Lab 3

Axel Holmberg (axeho681)

10/4/2020

Environment A

Below are the environments after 10, 100 and 1000 and 10000 runs done with the Greedy Policy. Questions and answers under images.

- *What has the agent learned after the first 10 episodes?*

As one can see in the image above the agent has not learned much in the first 10 iterations. The only q-value that has been updated is after having gotten a negative reward in $x = 3, y = 2$.

- *Is the final greedy policy (after 10000 episodes) optimal? Why / Why not ?*

The image above shows the path after 10000 iterations. The policy is almost optimal apart from mainly the area around $(4, 2)$ where the arrows should be up instead and in $(1, 3)$.

- *Does the agent learn that there are multiple paths to get to the positive reward ? If not, what could be done to make the agent learn this ?*

Yes, but the agent. Although it it mainly taking the path below the negative reward, which decreases the amount of possible paths it can take.

Environment B

As γ denounces the discount factor that means that it adjusts how much of the values of the Q-table should affect the new updated Q-value, called correction. One can see this quite clearly if one compares $\gamma = 0.75$ with $\gamma = 0.95$.

The big difference one can see in the images above is how all the q-values are a lot higher when $\gamma = 0.95$.

One more thing that is clearly visible in these images is the effect of ϵ. ϵ is the threshold for exploration. As $\epsilon = 0.5$ in both of the above that means than in each evaluation of the ϵ-greedy policy there is a 50% chance that it instead takes a random step. That means that it has a high chance of exploring more of the tiles. If one compares this with where $\epsilon = 0.1$ like the images below the results is quite clear.

As one can see it almost never takes a step beyond the tile with a reward of 5 as the low exploration rate makes it go straight to the tile with a reward of 5. This can also be seen in the graph below showing the rolling mean of the rewards where it aslmost always 5.

One can also look at the correction graph. The correction graph (see below) shows the correction of each step. This shows how much of a correction that occurs with each step.

The difference between these two are quite significant. The higher value of ϵ leads to more variation and mainly a higher correction overall. What this means is that with each episode the correction of the q values are higher. The reason for is that the higher the probability of acting greedily is the more different paths and more will be discorvered and the more correction is done each step and episode.

There are more to be analysed from the data and the graphs, but above is the key takeaways that I did of the graphs.

