Lab3

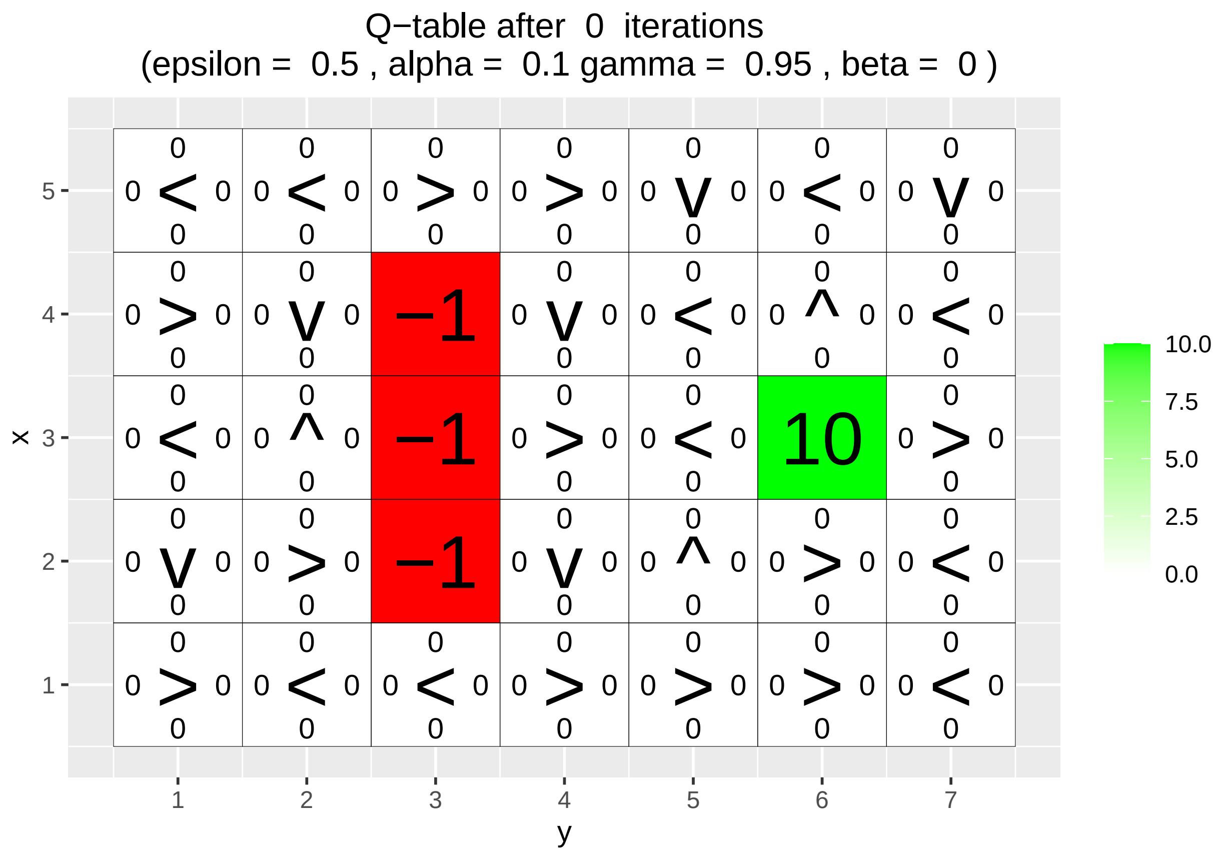
Christian von Koch

2020-10-08

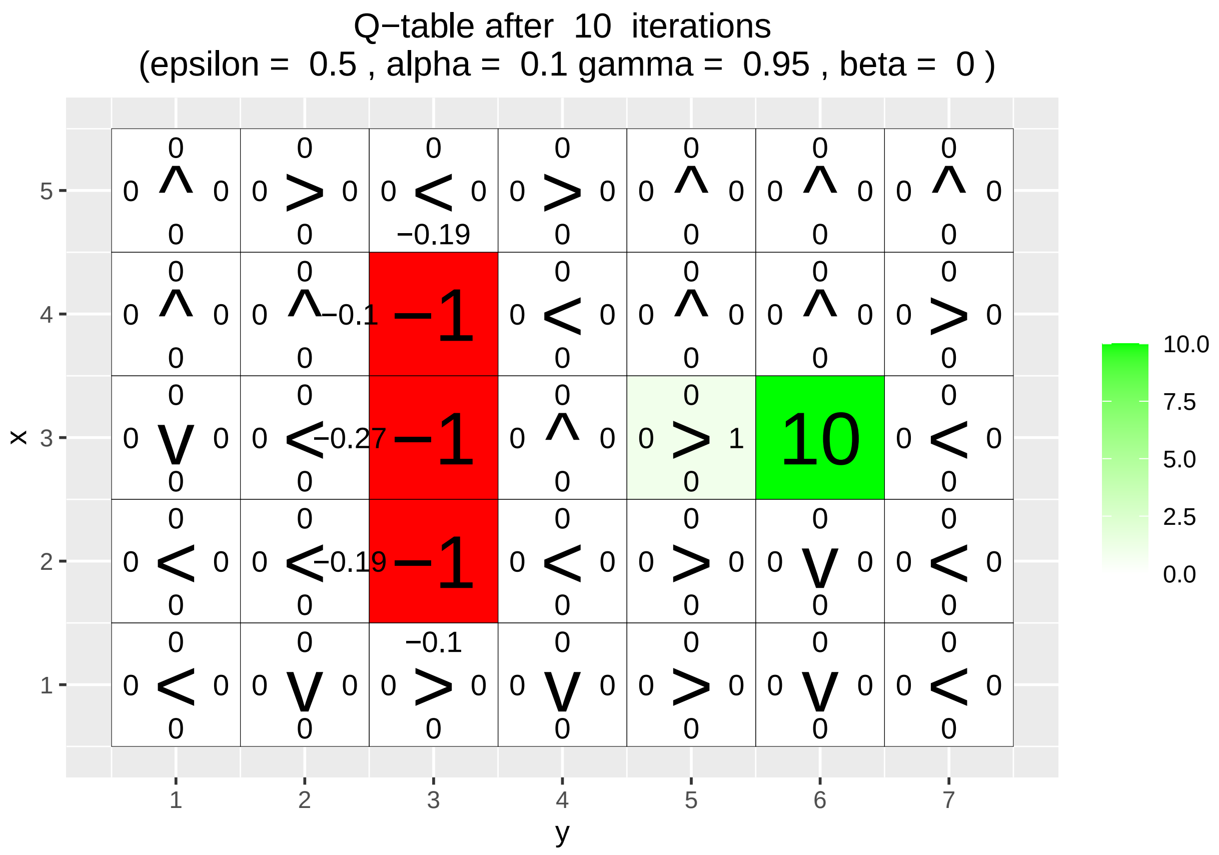
Assignment Q-learning Environment A

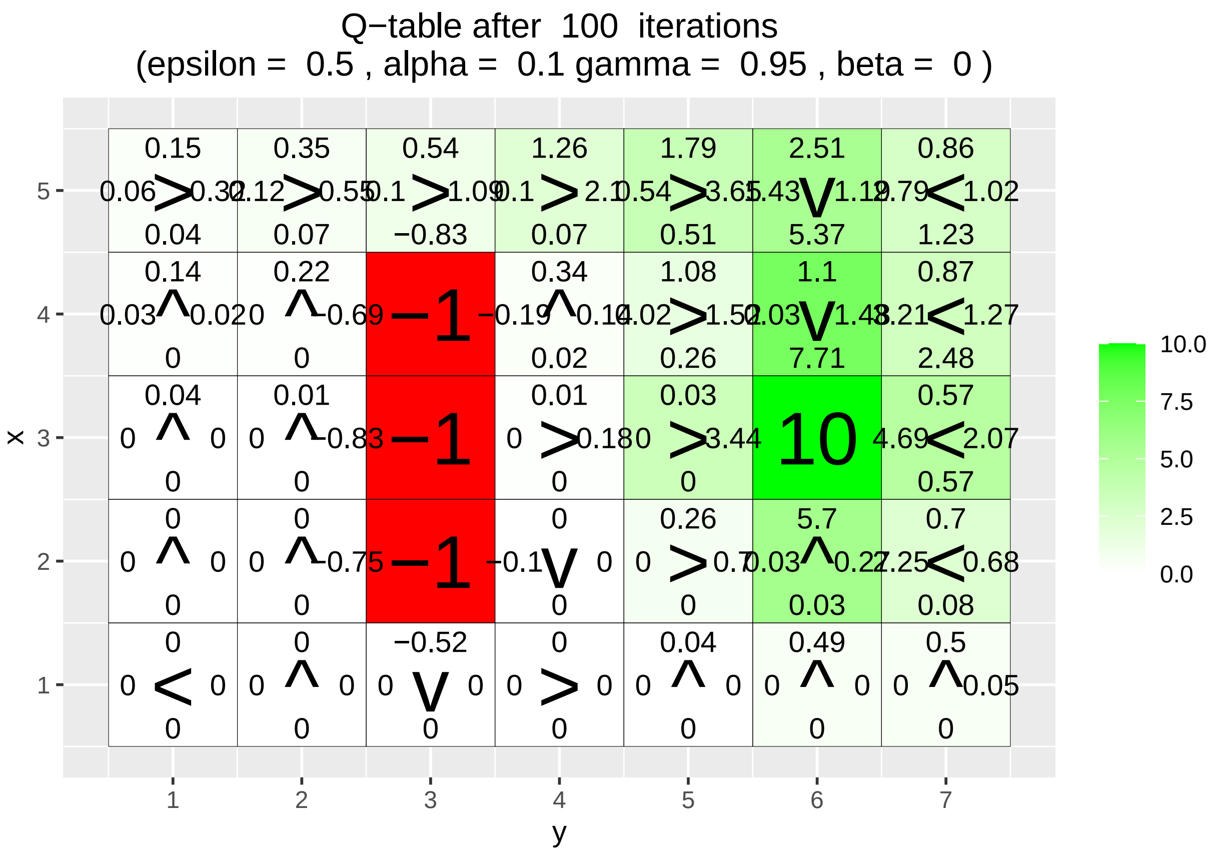
```r
GreedyPolicy <- function(x, y){

  # 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}.

  movement1 = which(q_table[x,y,] == max(q_table[x,y,]))
  if(length(movement1)>1) {
    return(sample(movement1, 1))
  } else {
    return(movement1)
  }
}

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}.

  movement1 = sample(which(q_table[x,y,] == max(q_table[x,y,])), 1)
  draw=runif(1, min=0, max=1)
  if(draw>epsilon) {
    return(movement1)
  } else {
    return(sample(c(1,2,3,4), 1))
  }

}
```

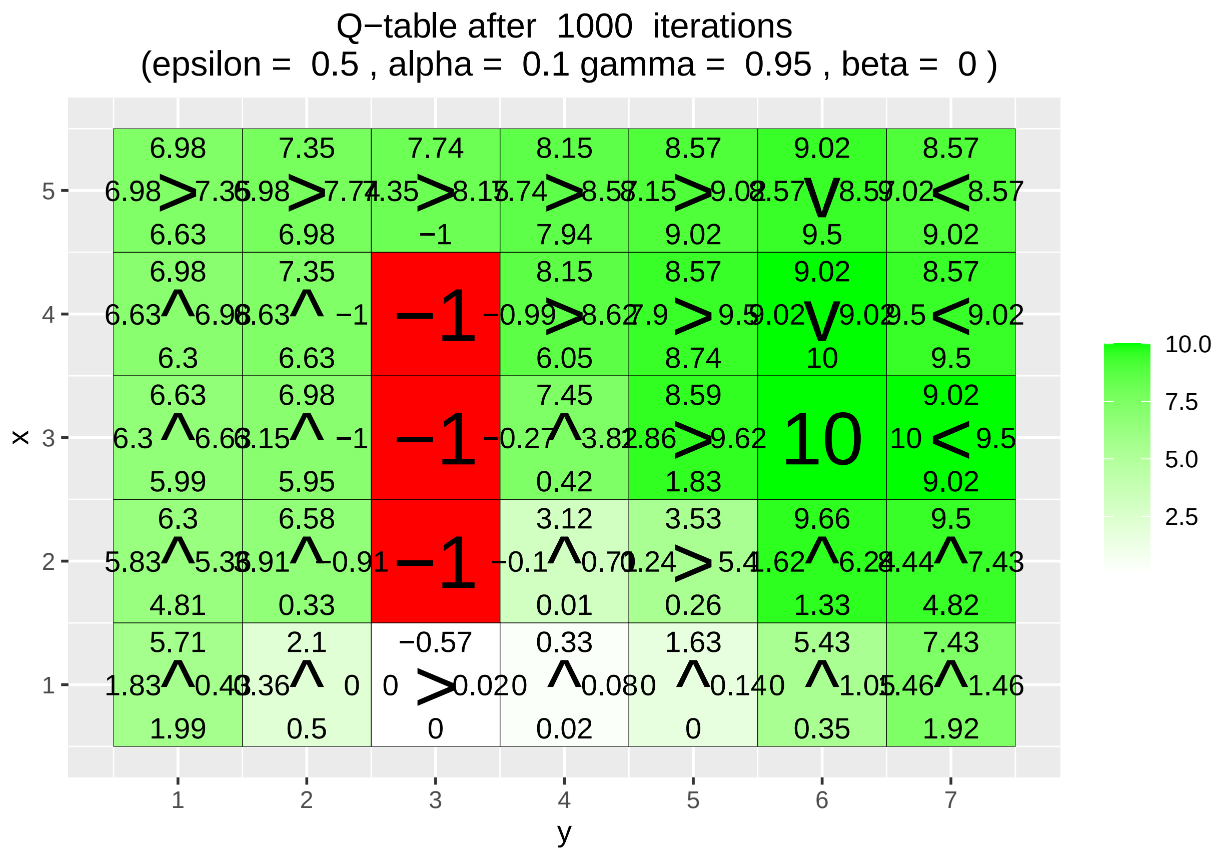
```
transition_model <- function(x, y, action, beta){

  # 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 <<-.

  q_state=start_state
  reward=0
  episode_correction=0

  repeat{
    # Follow policy, execute action, get reward.
    x=q_state[1]
    y=q_state[2]
    action=EpsilonGreedyPolicy(x, y, epsilon)
    new_state=transition_model(x, y, action, beta)
    reward=reward_map[new_state[1], new_state[2]]

    # Q-table update.
    episode_correction=episode_correction+reward+gamma*max(q_table[new_state[1], new_state[2],])-q_tabl
    correction=alpha*(reward+gamma*max(q_table[new_state[1], new_state[2],])-q_table[x,y,action])
    q_table[x,y,action]<<-q_table[x,y,action]+correction
    q_state=new_state

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

}

# Environment A (learning)

set.seed(12345)
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()
```

