. Since at each episode we are randomizing the terminal state, for Q learning it would enormous memory space and also would end up non optimal policy for the agent. In other words why Q learning cannot be used is, the algorithm does not GENERALIZES well in terms of landing at terminal state REINFORCE does.

ENV E

```
train_goals <- list(c(4,1), c(4,2), c(4,3), c(4,4))
val_goals <- list(c(3,4), c(2,3), c(1,1))
```

```r
set_weights(model,initial_weights)

show_validation(0)

for(i in 1:5000){
  if(i%%10==0) cat("episode", i,"\n")
  goal <- sample(train_goals, size = 1)
  reinforce_episode(unlist(goal))
}

show_validation(5000)
```

In this environment, the agent is trained only on top row of the grid. One of the objective of using REINFORCE algorithm is that agent is trained on Random Goal state at each Episode so that during test time the agent GENERALIZES the procedure of landing at Goal state irrespective of position of the Goal state.

Since it is trained at random positions only in top row, the agent fails to learn optimal policy in landing at Goal state. Although the agent tries to land in Goal state accidentally due to random nature of the algorithm but it has not learnt a good policy overall.

The reason why the results ENV D an ENV E differs is in ENV E the algorithm is trained on Random Goal state without any constraint so that the algorithm during test time it GENERALIZES the action and ends up in Goal state. Where as in ENV E there is a constraint that the algorithm is only trained on top row, the agent cannot GENERALIZE the actions and out of randomness hardly it might land in GOAL state accidentally.