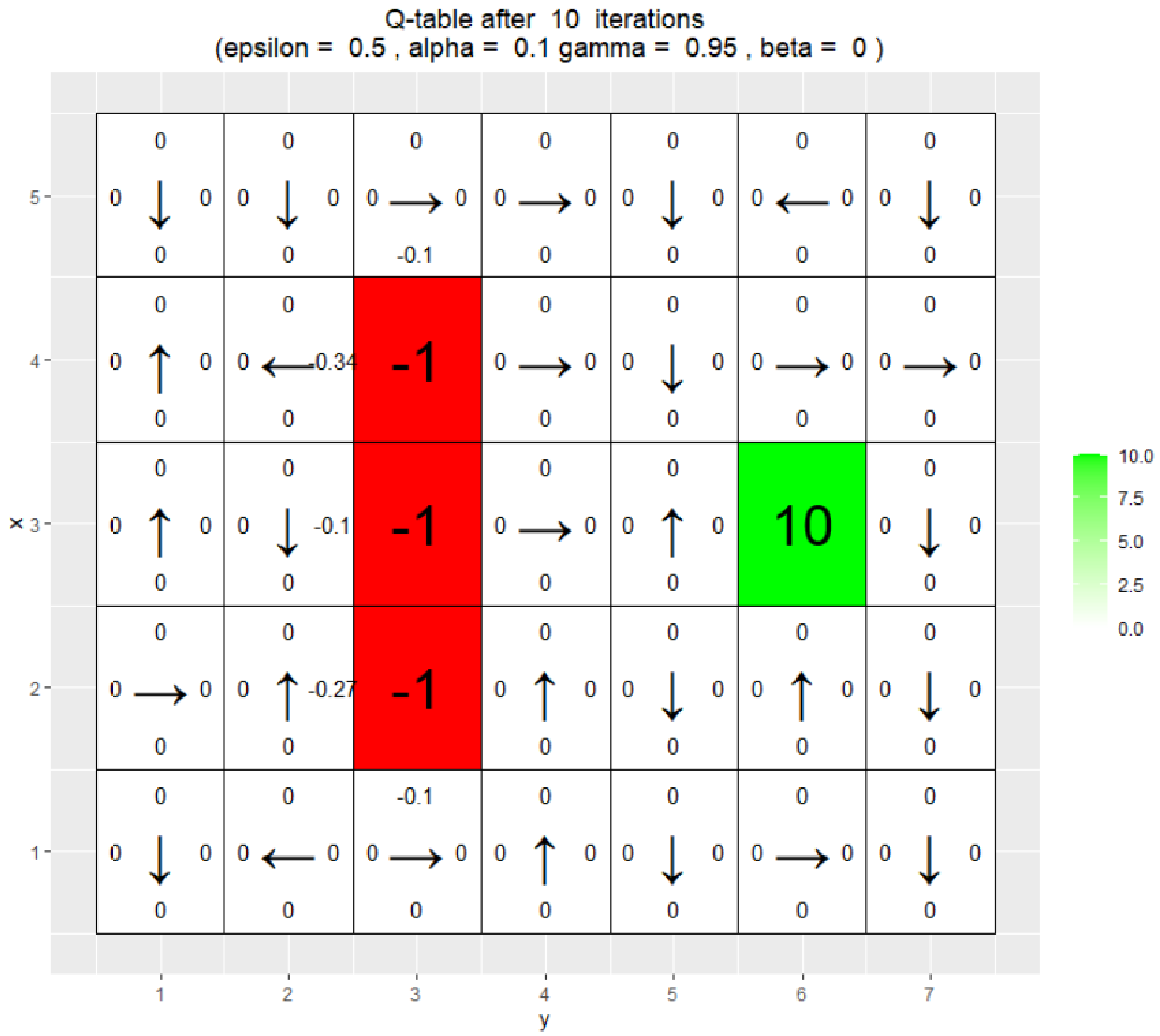

Figure 1: 10 iterations

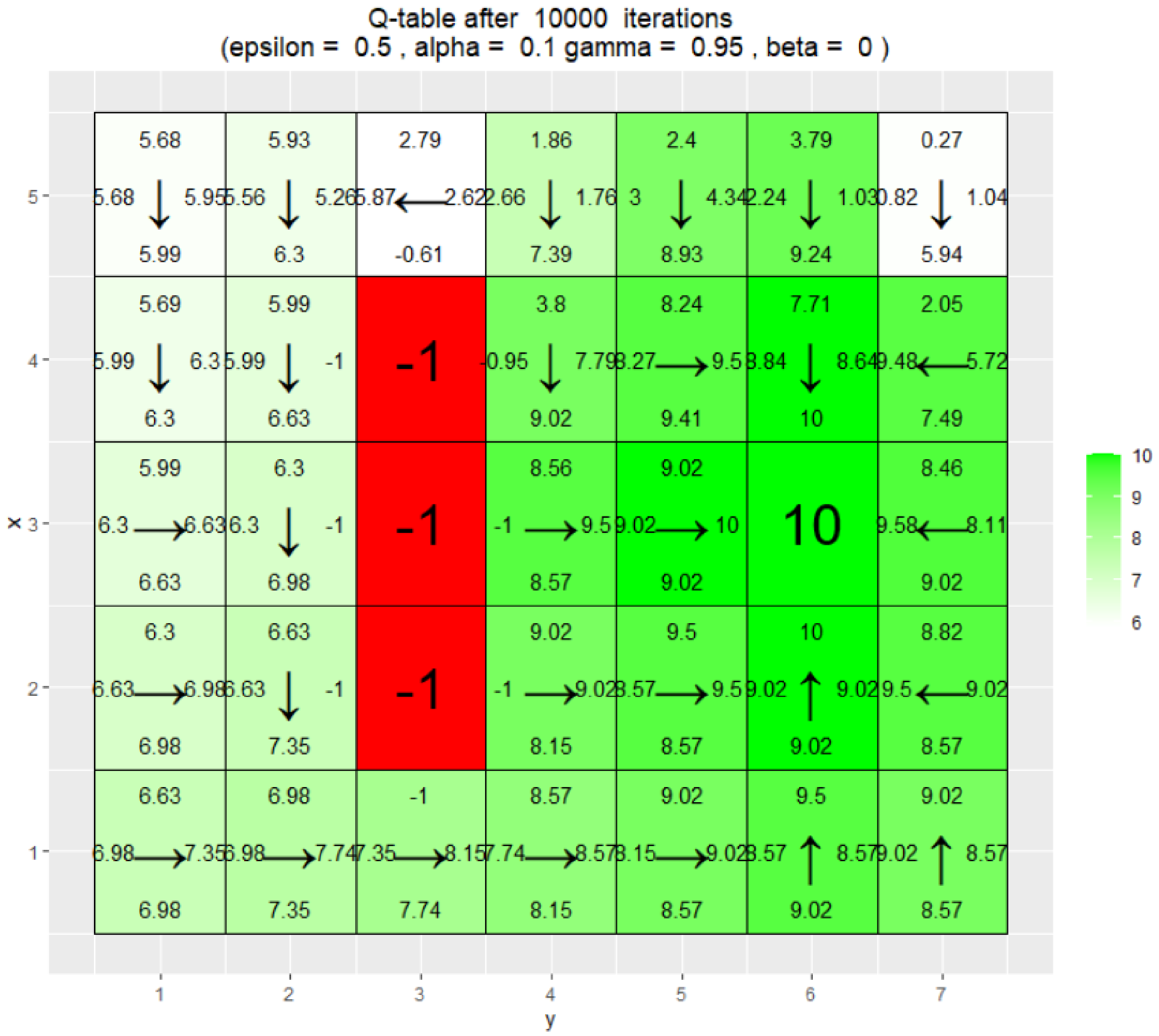

Figure 2: 10 000 iterations

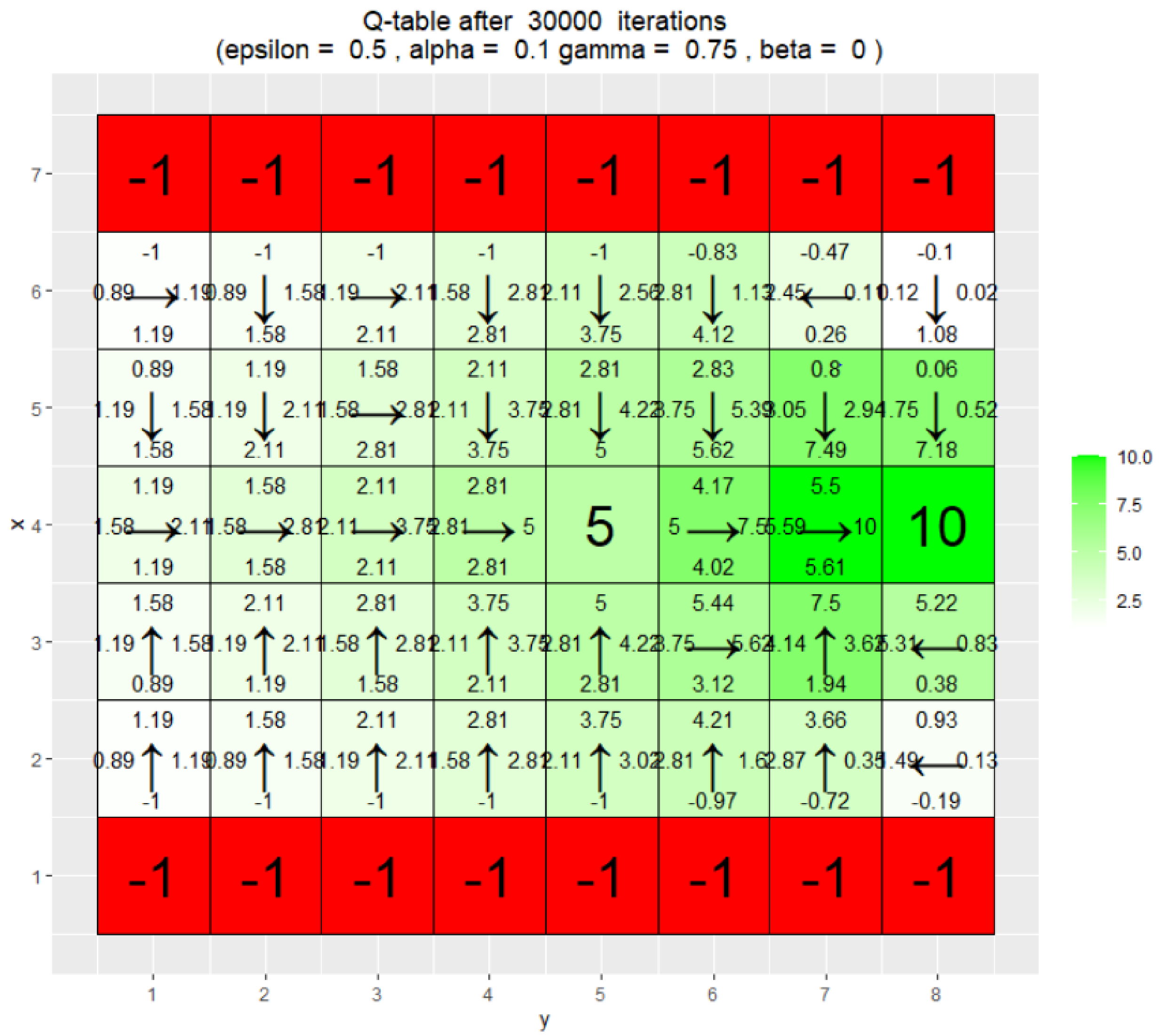

Figure 3: $\gamma = 0.75, \epsilon = 0.5$

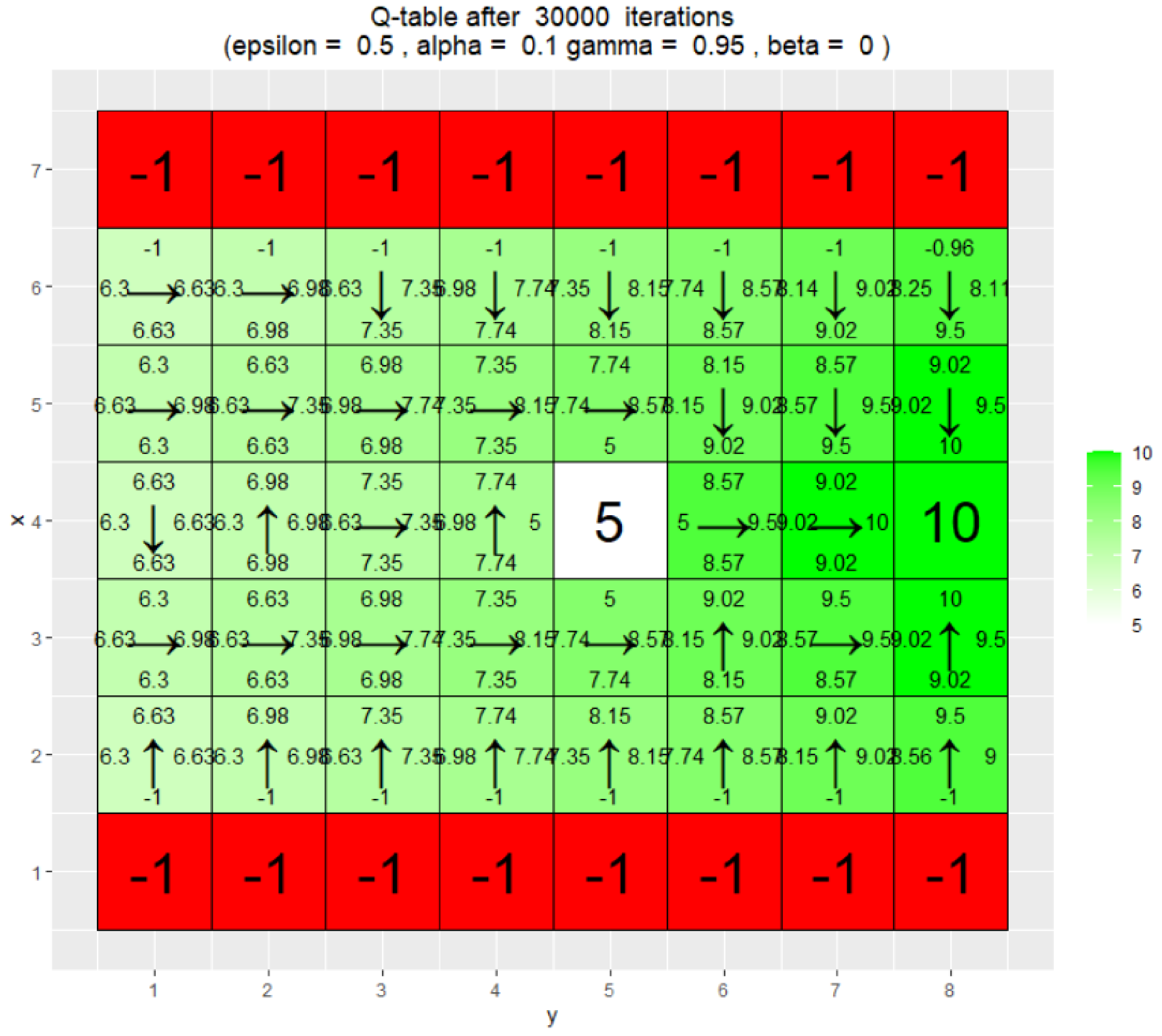

Figure 4: $\gamma = 0.95, \epsilon = 0.5$

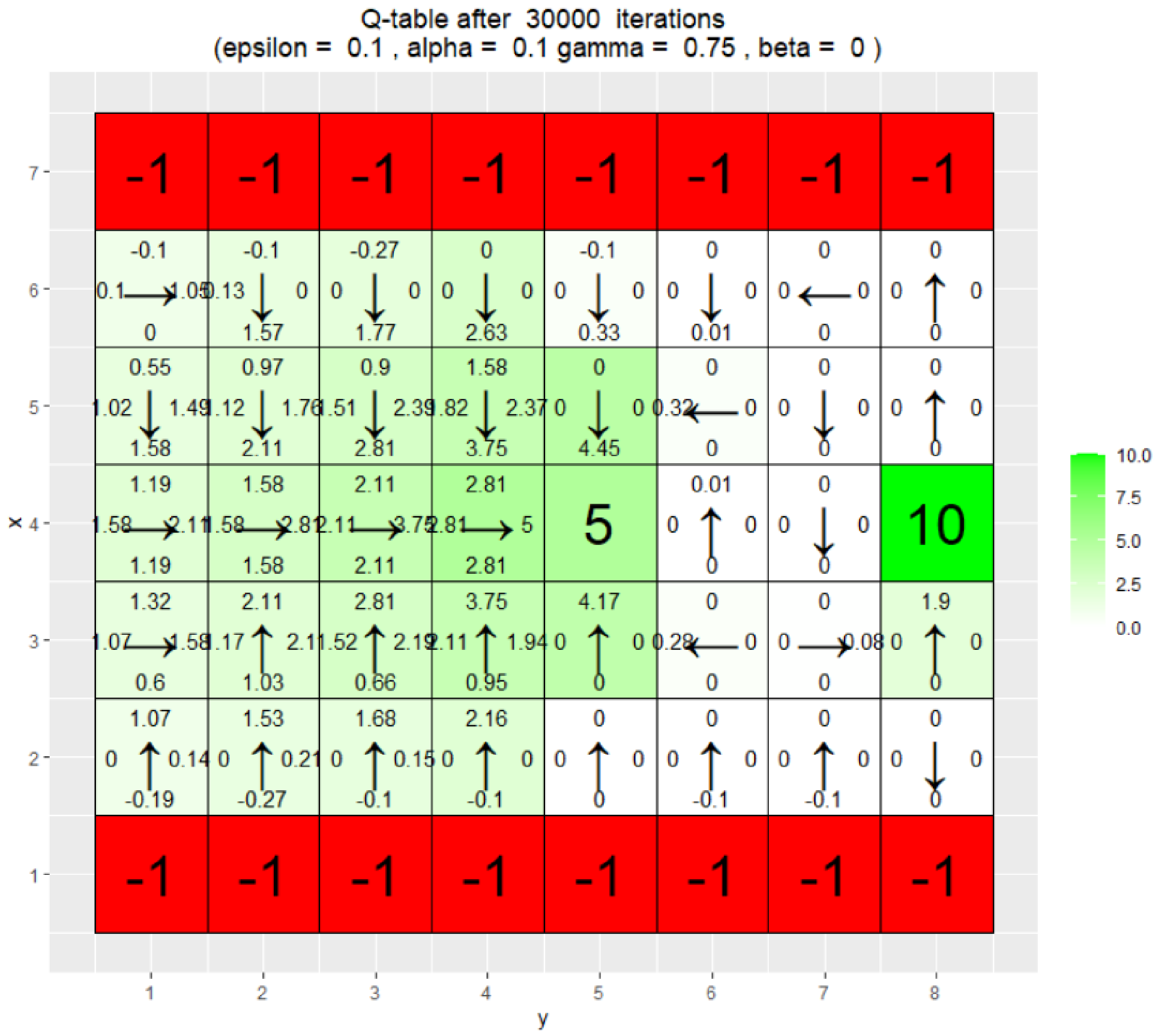