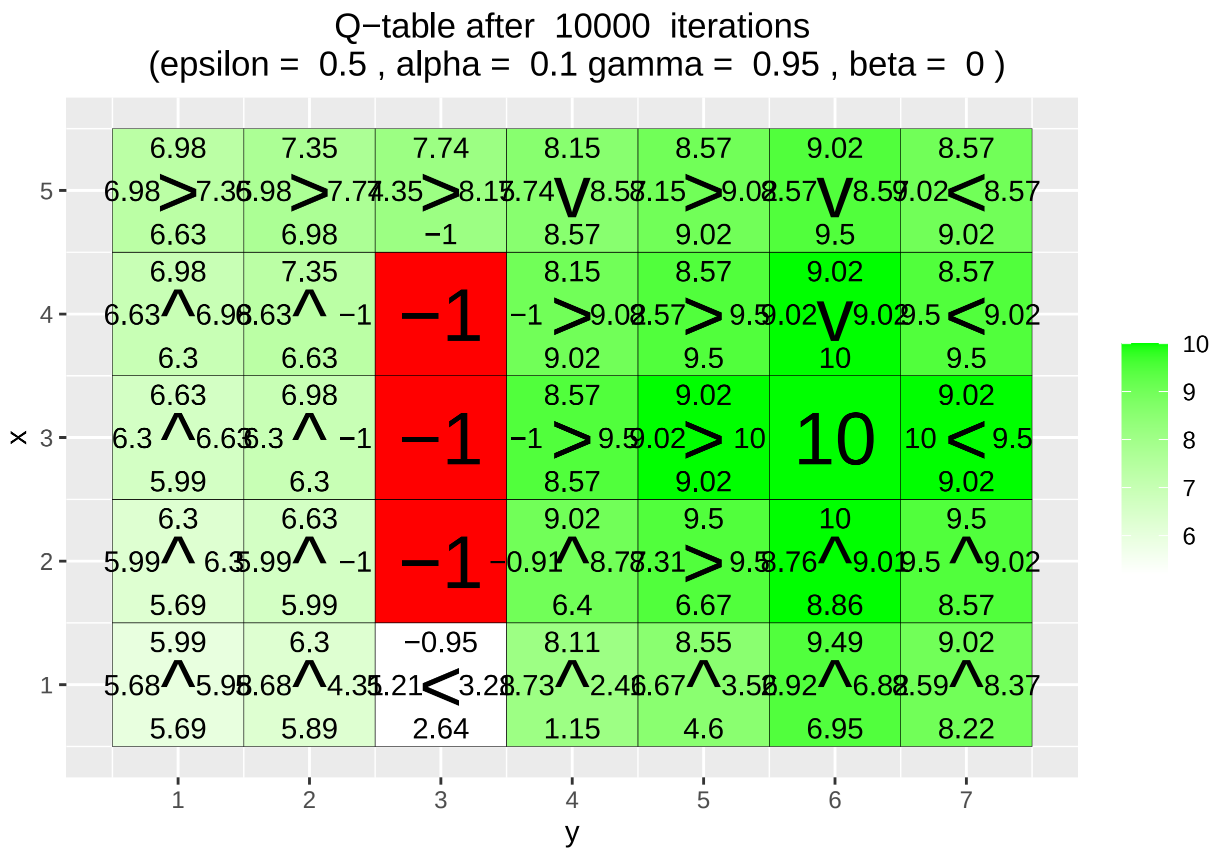


```
for(i in 1:10000){
  foo <- q_learning(start_state = c(3,1))

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


Q−table after 10 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
1
2
3
4
5
6
7
10.0
7.5
5.0
2.5
0.0
−0.19
−0.1
−1
−0.27
1
10
−0.19
−1
−1
−0.1


Q−table after 100 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


Q−table after 1000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
1
2
3
4
5
6
7
10.0
7.5
5.0
2.5
−1
10



After 10 iterations the agent has to some extent learned to avoid the negative rewards as well as to catch the positive reward when the agent is at a state directly adjacent to the rewards. Since the agent has only had 10 episodes to learn it has not explored all of the states and has therefore not learned what the optimal action is in all of the states.

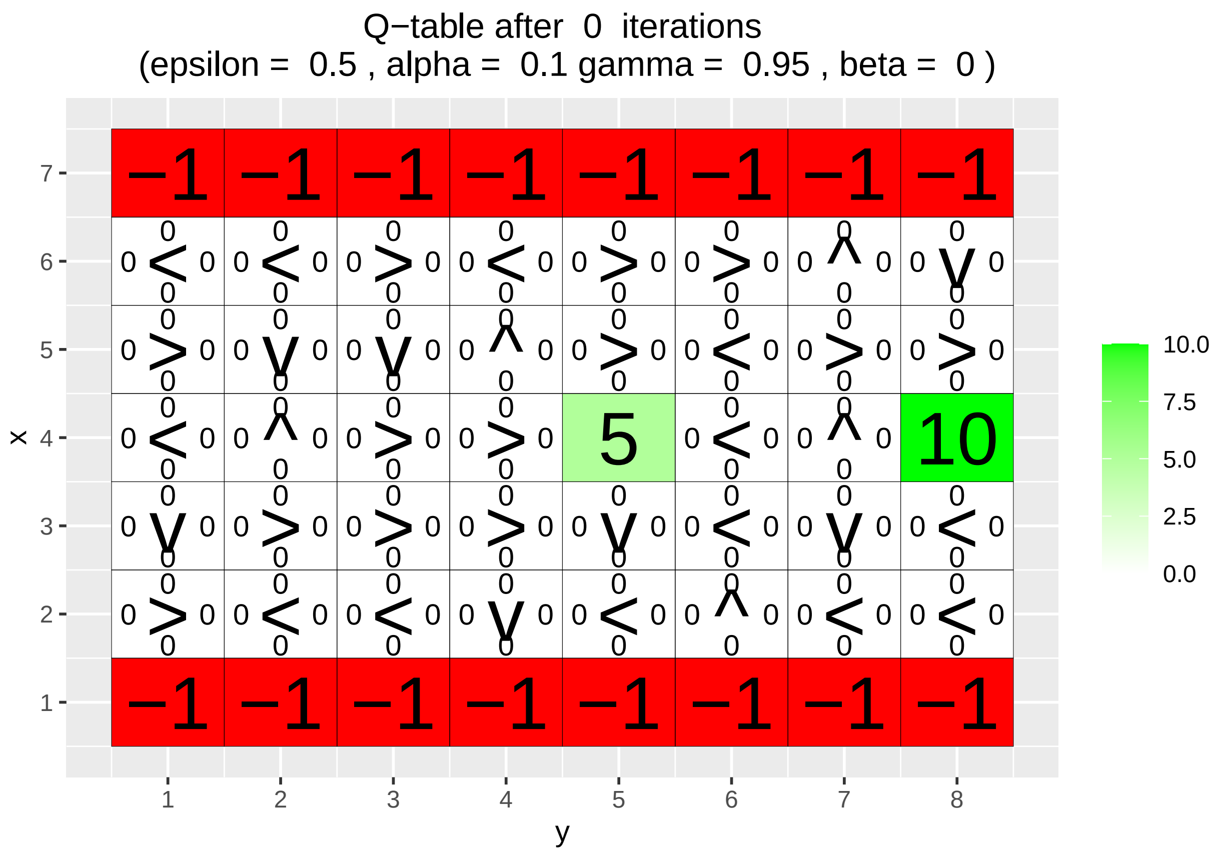
The final greedy policy after 10000 iteration is not optimal since it is evident from the output of the Q-table that all of the states doesn't suggest high rewards. For example the state $(1, 3)$ has a very low reward in comparison to the other states which can be observed through the white color of that particular state. This further suggests that this particular state might have been less explored than the other states and therefore its reward has not been adjusted. If no optimal action points towards a specific state it is also less likely that this state will be explored - this is something that can be confirmed by the state $(1, 3)$ since no arrow points towards this state. The only way for this state to be explored is therefore if the agent chooses a set of actions randomely from the initial state and through this reaches the particular state - an event which is very unlikely.

The agent has not fully learned that there are multiple paths to get to the positive reward. To increase the chance of the agent learning this, one might set a higher value of ϵ to increase the explore rate or set a higher value of γ to weight the future rewards more than just the present one.

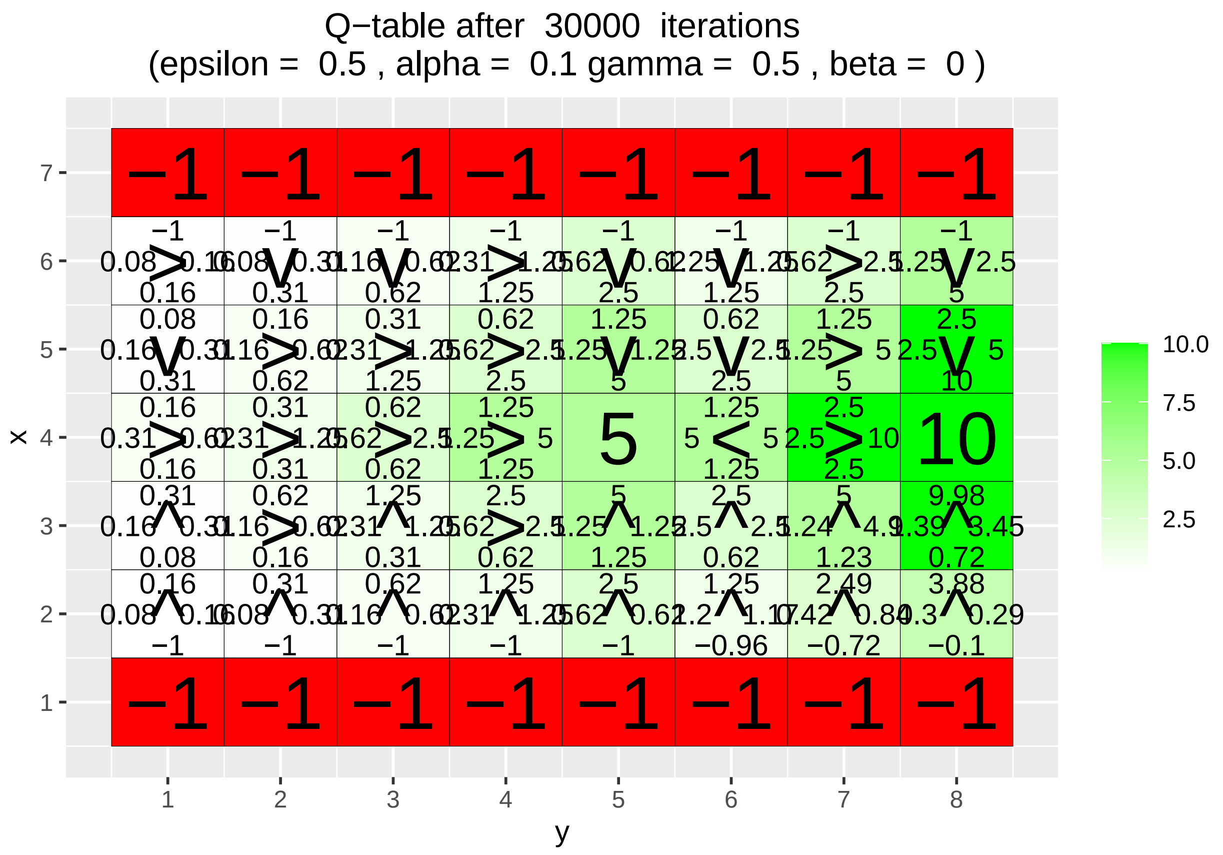
Assignment Q-learning Environment B

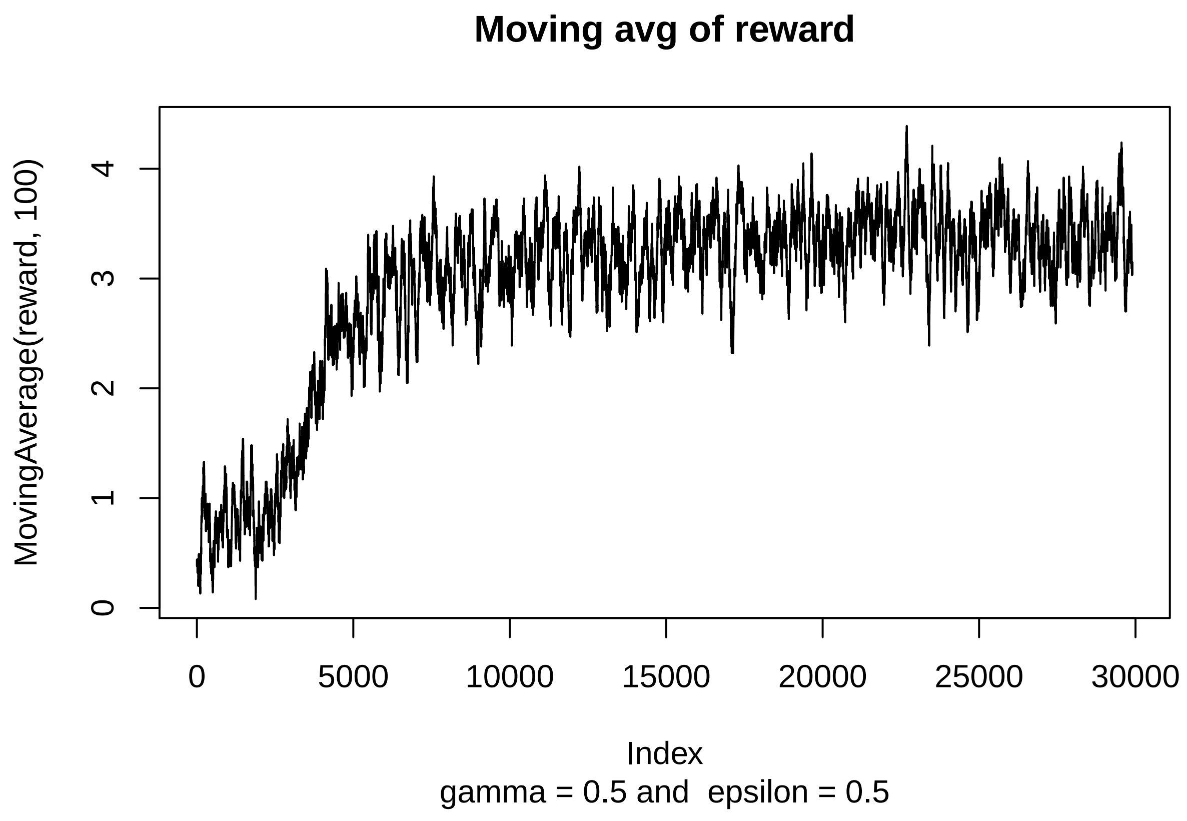
```r
# Environment B (the effect of epsilon and gamma)
set.seed(12345)