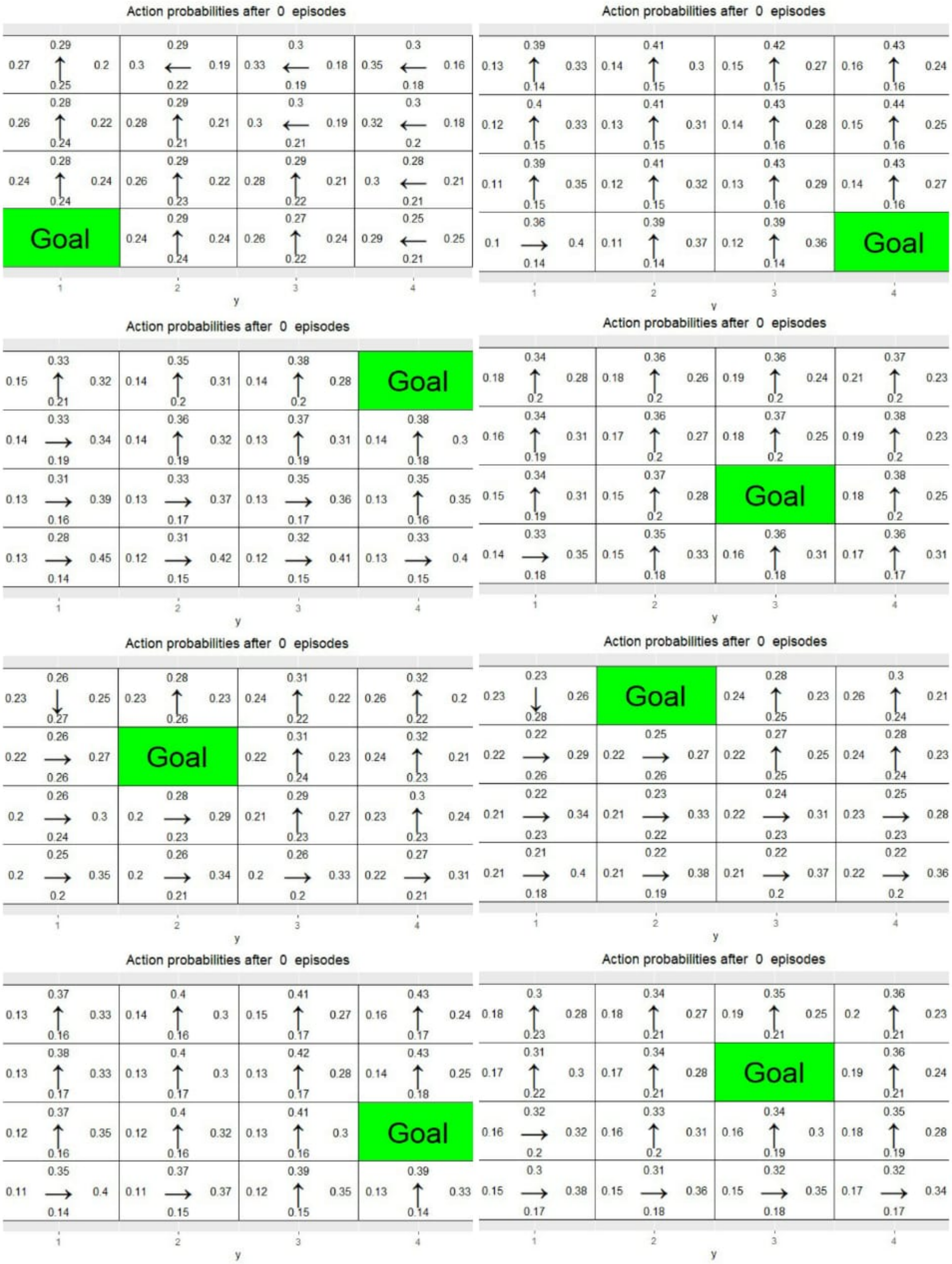


Figure 1: "ENV_D_TRAINING"