Figure 5: $\gamma = 0.75, \epsilon = 0.1$

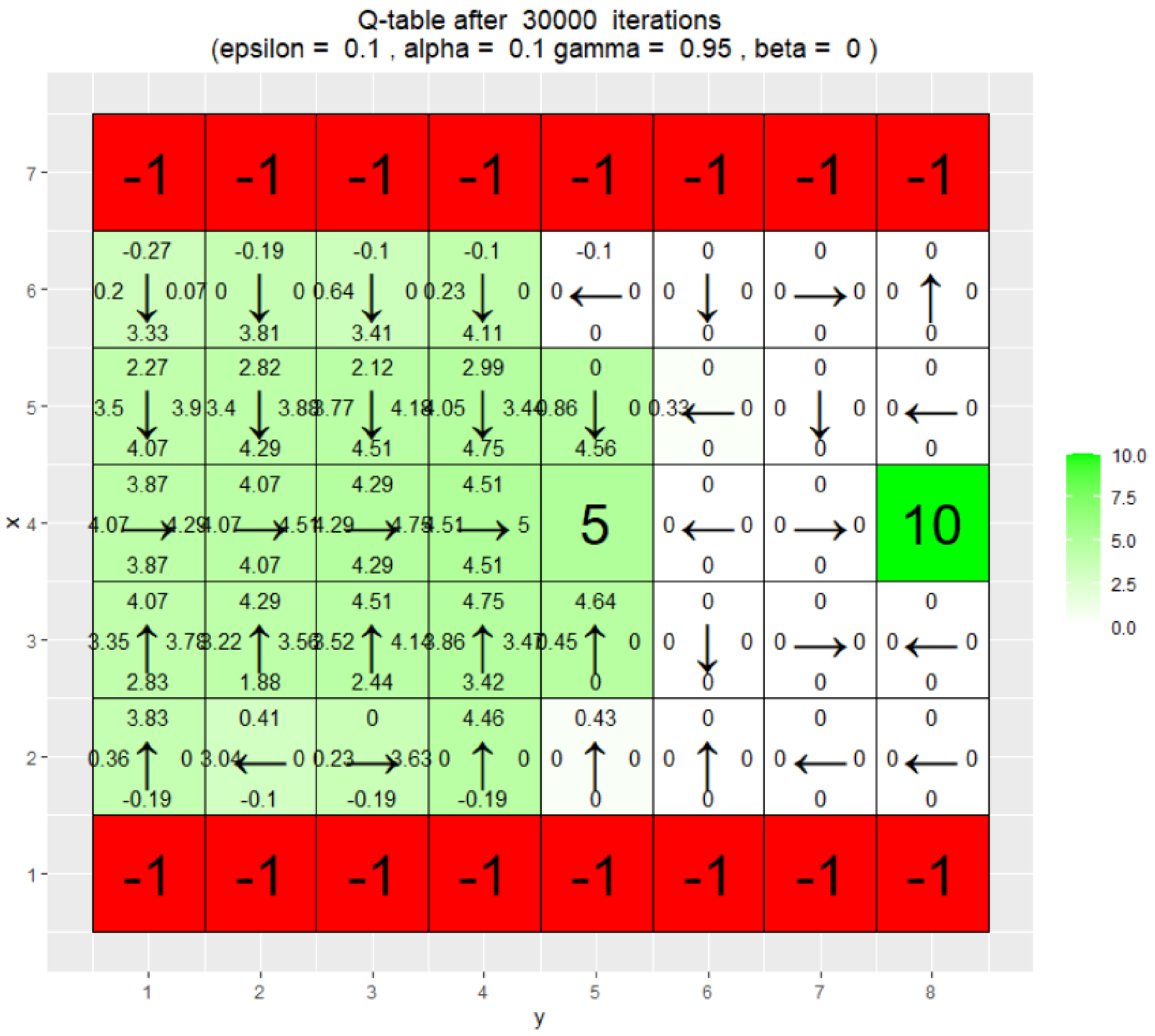

Figure 6: $\gamma = 0.95, \epsilon = 0.1$

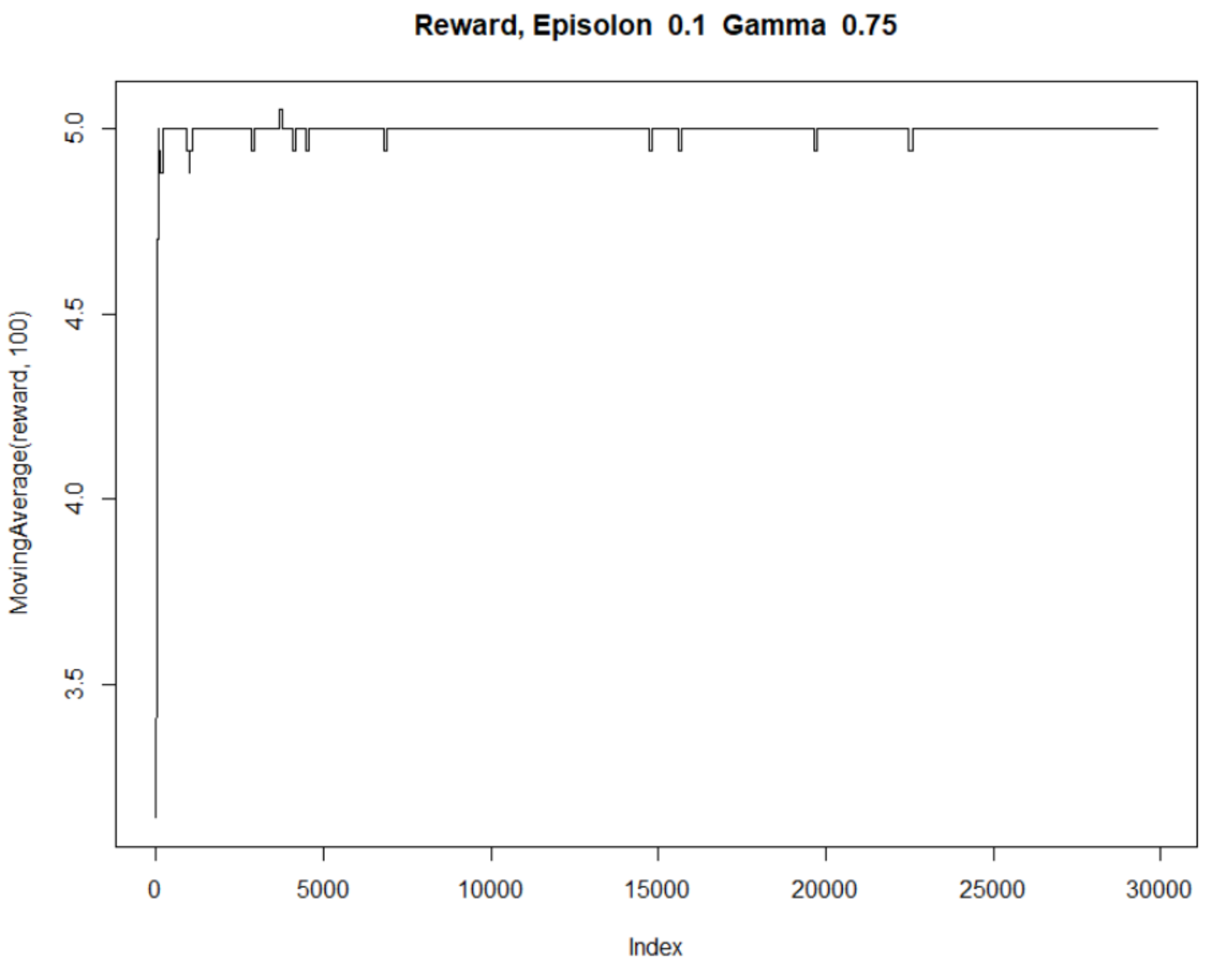

Figure 7: $\gamma = 0.75, \epsilon = 0.1$

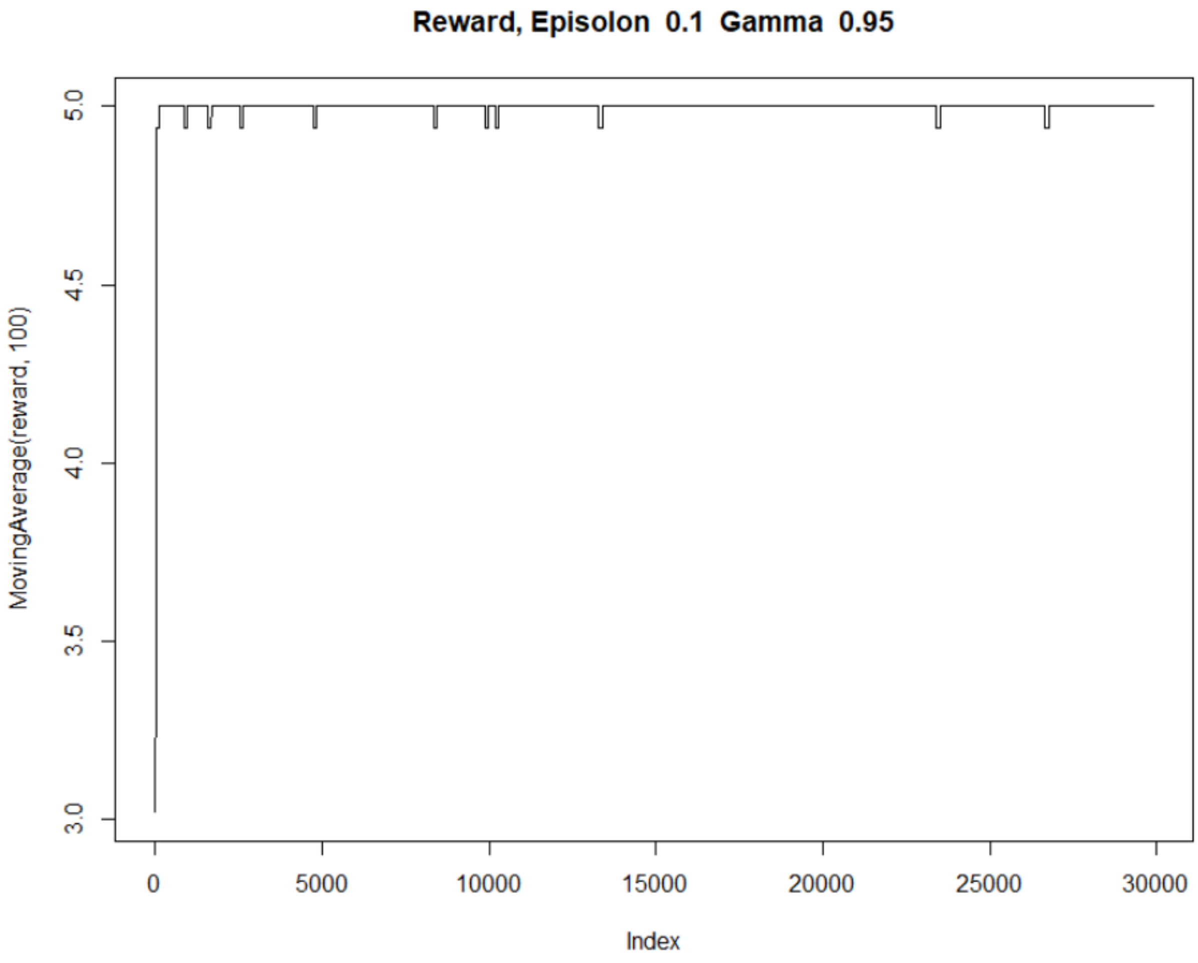

Figure 8: $\gamma = 0.95, \epsilon = 0.1$

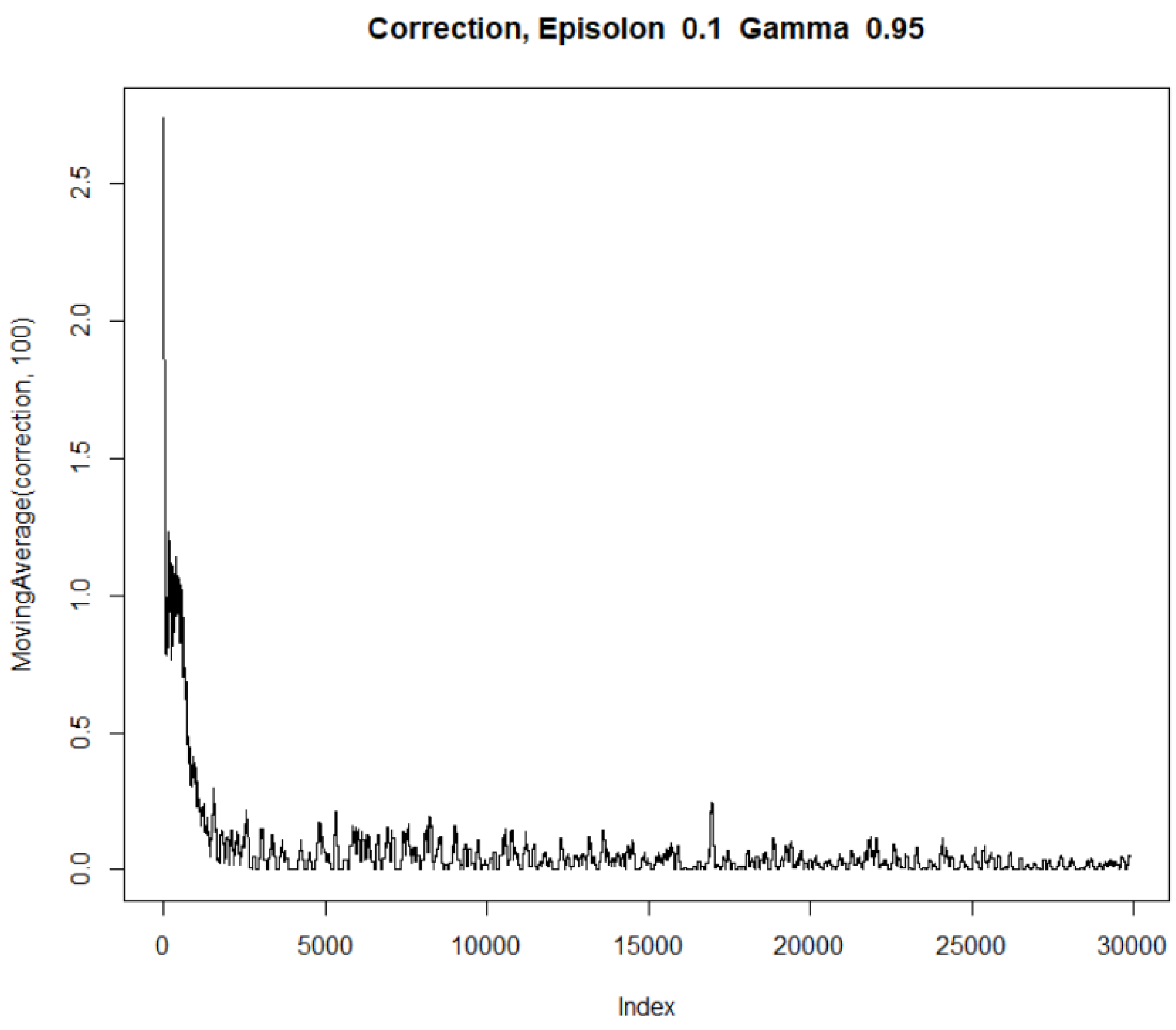