H <- 7
W <- 8
```

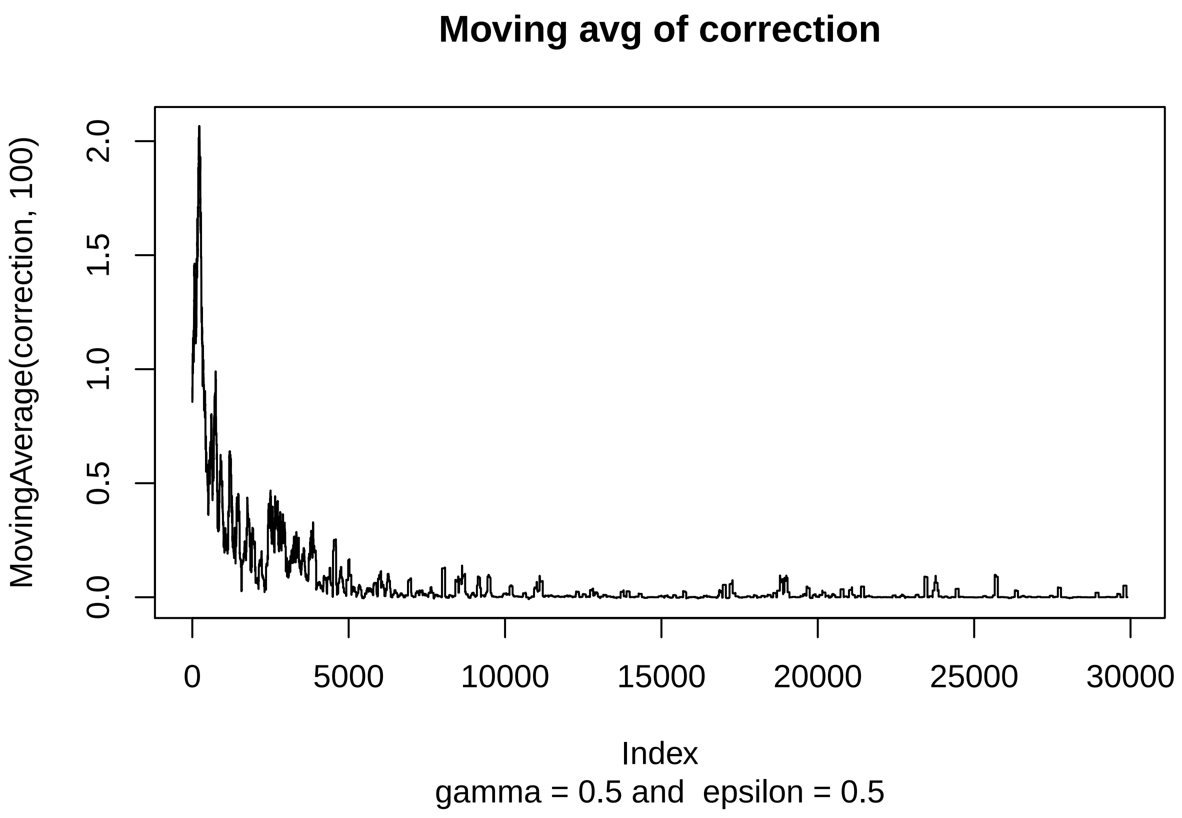
```r
reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,] <- -1
reward_map[7,] <- -1
reward_map[4,5] <- 5
reward_map[4,8] <- 10

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

vis_environment()
```

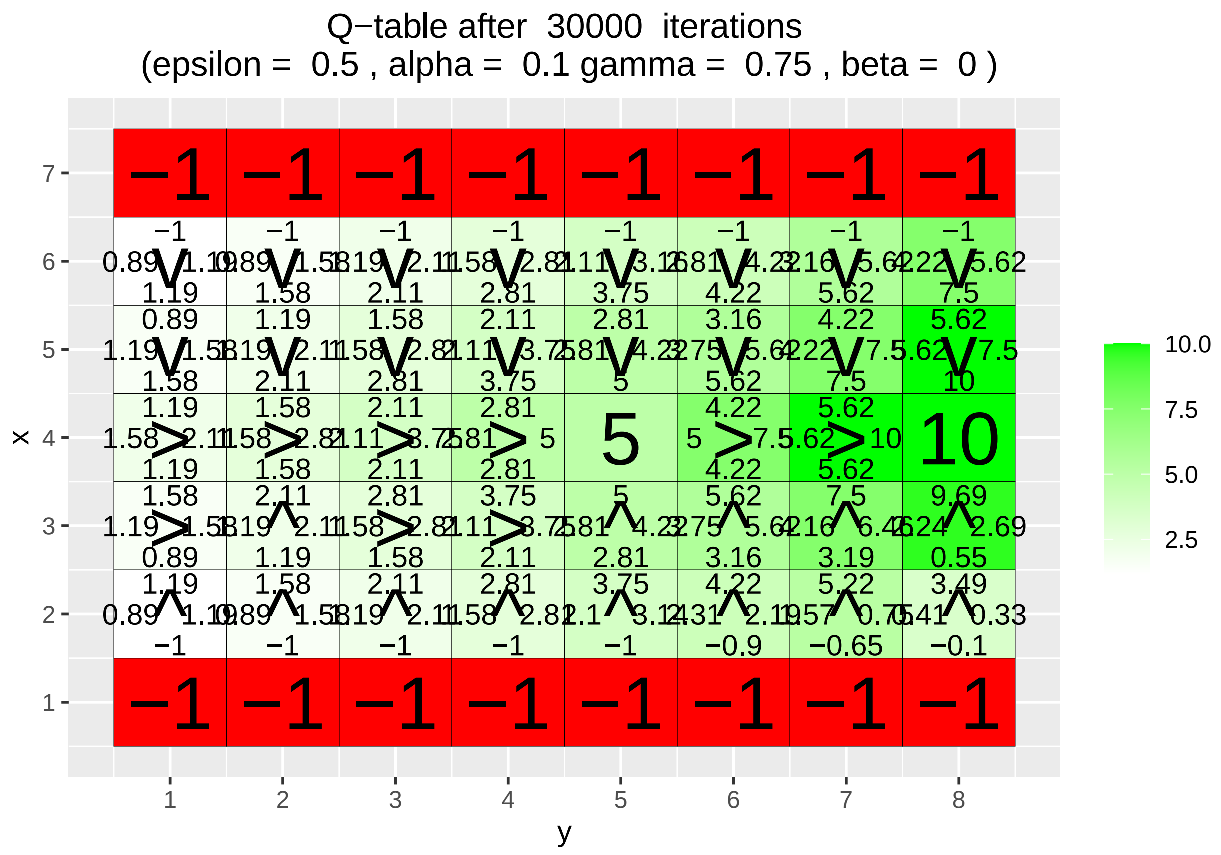


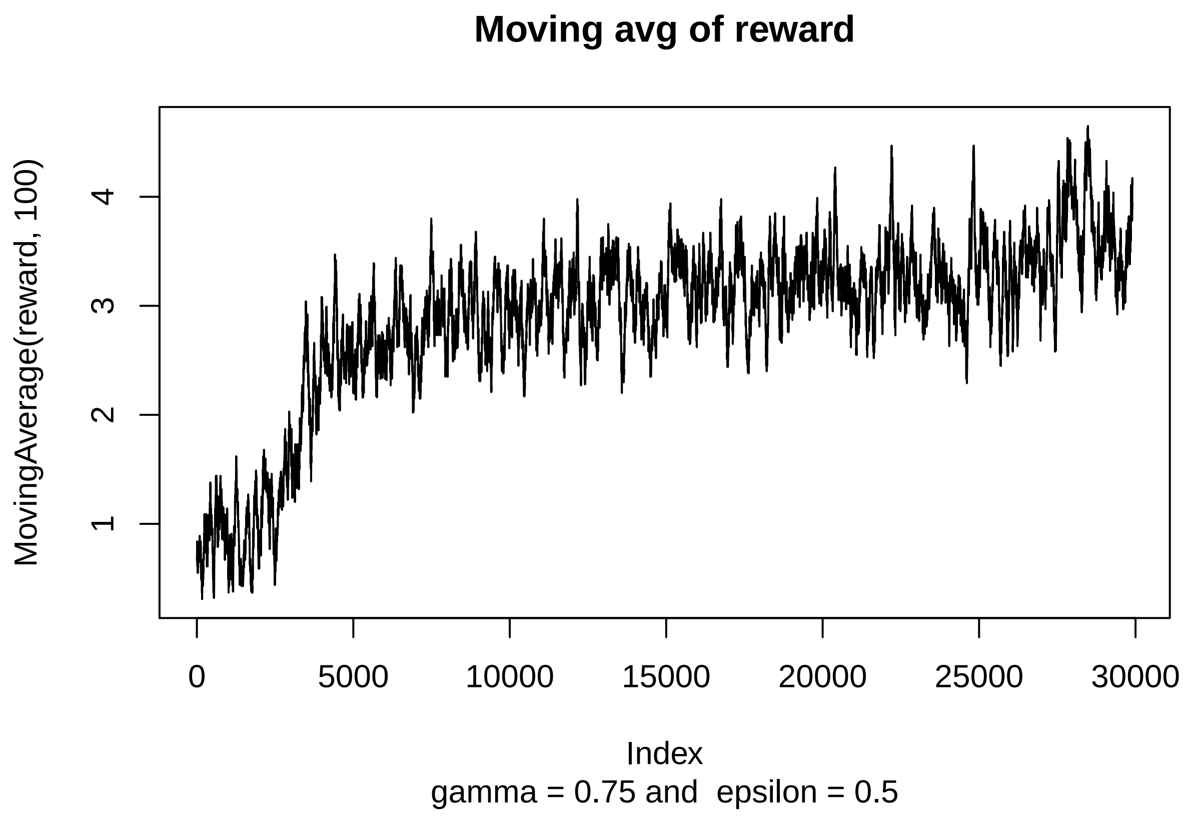
```r
MovingAverage <- function(x, n){