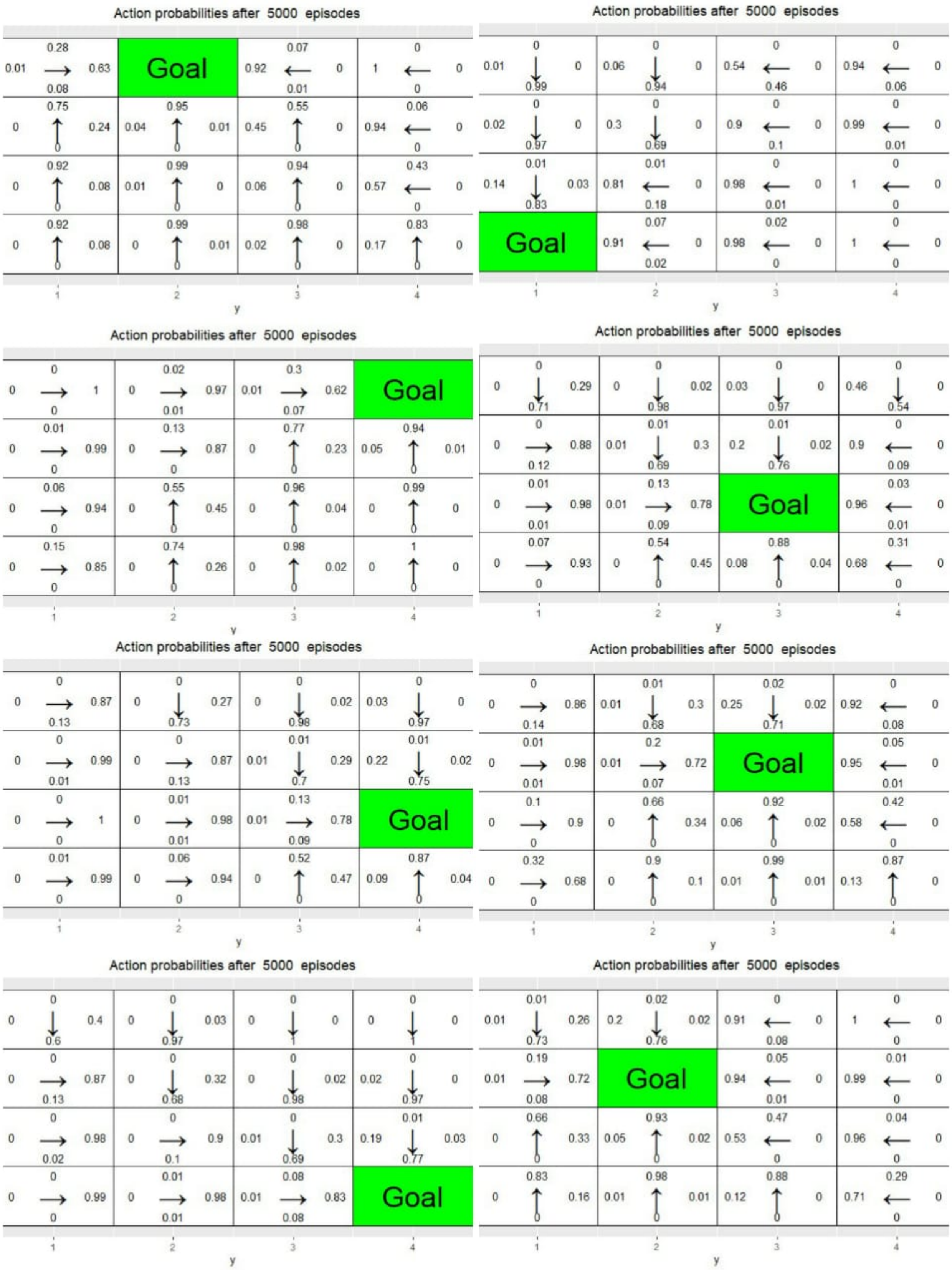


Figure 2: "ENV_D_TEST"

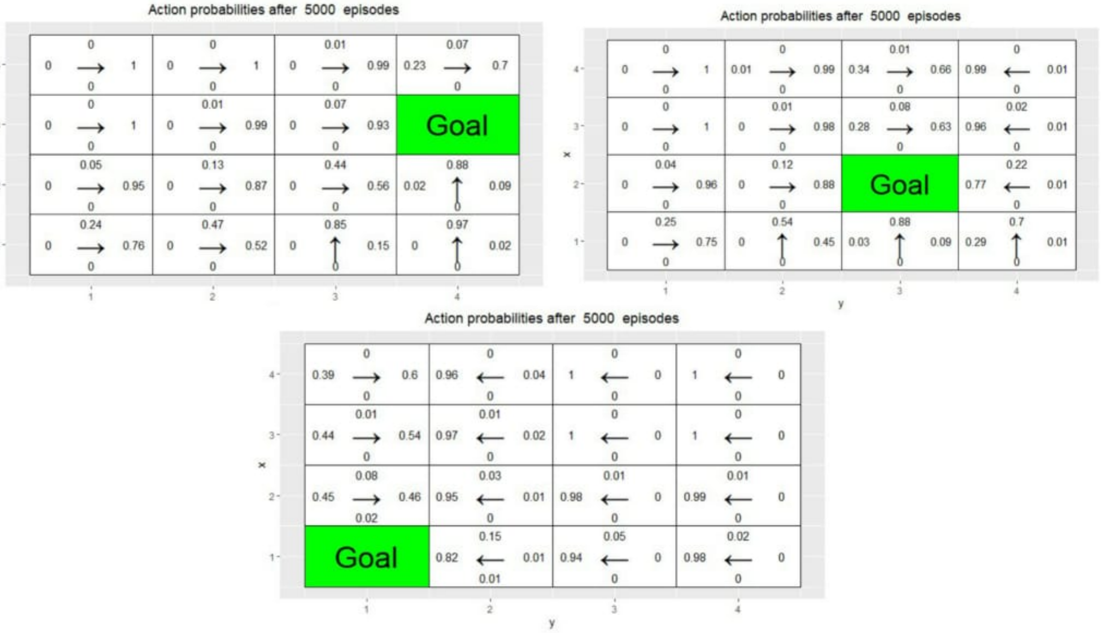


Figure 3: “ENV_E”