Figure 9: $\gamma = 0.95, \epsilon = 0.1$

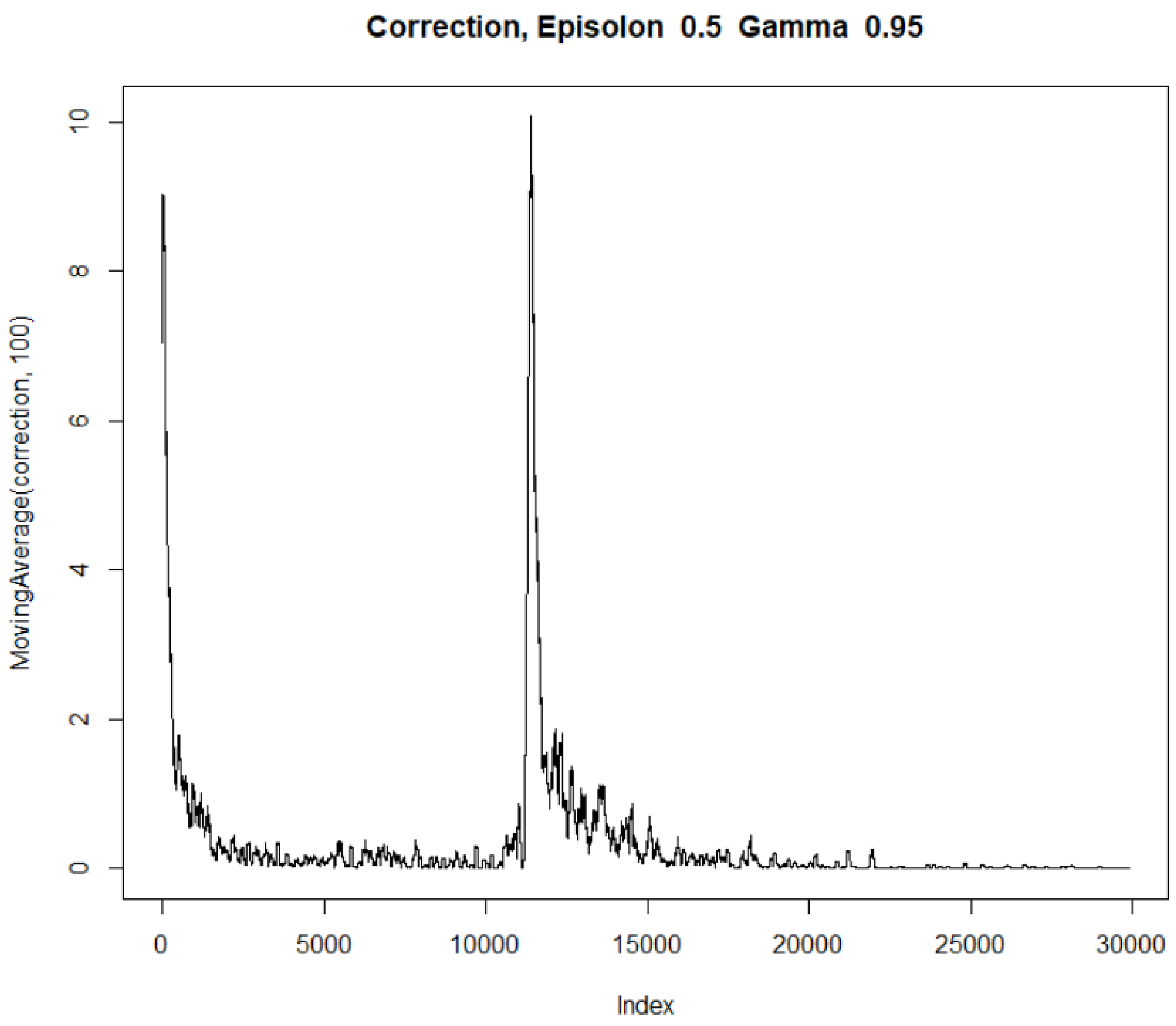

Figure 10: $\gamma = 0.95, \epsilon = 0.5$

Environment C

β is the slipping factor. The slipping factor can make the action "slip" and make the action be something different. The β is the probability of the agent slipping to the side when trying to move with each step. So as one