  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", main="Moving avg of reward", sub=paste(expression(gamma),"
                                                                                     expression(epsilon)
  plot(MovingAverage(correction,100),type = "l", main="Moving avg of correction", sub=paste(expression(
                                                                                             expression(
}
```

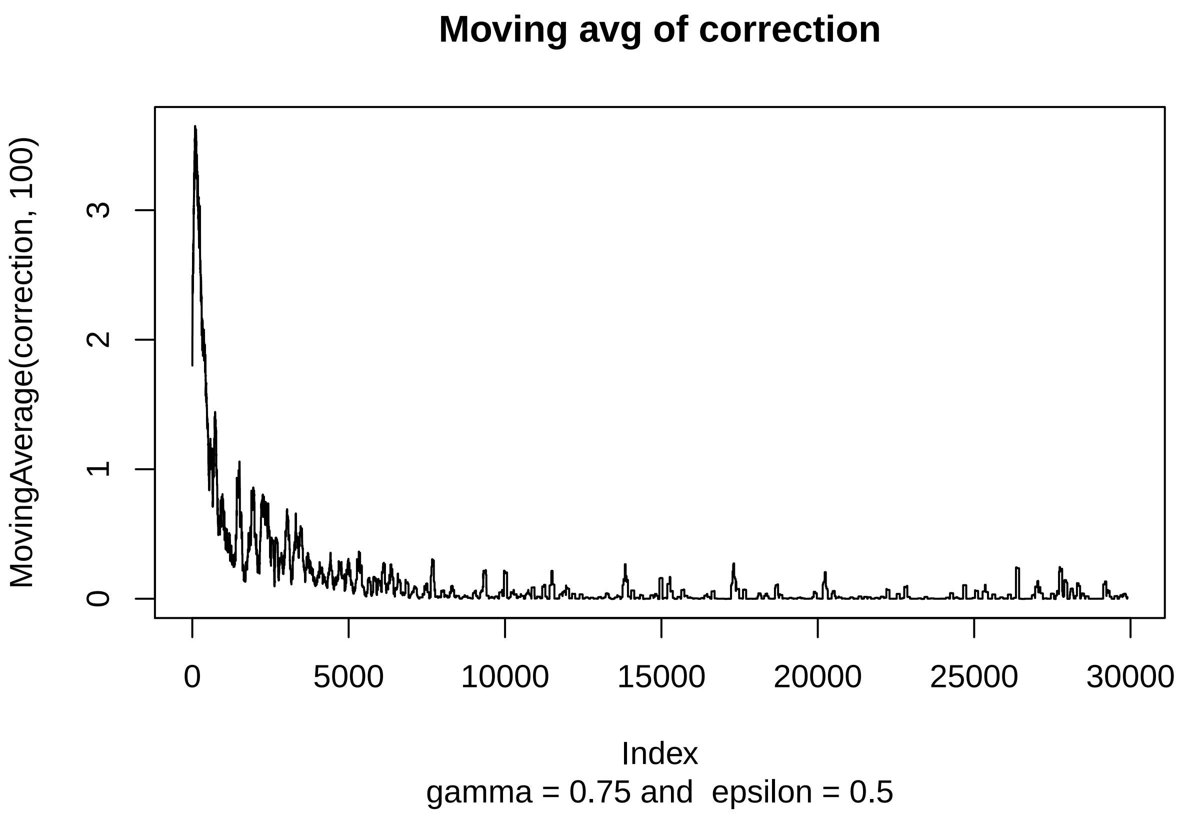


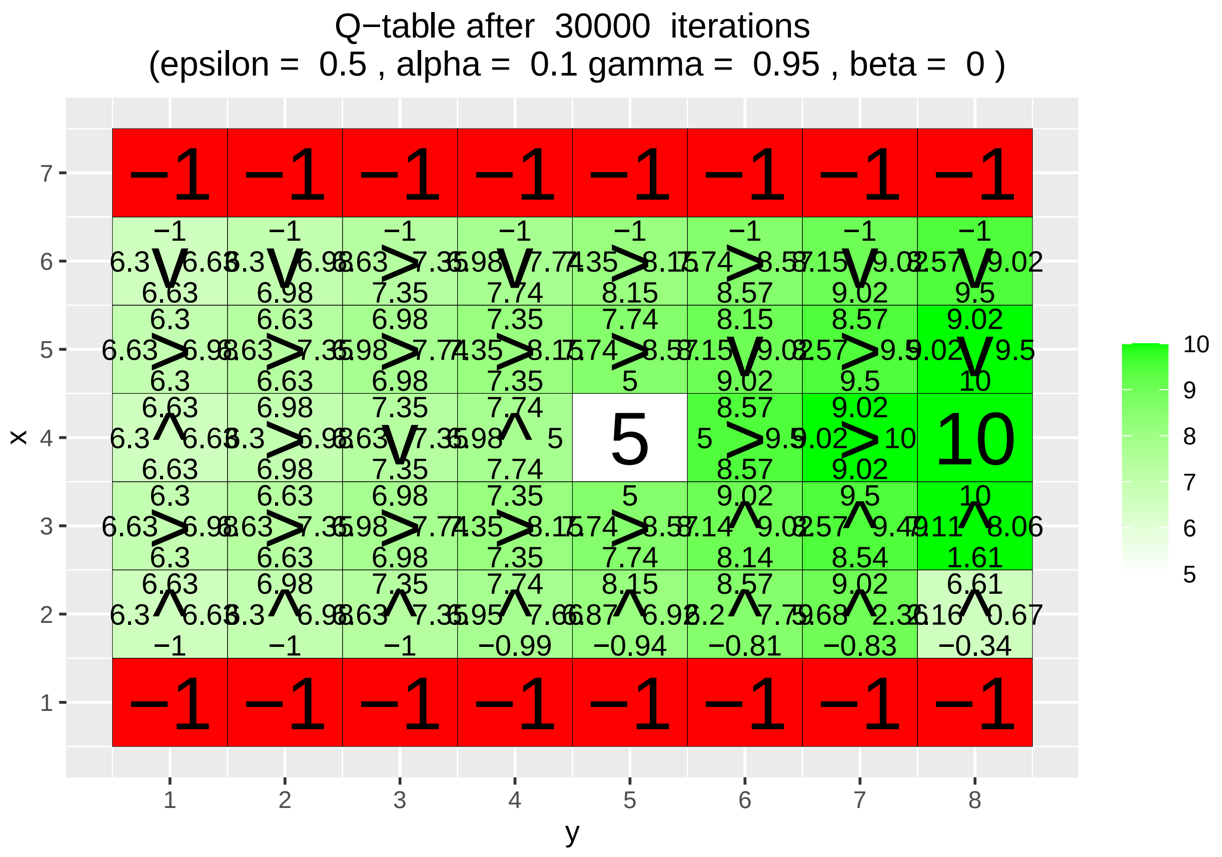

Moving avg of reward
MovingAverage(reward, 100)
0
1
2
3
4
0
5000
10000
15000
20000
25000
30000
Index
gamma = 0.5 and epsilon = 0.5

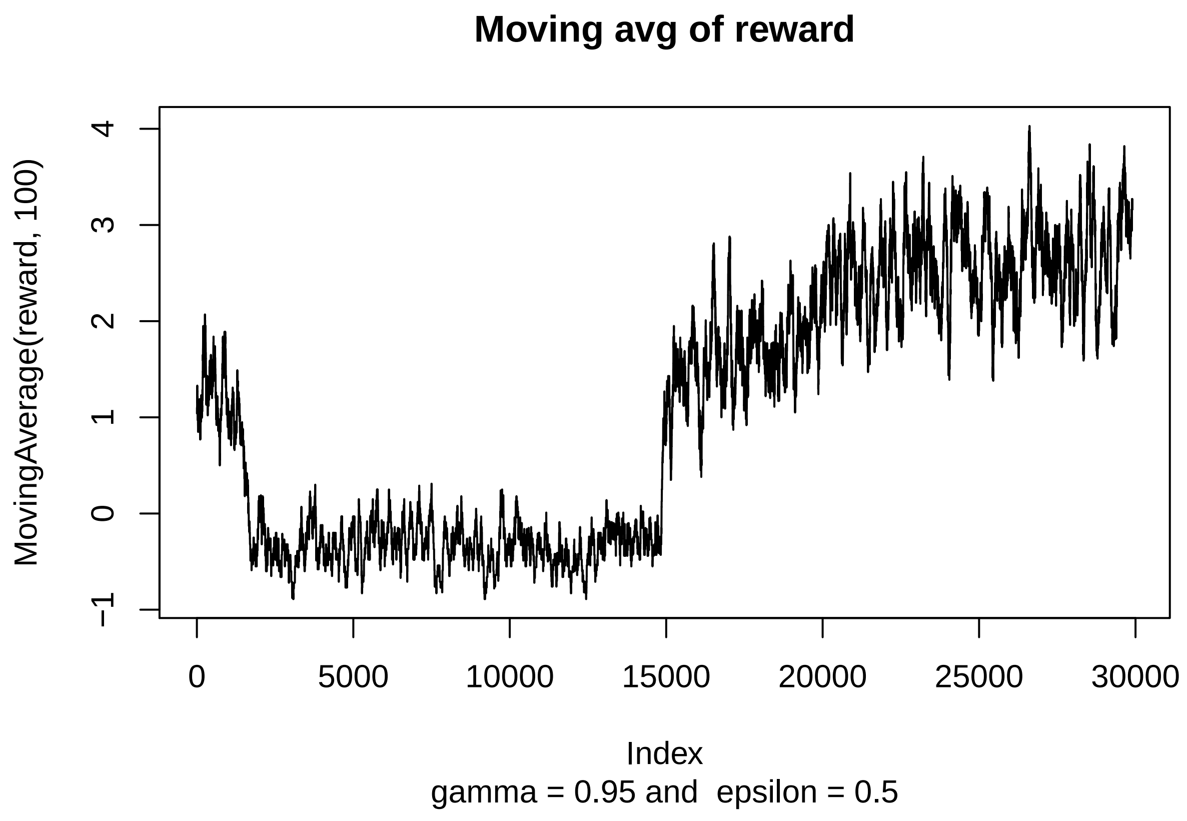

Moving avg of correction
MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
2.0
0
5000
10000
15000
20000
25000
30000
Index
gamma = 0.5 and epsilon = 0.5

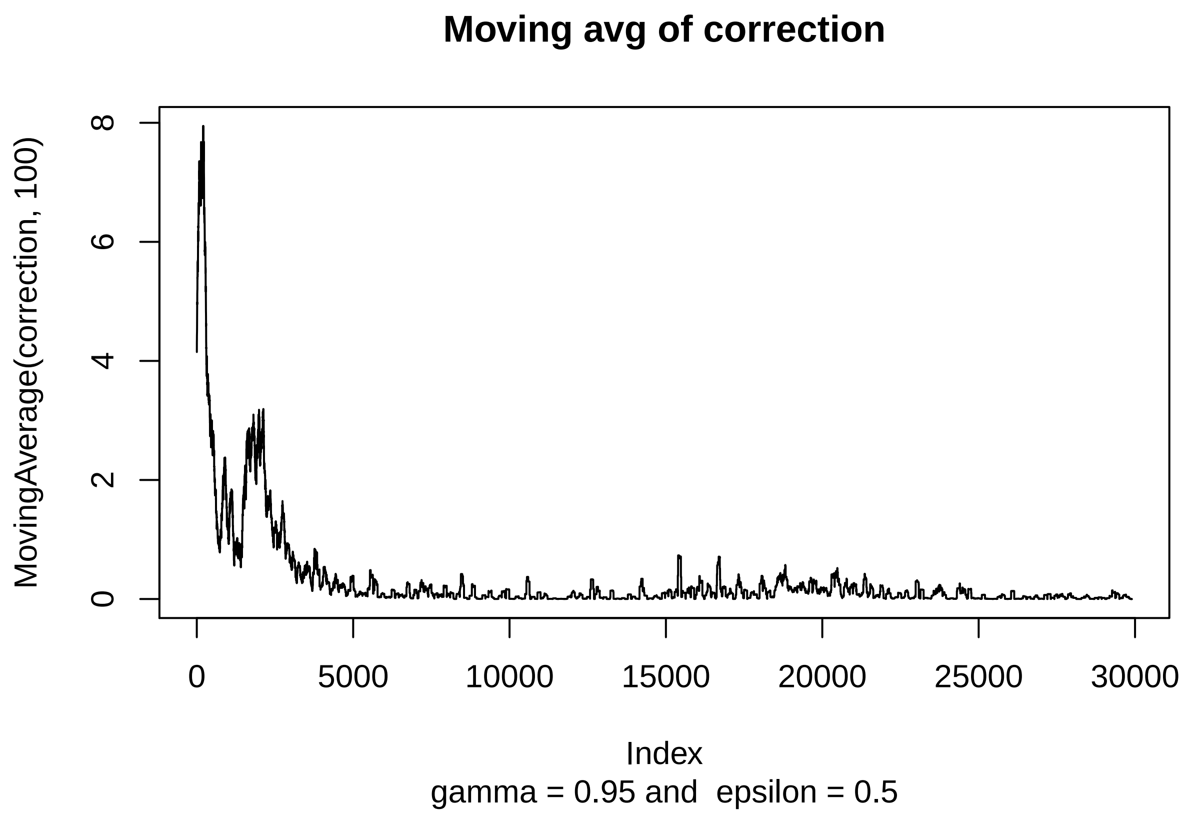

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
1
2
3
4
5
6
7
8
10.0
7.5
5.0
2.5
−1
5
10


Moving avg of reward
MovingAverage(reward, 100)
1
2
3
4
0
5000
10000
15000
20000
25000
30000
Index
gamma = 0.75 and epsilon = 0.5

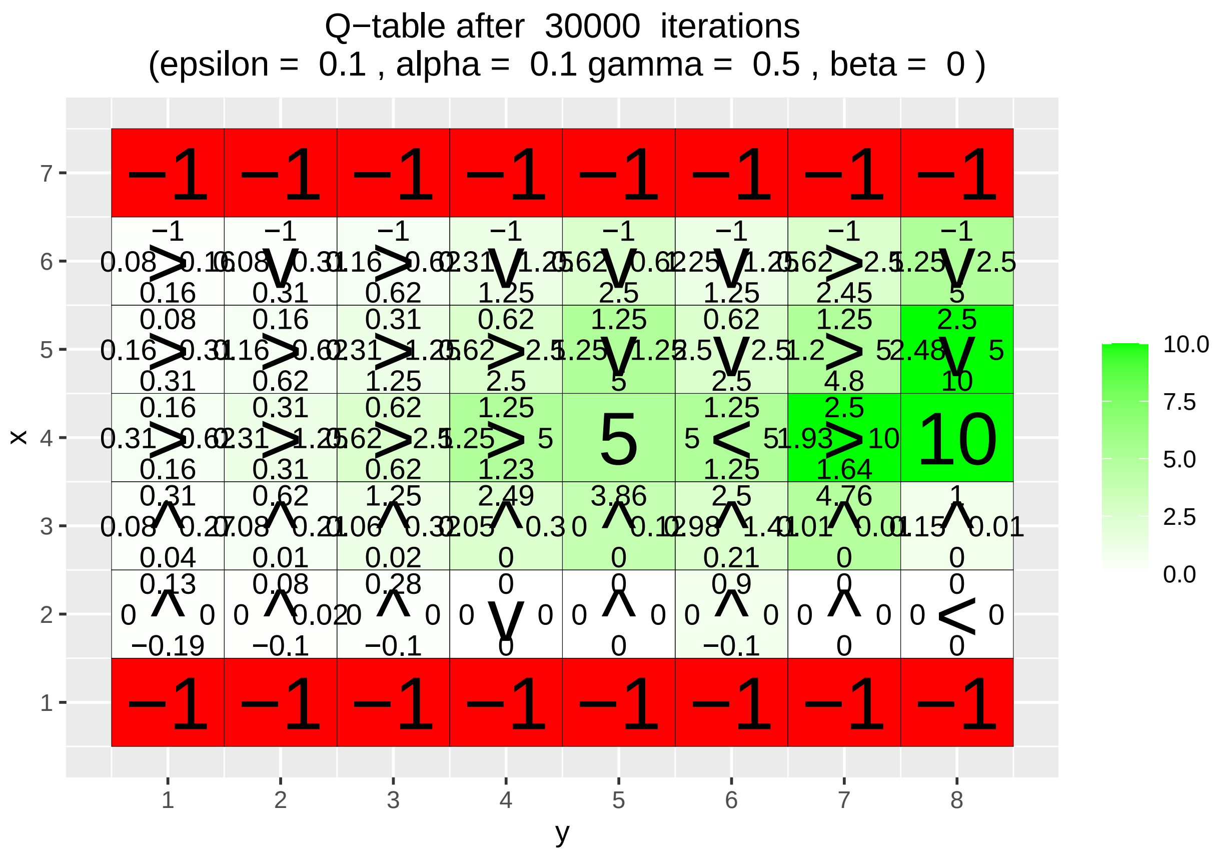

Moving avg of correction
MovingAverage(correction, 100)
0
1
2
3
0
5000
10000
15000
20000
25000
30000
Index
gamma = 0.75 and epsilon = 0.5

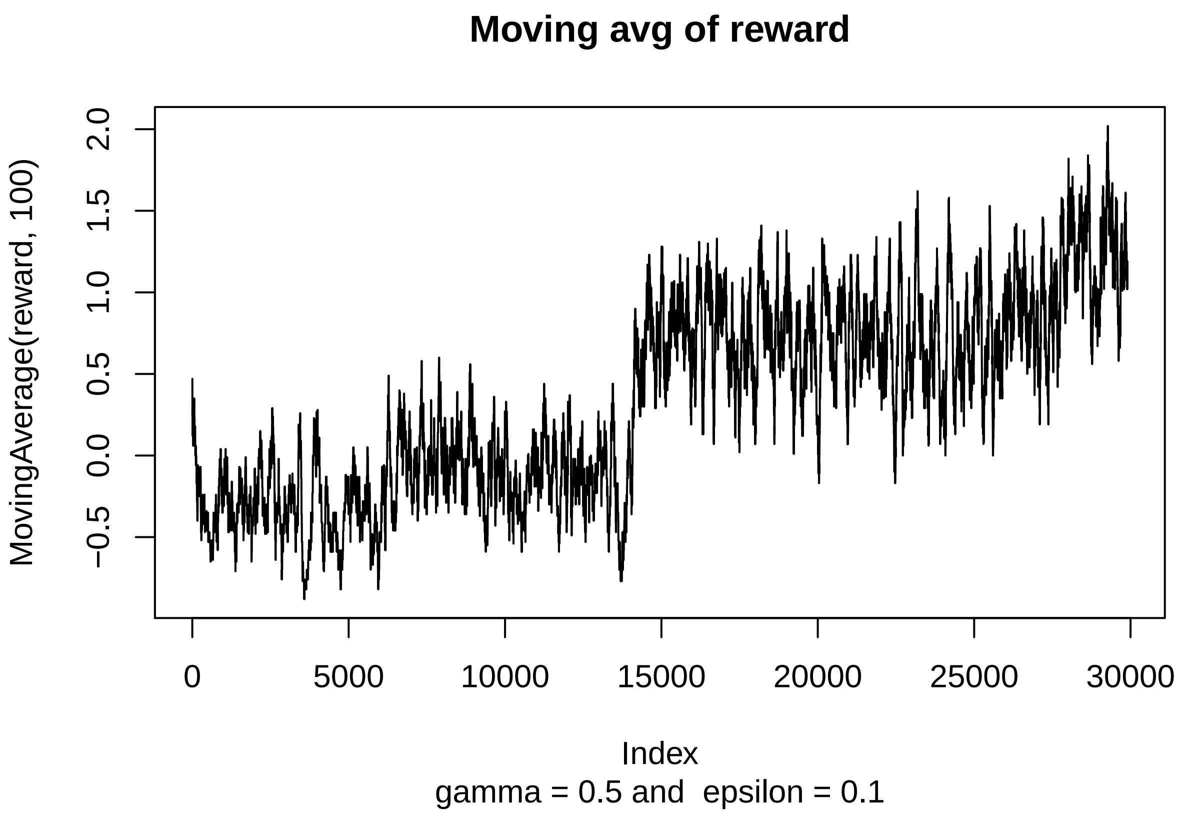

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10
9
8
7
6
5

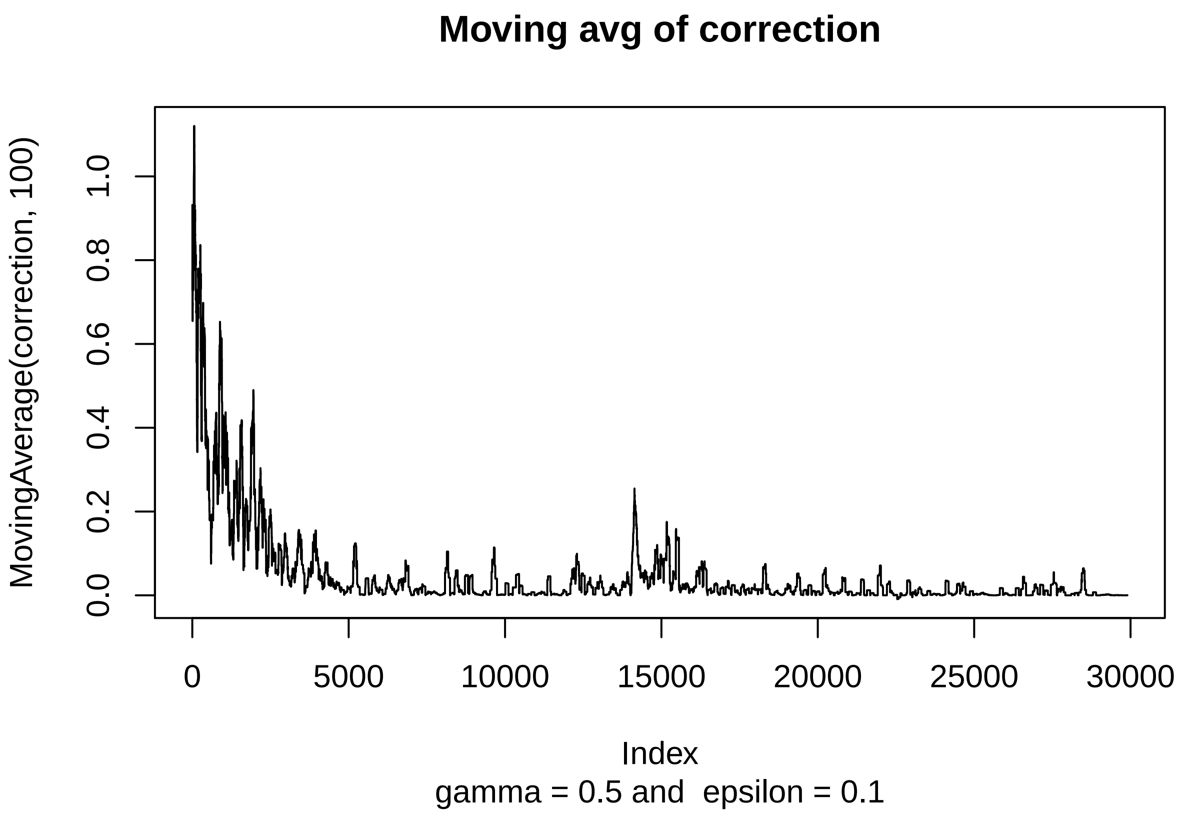

Moving avg of reward
MovingAverage(reward, 100)
−1
0
1
2
3
4
0
5000
10000
15000
20000
25000
30000
Index
gamma = 0.95 and epsilon = 0.5

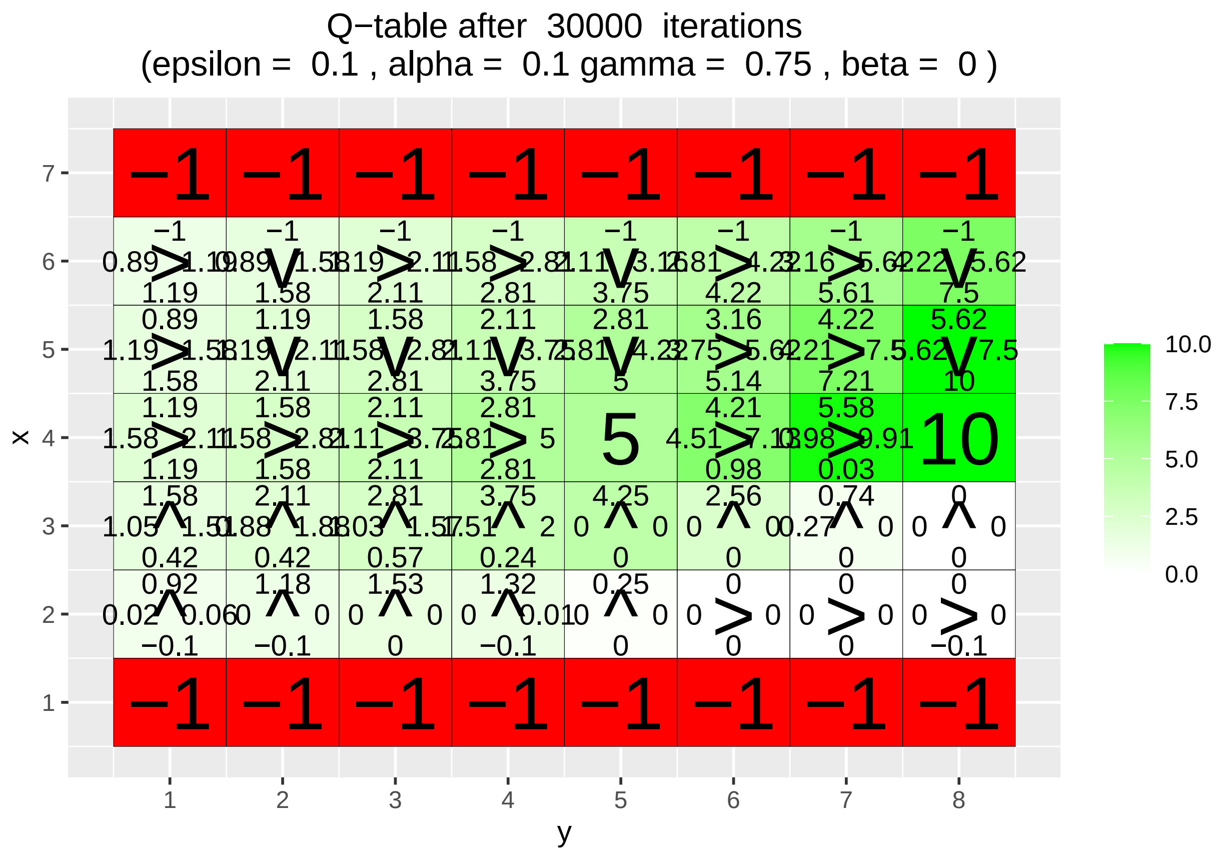


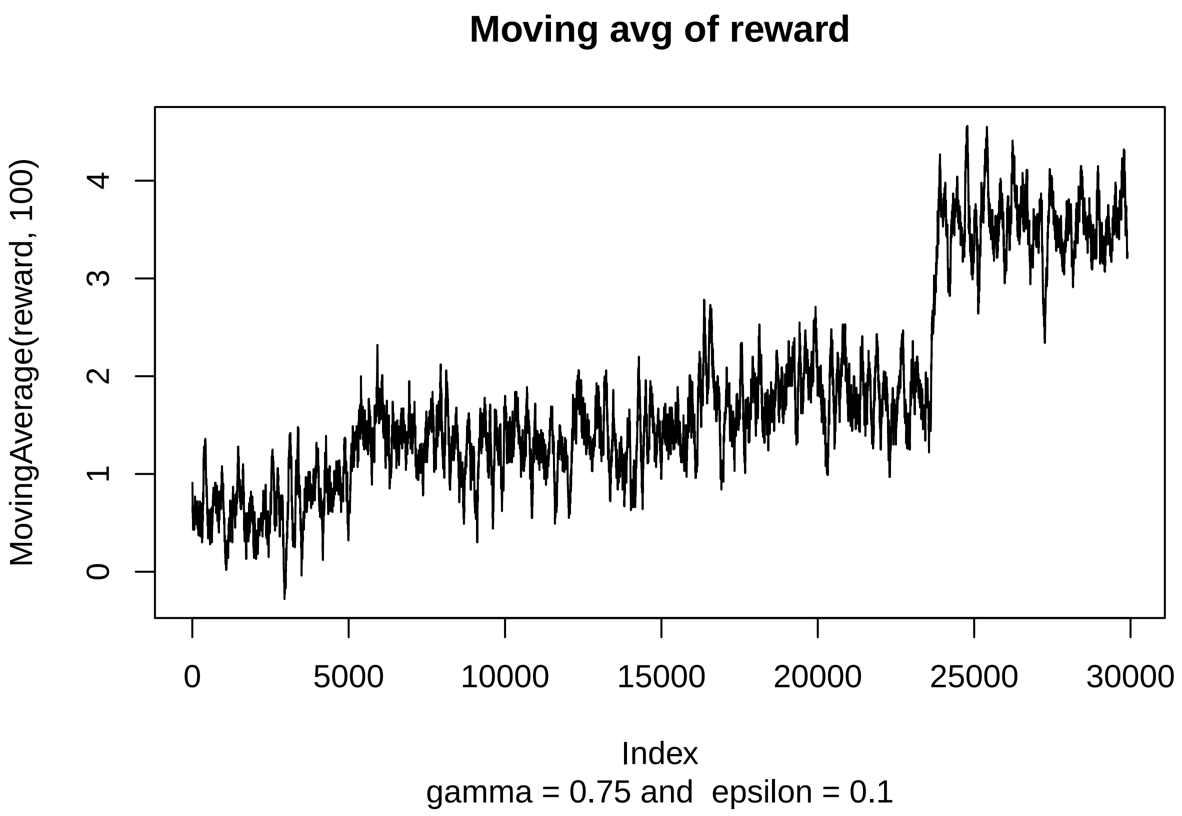
```r
for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  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", main="Moving avg of reward", sub=paste(expression(gamma),"=
                                                                                  expression(epsilon)
  plot(MovingAverage(correction,100),type = "l", main="Moving avg of correction", sub=paste(expression(g
                                                                                          expression(e
}
```

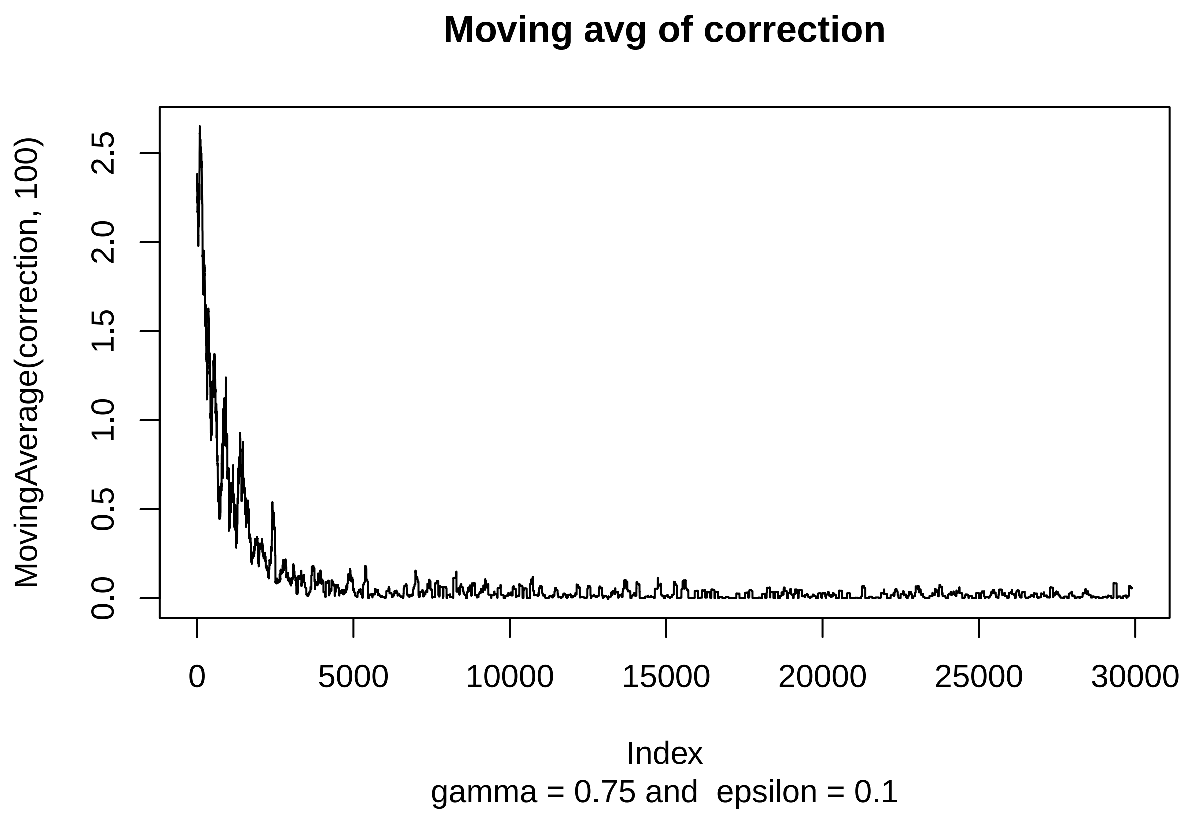

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.5 , beta = 0)
x
y
1
2
3
4
5
6
7
8
−1
5
10
10.0
7.5
5.0
2.5
0.0

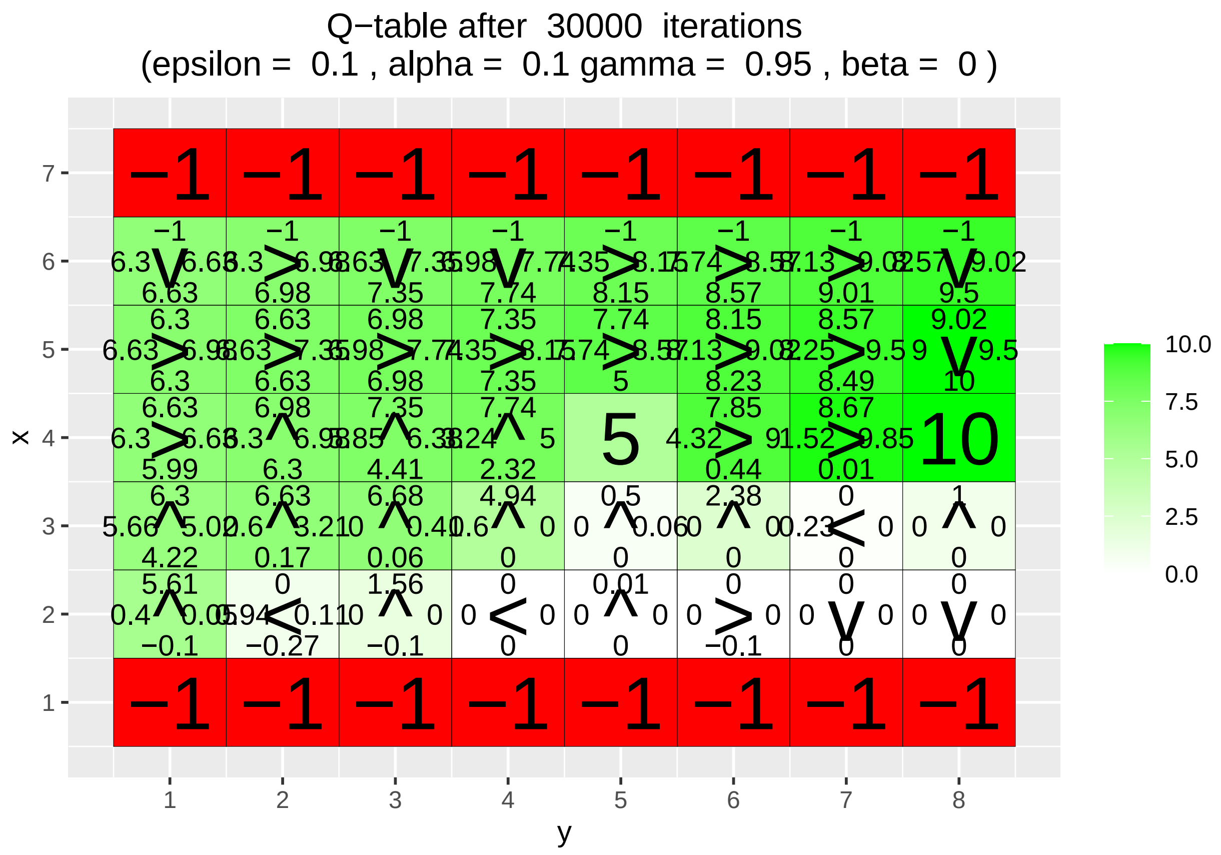

Moving avg of reward
MovingAverage(reward, 100)
−0.5
0.0
0.5
1.0
1.5
2.0
0
5000
10000
15000
20000
25000
30000
Index
gamma = 0.5 and epsilon = 0.1

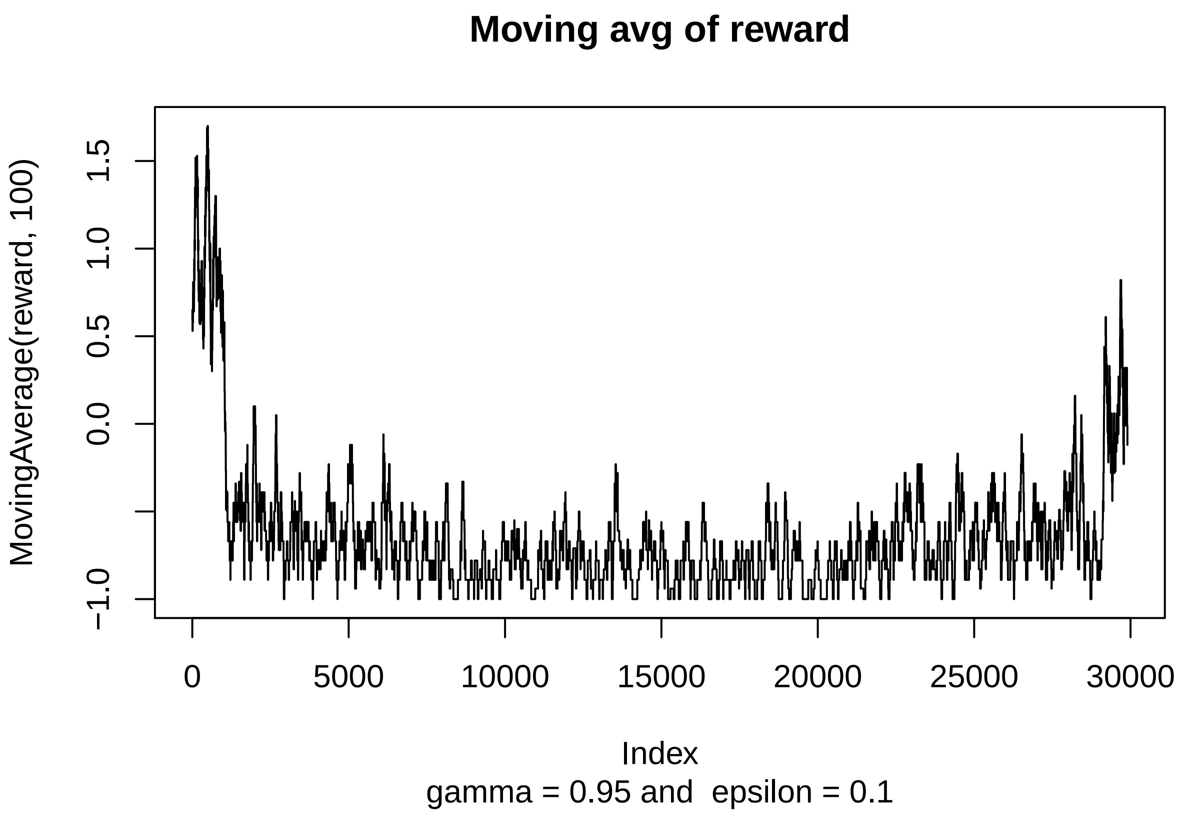

Moving avg of correction
MovingAverage(correction, 100)
0.0
0.2
0.4
0.6
0.8
1.0
0
5000
10000
15000
20000
25000
30000
Index
gamma = 0.5 and epsilon = 0.1

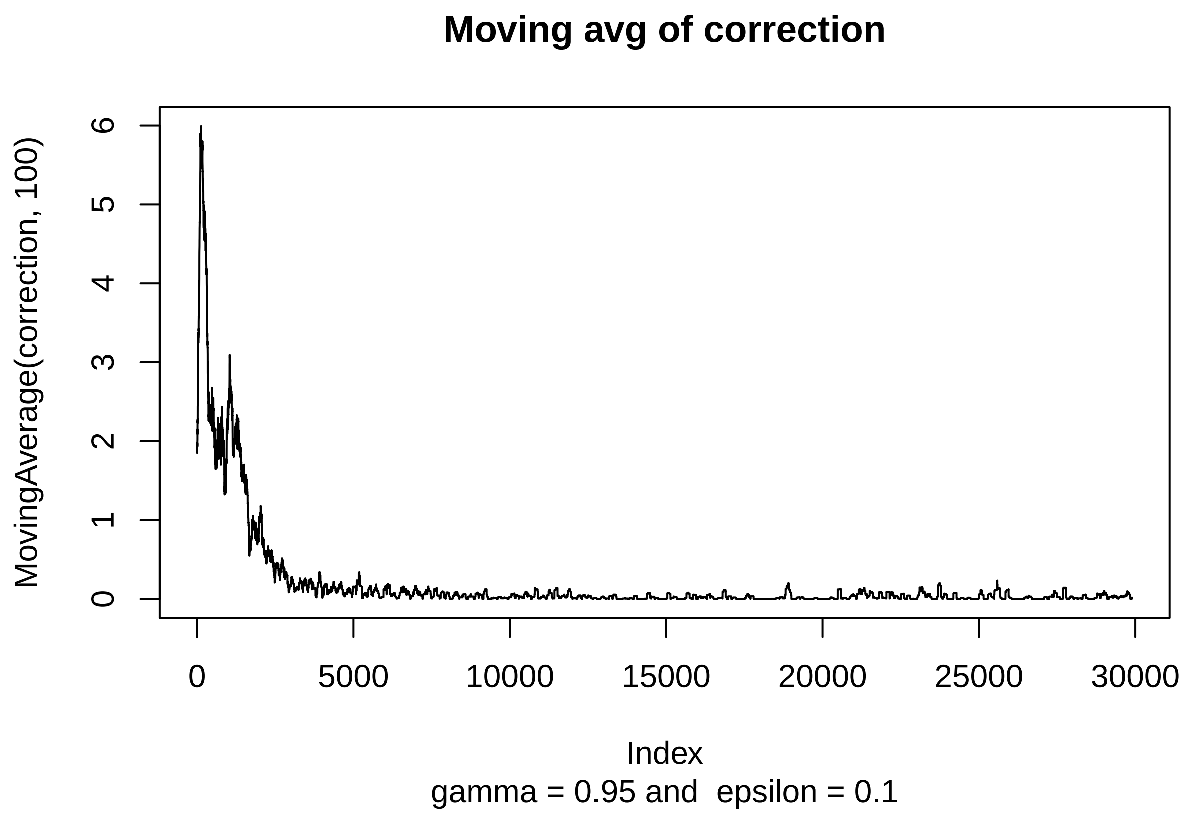

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
1
2
3
4
5
6
7
8
10.0
7.5
5.0
2.5
0.0
−1
5
10


Moving avg of reward
MovingAverage(reward, 100)
0
1
2
3
4
0
5000
10000
15000
20000
25000
30000
Index
gamma = 0.75 and epsilon = 0.1


Moving avg of correction
MovingAverage(correction, 100)
0.0
0.5
1.0
1.5
2.0
2.5
0
5000
10000
15000
20000
25000