can see in the examples below - the higher the β the higher is the probability that the agent takes the path where $x = 3$ as there is a smaller chance of it slipping in to the negative reward.

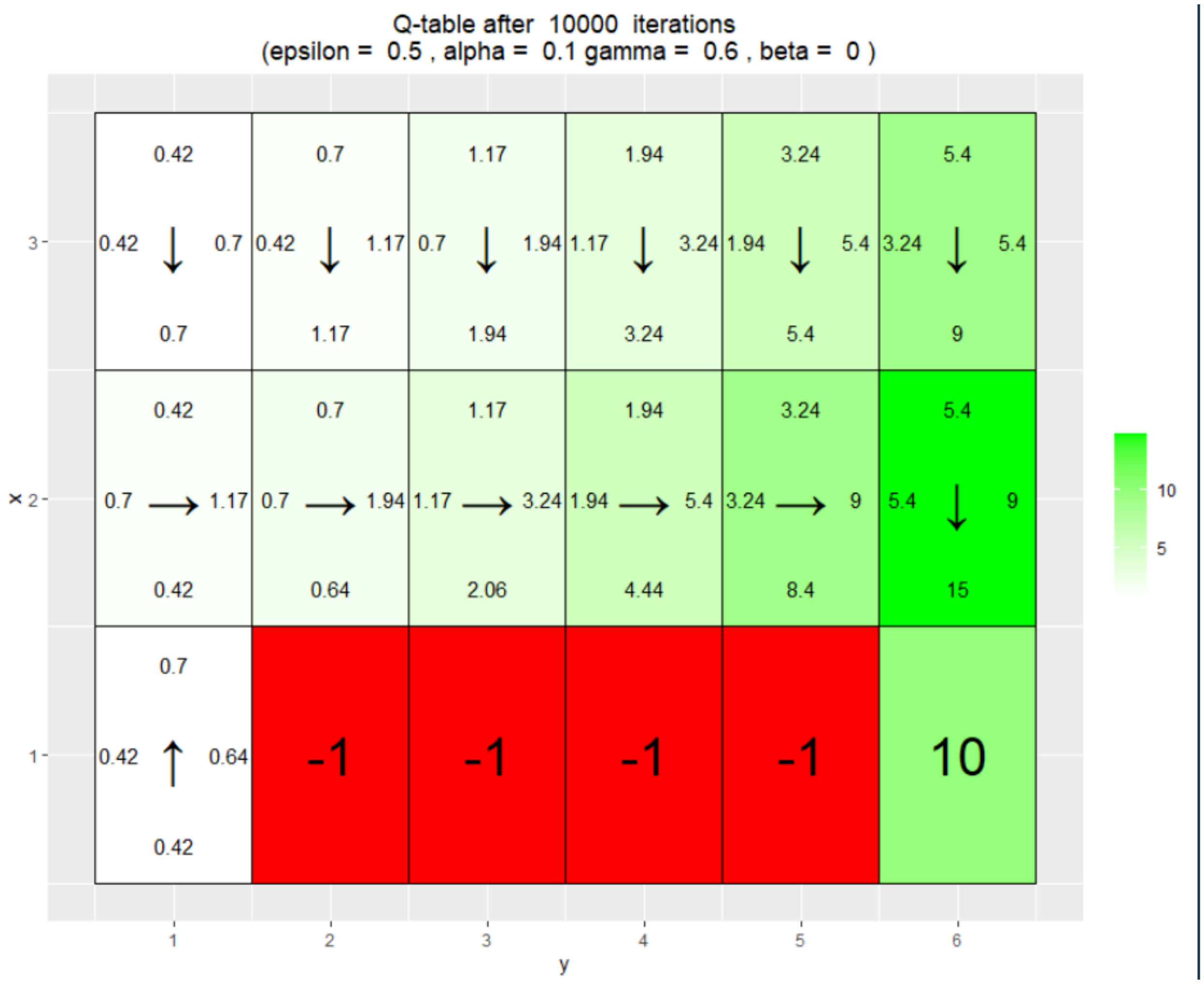

Figure 11: $\beta = 0$

Environment D

- *Has the agent learned a good policy? Why / Why not?*

Yes, the agent has learned a good policy as in almost all of the cases the agent would end up at the goal no matter what the starting position would be.

Example:

Example with one bad probability in $(1, 4)$:

- *Could you have used the Q-learning algorithm to solve this task?*

No, it would not work as the goals move, which Q-learning would not be able to handle.