30000
Index
gamma = 0.75 and epsilon = 0.1


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


Moving avg of reward
MovingAverage(reward, 100)
−1.0
0.0
0.5
1.0
1.5
0
5000
10000
15000
20000
25000
30000
Index
gamma = 0.95 and epsilon = 0.1

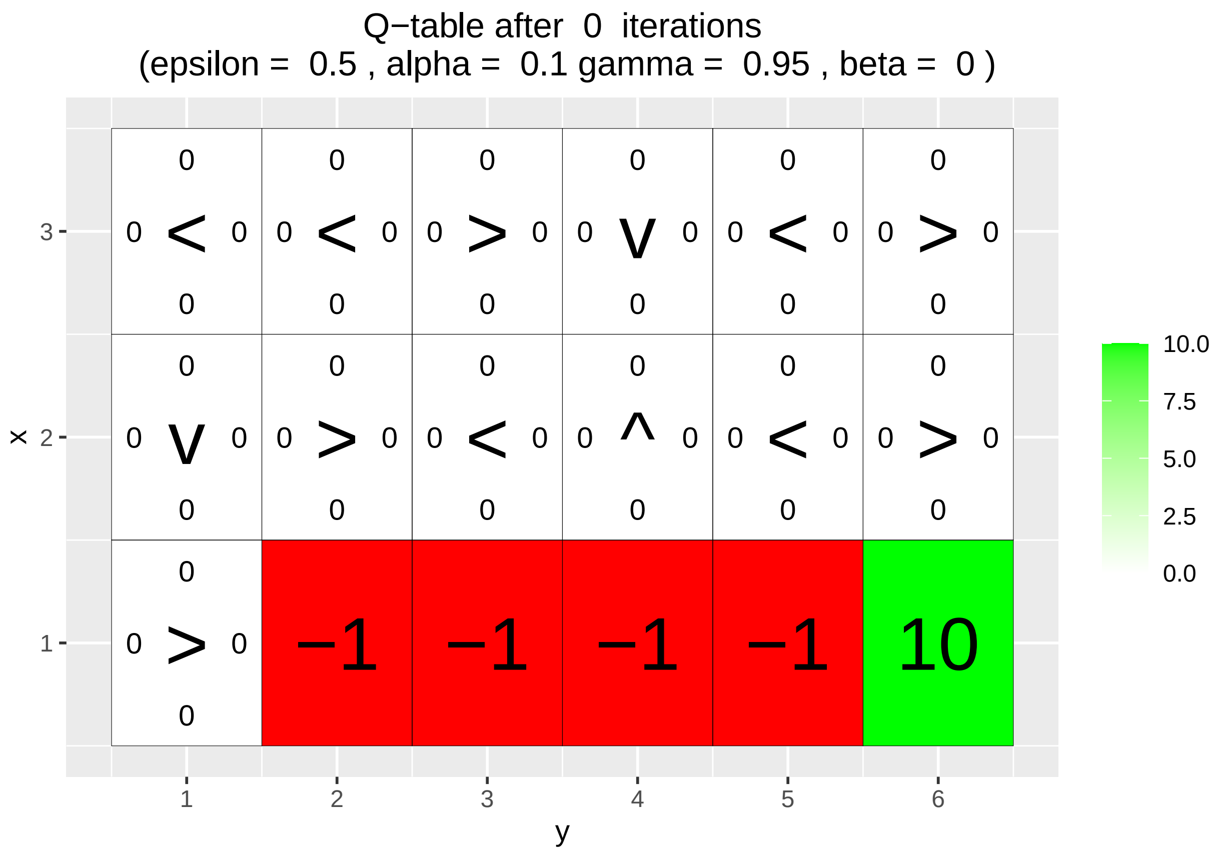


When investigating the plots above it is evident that the policies derived as well as how well the Q-table is explored differ depending on the two parameters γ and ϵ. In the first iteration, when $\gamma = 0.5$ and $\epsilon = 0.5$, the agent tends to choose the reward 5 when the agent is close to it which implies that the weight of the future rewards is not sufficiently high enough. In the third iteration however, when $\gamma = 0.9$, the optimal policy suggests that the agent should move past the reward which yields 5 and continue towards the reward which yields 10. Comparing the first, second and third iteration, when $\epsilon = 0.5$, with the fourth, fifth and sixth iteration, when $\epsilon = 0.1$, it is evident from the outputted Q-tables that quite many states have not been explored yet (since their reward is 0) or have been explored very few times (since their reward is close to 0). To conclude, the ϵ-parameter controlls how exporative the agent should be and the γ-parameter controlls how much the agent should consider future rewards in comparison to rewards close to it. ’

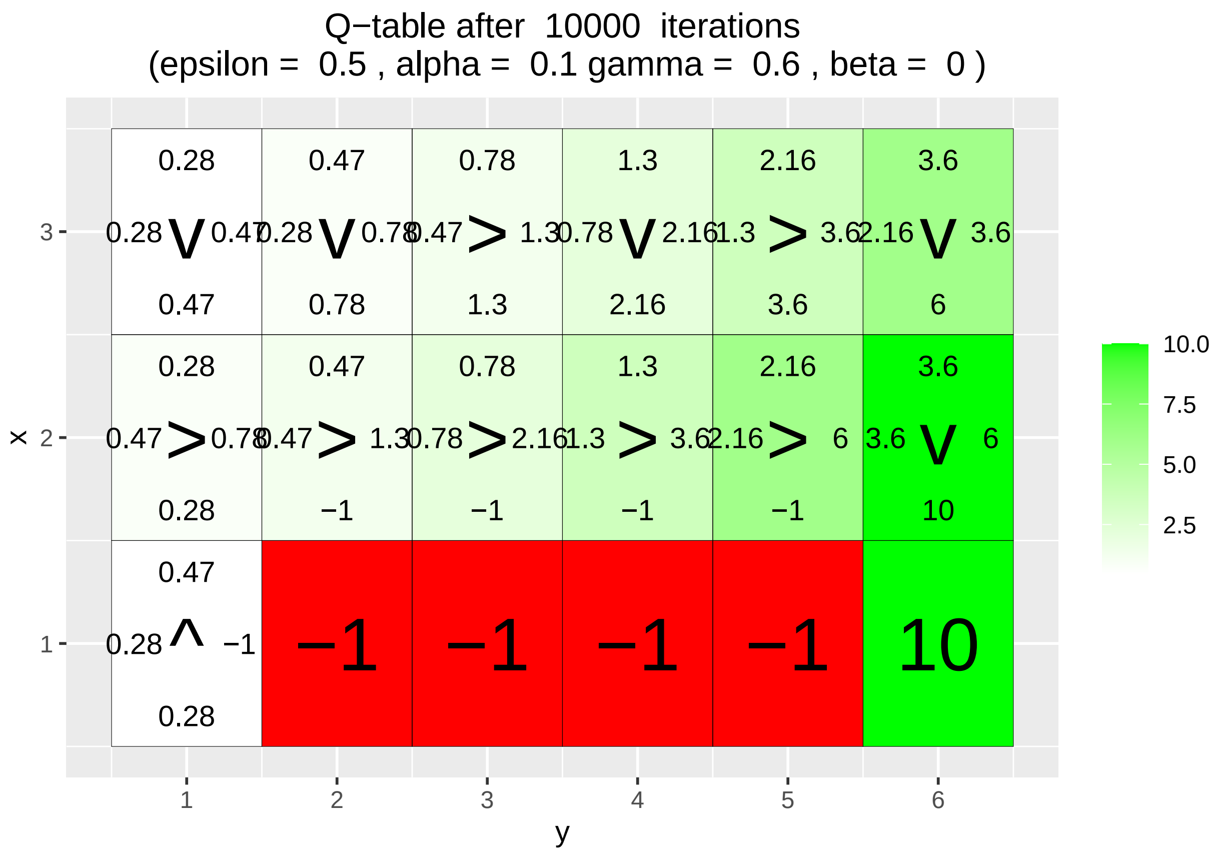
Assignment Q-learning Environment C

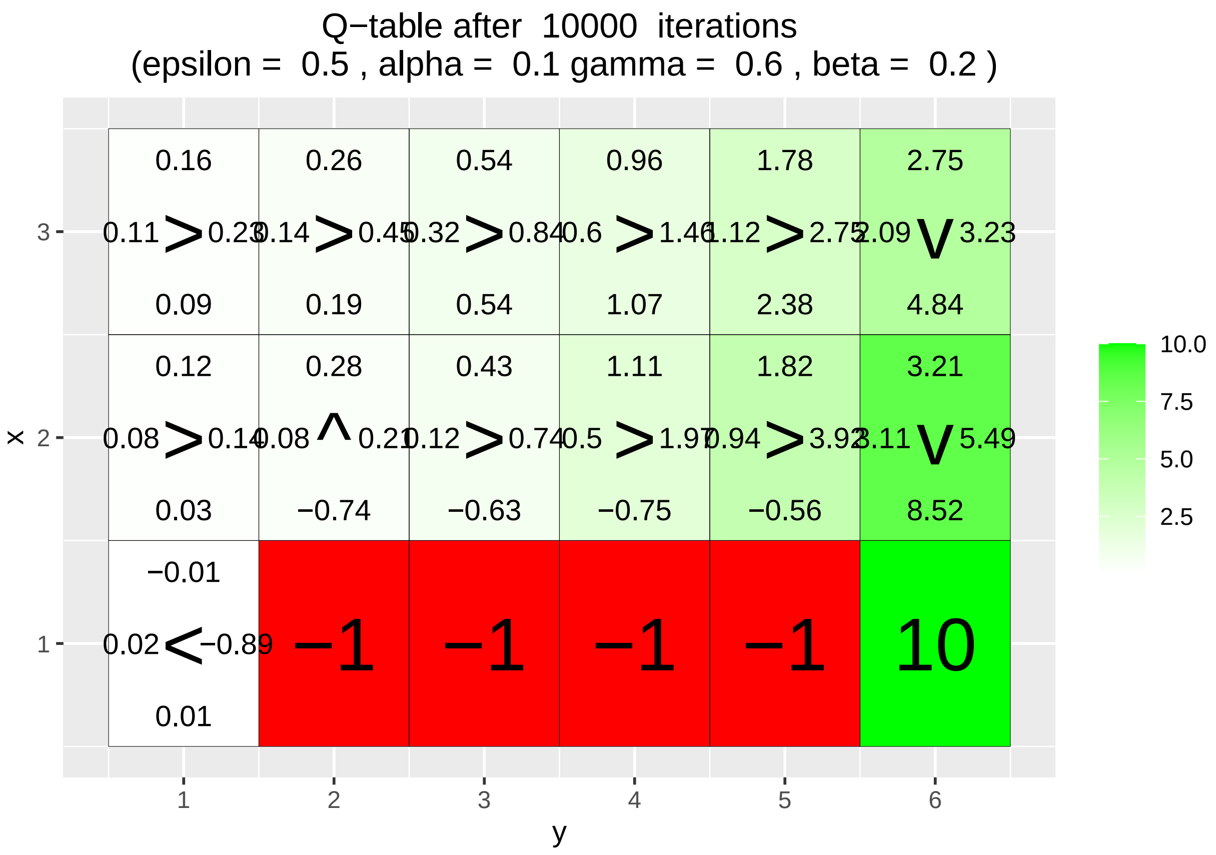
```
# Environment C (the effect of beta).

set.seed(12345)
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))
```

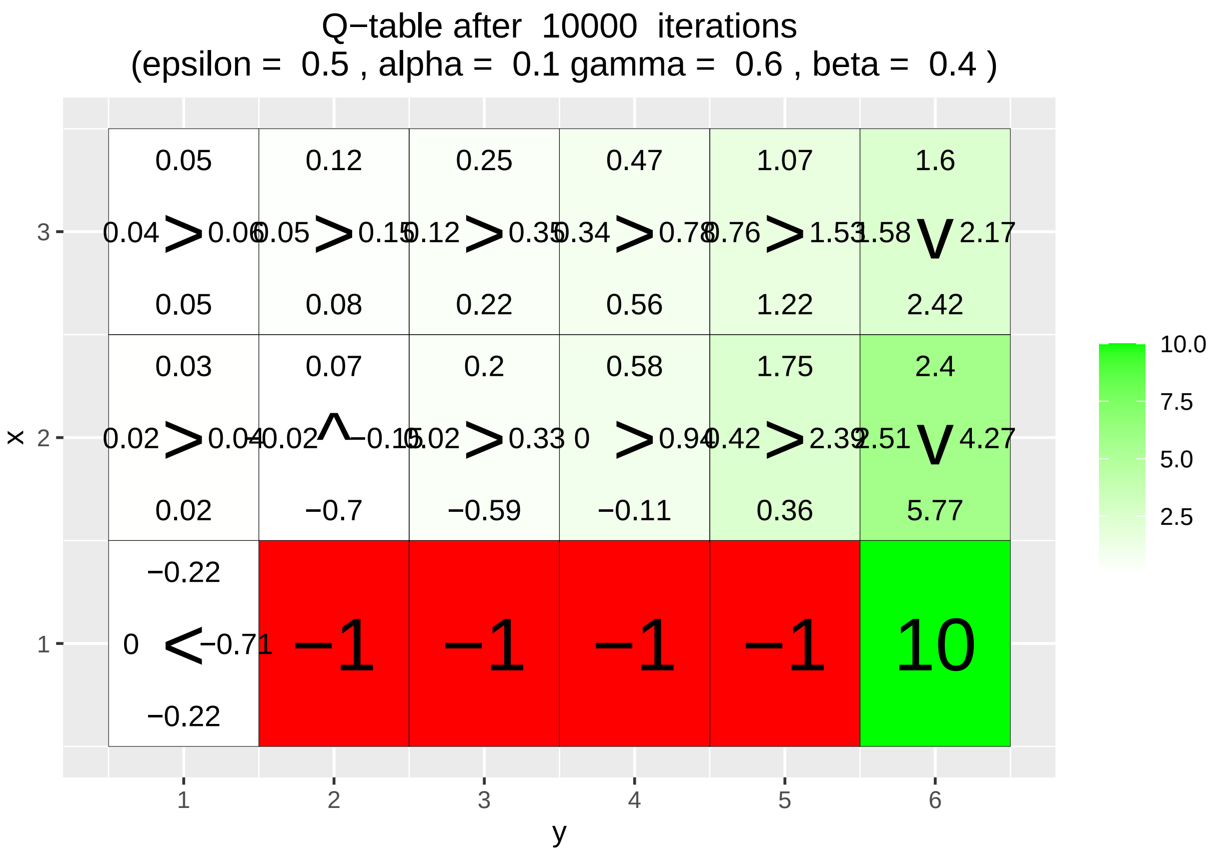
```r
vis_environment()
```

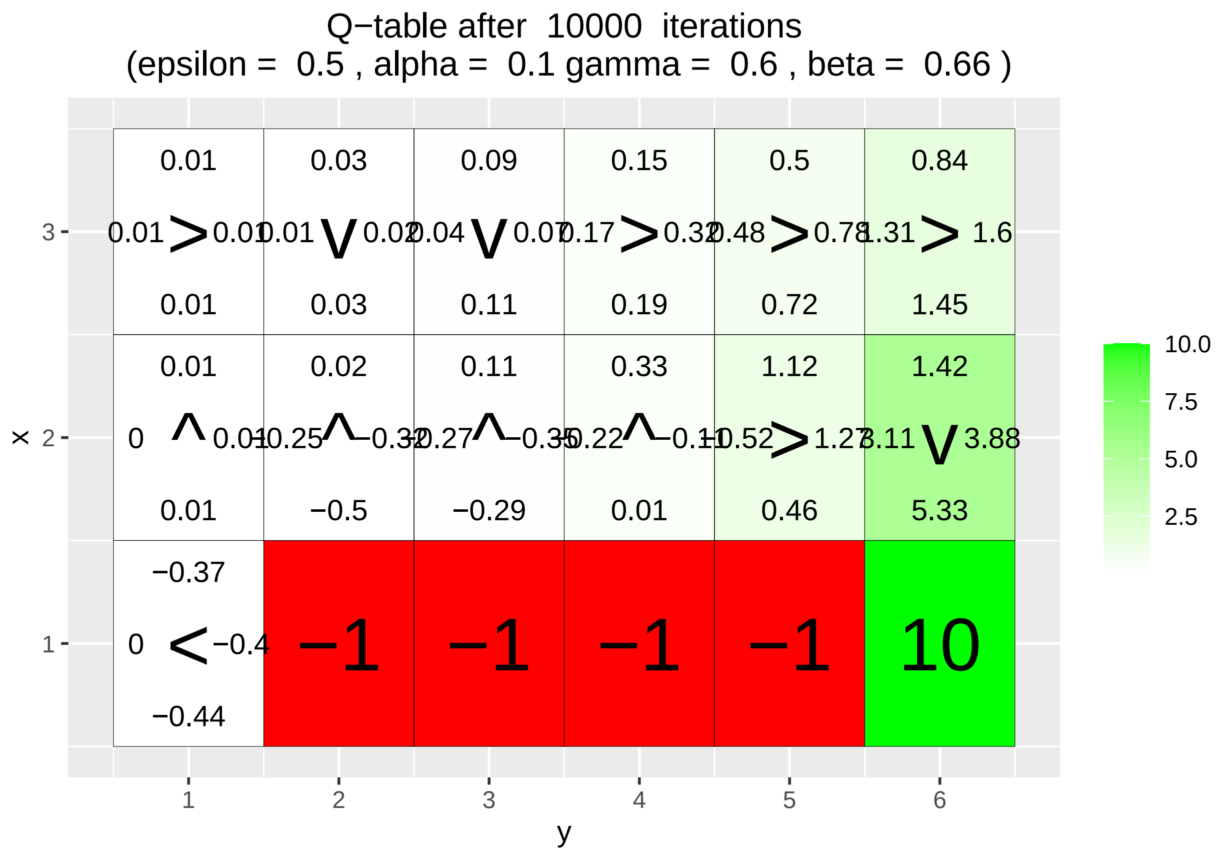