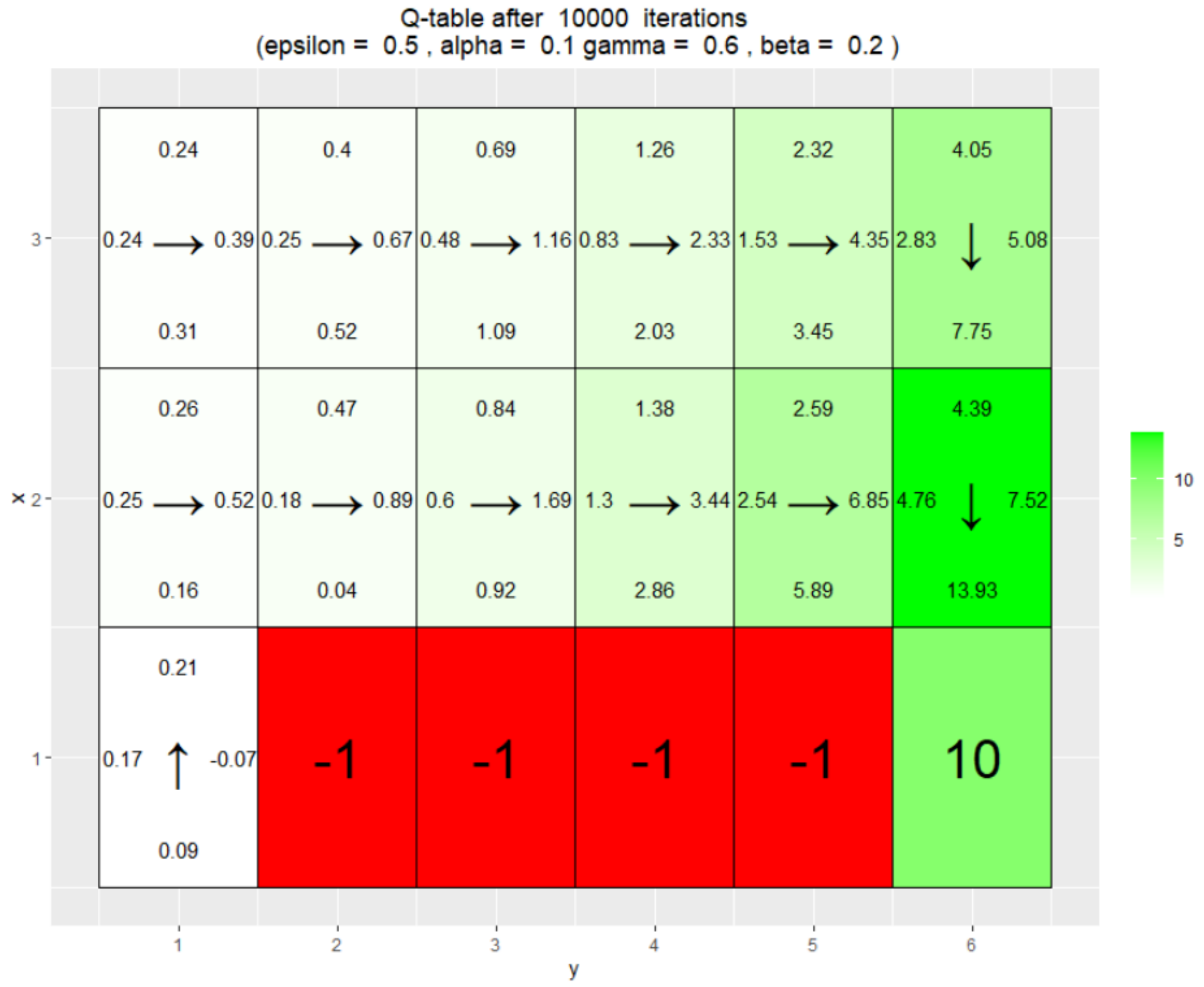

Figure 12: $\beta = 0_2$

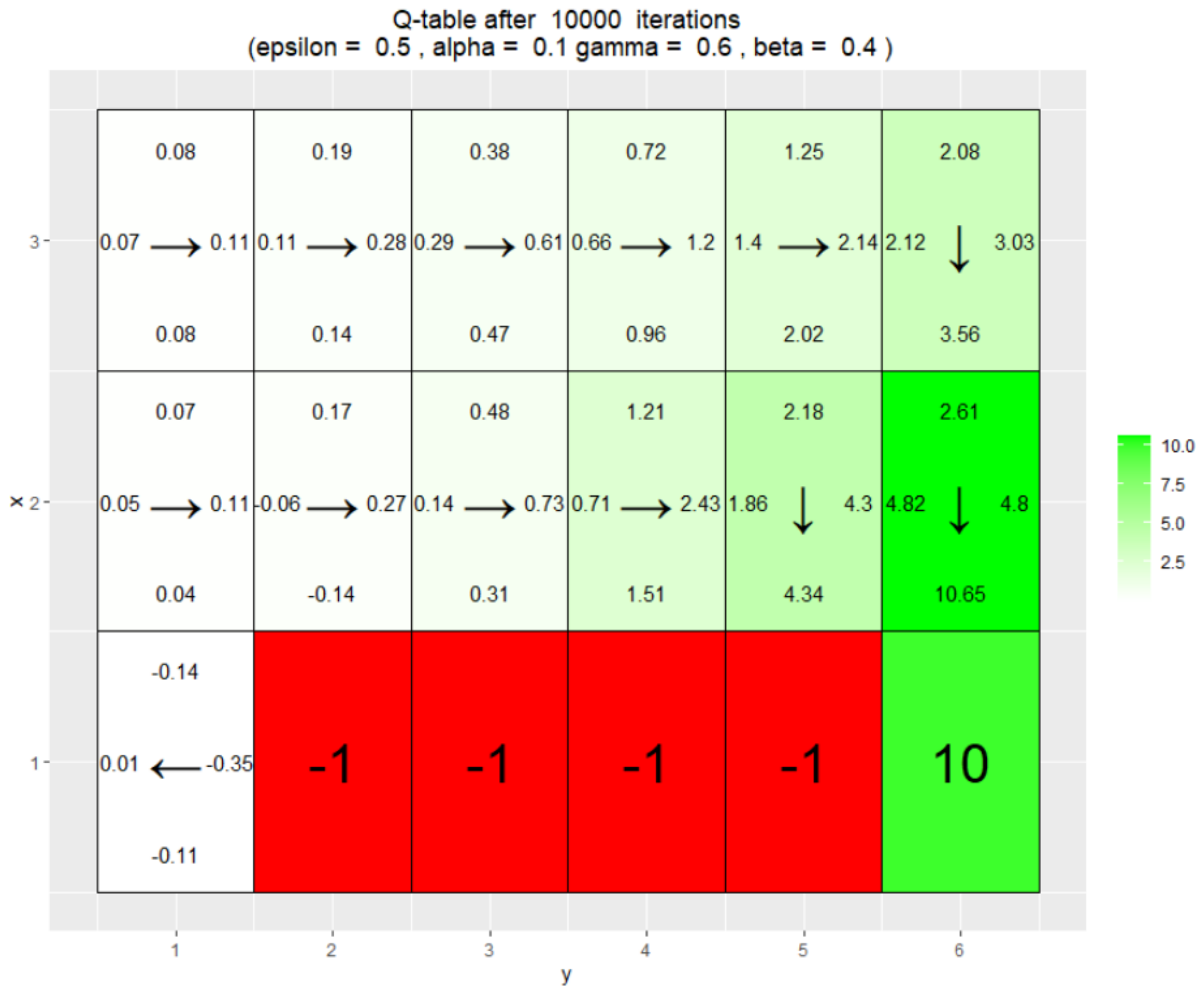

Figure 13: $\beta = 0_4$