```r
for(j in c(0,0.2,0.4,0.66)){
  q_table <- array(0,dim = c(H,W,4))

  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)
x
y
1
2
3
4
5
6
10.0
7.5
5.0
2.5
−1
10


Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.2)
0.16
0.11 > 0.23
0.09
0.26
0.14 > 0.45
0.19
0.54
0.32 > 0.84
0.54
0.96
0.6 > 1.46
1.07
1.78
1.12 > 2.75
2.38
2.75
2.09 V 3.23
4.84
0.12
0.08 > 0.14
0.03
0.28
0.08 ^ 0.21
−0.74
0.43
0.12 > 0.74
−0.63
1.11
0.5 > 1.97
−0.75
1.82
0.94 > 3.93
−0.56
3.21
3.11 V 5.49
8.52
−0.01
0.02 < −0.89
0.01
−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
0.04 > 0.06
0.05
0.12
0.05 > 0.15
0.08
0.25
0.12 > 0.35
0.22
0.47
0.34 > 0.78
0.56
1.07
0.76 > 1.53
1.22
1.6
1.58 V 2.17
2.42
0.03
0.02 > 0.04
0.02
0.07
−0.02 ^ −0.15
−0.7
0.2
0.02 > 0.33
−0.59
0.58
0 > 0.94
−0.11
1.75
0.42 > 2.39
0.36
2.4
2.51 V 4.27
5.77
−0.22
0 < −0.71
−0.22
−1
−1
−1
−1
10
3
2
1
x
1
2
3
4
5
6
y
10.0
7.5
5.0
2.5



From the Q-tables outputted above it is evident that the β-parameter affects the learned policy by forcing the agent to make actions which is not optimal. In the first table, when $\beta = 0$, the agent learns the optimal policy without troubles. No randomness has been introduced to the agent to stop it from learning. However, as β increases, it is evident that the agent experiences more and more issues in learning the optimal path. For example, when $\beta = 0.66$, it is quite probable that the episode terminates in a negative reward. This would result in the agent *avoiding* the negative reward rather than *aiming* for the higher reward when the agent is far away from the higher reward. If the most likely event for the agent is to slip into the negative reward when it is far away from the higher reward it will never learn to modify the rewards of this state to point at the high reward since this reward is unlikely to be introduced when the agent is present in those states. To conclude, higher β-value means more randomness into the action taken which might be good to prevent the agent from getting stuck at places, but in this case it is of absolute no good since the agent will only learn to avoid the negative rewards and not to aim for the high reward.

Assignment REINFORCE Environment D

(Could not run the code. Looked at the outputs from another computer)

Looking at the plots it seems that the agent has learned a good policy since all the states guide the agent towards the goal. It is although evident that some of the states point towards the goal with not so high probability even if the goal is right next to the state.

The Q-learning algorithm would be possible to model against the task - this is if we could add goal coordinates in the Q-table. It would however be more difficult to train towards one goal and then validate towards another since the goal might be in unseen territories.

Assignment REINFORCE Environment E

The agent has not learned a good policy since not all states point towards the goal.

The results differ in comparison to environment D since in environment E the agent has never learned to reach a goal through going down (since the goals were all in the top row in training). This is why in all the states below the goal the agent has not learned to move up but instead suggests the agent to go in another direction.

Appendix

```r
# By Jose M. Peña and Joel Oskarsson.
# For teaching purposes.
# jose.m.pena@liu.se.

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

use_condaenv("3.6-lab3", required = TRUE)
# install.packages("ggplot2")
# install.packages("vctrs")
library(ggplot2)


# 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){

  # 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}.

  movement1 = which(q_table[x,y,] == max(q_table[x,y,]))
  if(length(movement1)>1) {
    return(sample(movement1, 1))
  } else {
    return(movement1)
  }
}

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}.

  movement1 = sample(which(q_table[x,y,] == max(q_table[x,y,])), 1)
  draw=runif(1, min=0, max=1)
  if(draw>epsilon) {
    return(movement1)
  } else {
    return(sample(c(1,2,3,4), 1))
  }
```

```r
}

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 <<-.

  q_state=start_state
  reward=0
  episode_correction=0

  repeat{
    # Follow policy, execute action, get reward.
    x=q_state[1]
    y=q_state[2]
    action=EpsilonGreedyPolicy(x, y, epsilon)
    new_state=transition_model(x, y, action, beta)
    reward=reward_map[new_state[1], new_state[2]]

    # Q-table update.
    episode_correction=episode_correction+reward+gamma*max(q_table[new_state[1], new_state[2],])-q_tabl
    correction=alpha*(reward+gamma*max(q_table[new_state[1], new_state[2],])-q_table[x,y,action])
    q_table[x,y,action]<<-q_table[x,y,action]+correction
    q_state=new_state

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

}

#####################################################################################################
# Q-Learning Environments
#####################################################################################################

# Environment A (learning)

set.seed(12345)
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)

set.seed(12345)

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)
  plot(MovingAverage(reward,100),type = "l", main="Moving avg of reward", sub=paste(expression(gamma),"=
                                                                                  expression(epsilon)
  plot(MovingAverage(correction,100),type = "l", main="Moving avg of correction", sub=paste(expression(g
                                                                                          expression(e
}

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", main="Moving avg of reward", sub=paste(expression(gamma),"=
                                                                                  expression(epsilon)
  plot(MovingAverage(correction,100),type = "l", main="Moving avg of correction", sub=paste(expression(g
                                                                                          expression(e
}

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

```r
set.seed(12345)
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)
}

## Questions regarding REINFORCEMENT

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)

}
```

```r
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
    }
  }

}

#####################################################################################################
# REINFORCE Environments
#####################################################################################################

# Environment D (training with random goal positions)

H <- 4
W <- 4
library(keras)
library(tensorflow)
library(reticulate)
library(tidyverse)
```

```r
# Define the neural network (two hidden layers of 32 units each).
model <- keras_model_sequential()
model %>%
  layer_dense(units = 32, input_shape = c(4), activation = 'relu') %>%
  layer_dense(units = 32, activation = 'relu') %>%
  layer_dense(units = 4, activation = 'softmax')

compile(model, loss = "categorical_crossentropy", optimizer = optimizer_sgd(lr=0.001))

initial_weights <- get_weights(model)

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

show_validation <- function(episodes){

  for(goal in val_goals)
    vis_prob(goal, episodes)

}

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)

# Environment E (training with top row goal positions)

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))

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)
```