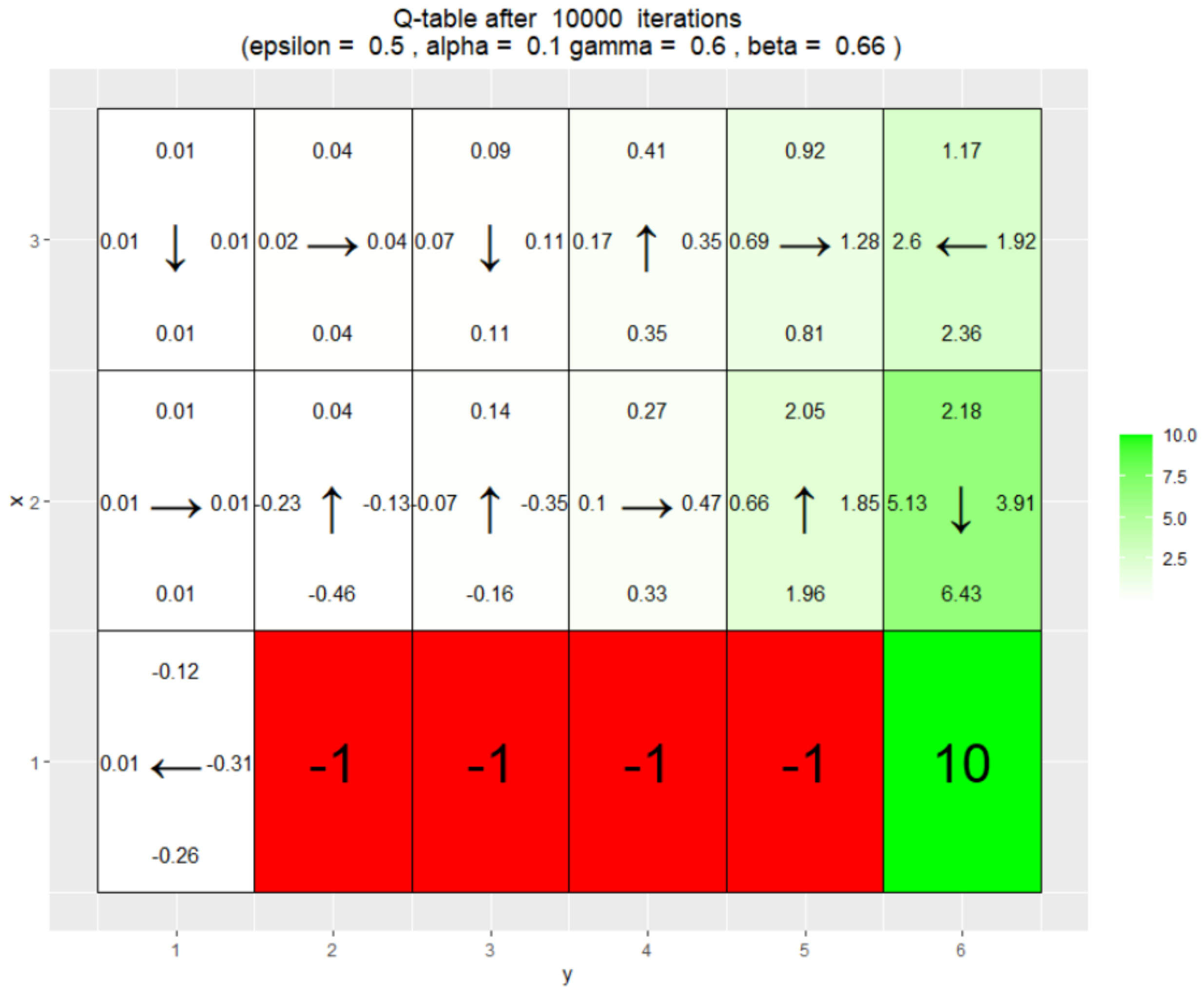

Figure 14: $\beta = 0_66$

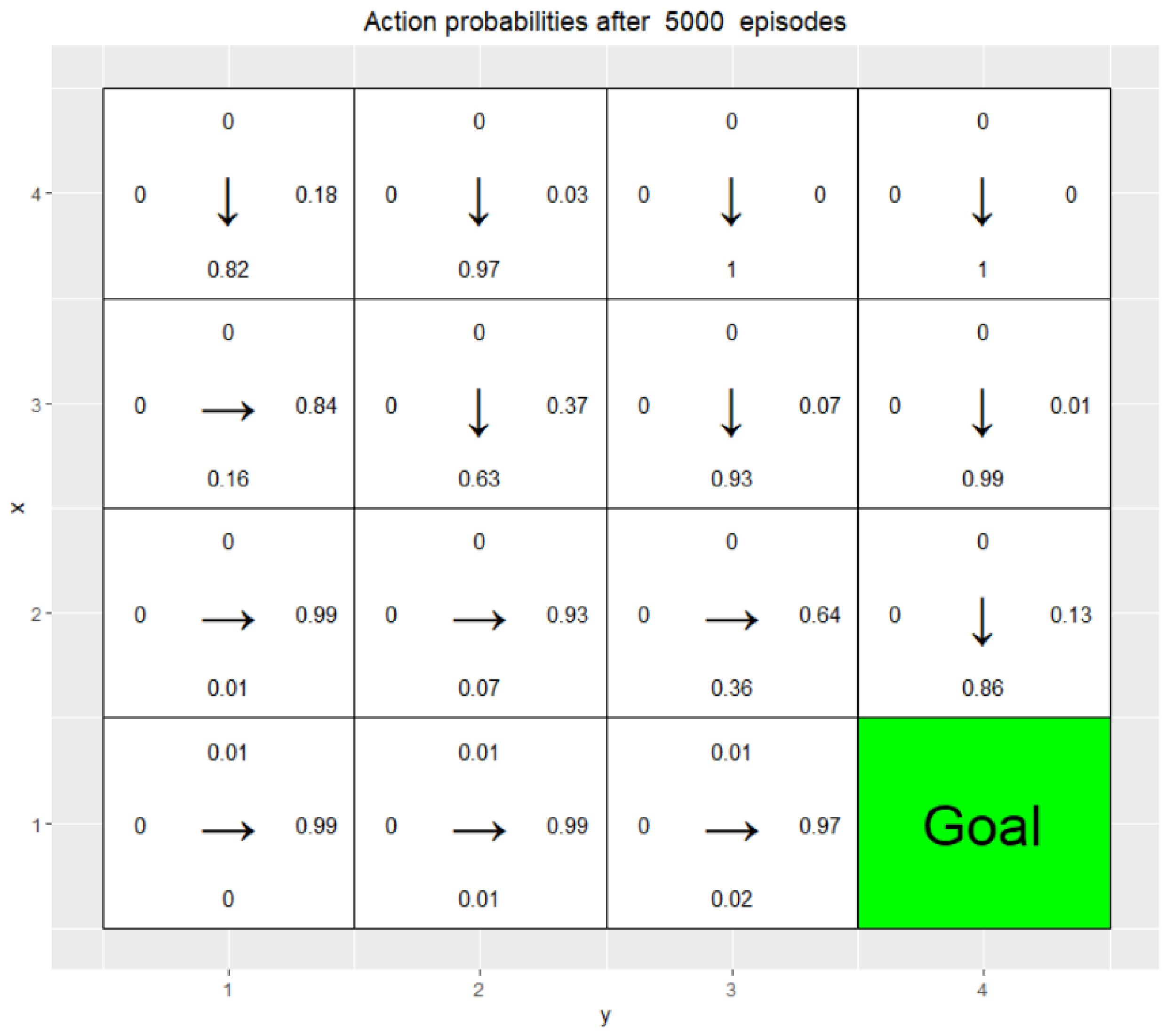

Figure 15: Reinforce with goal in $(1, 4)$

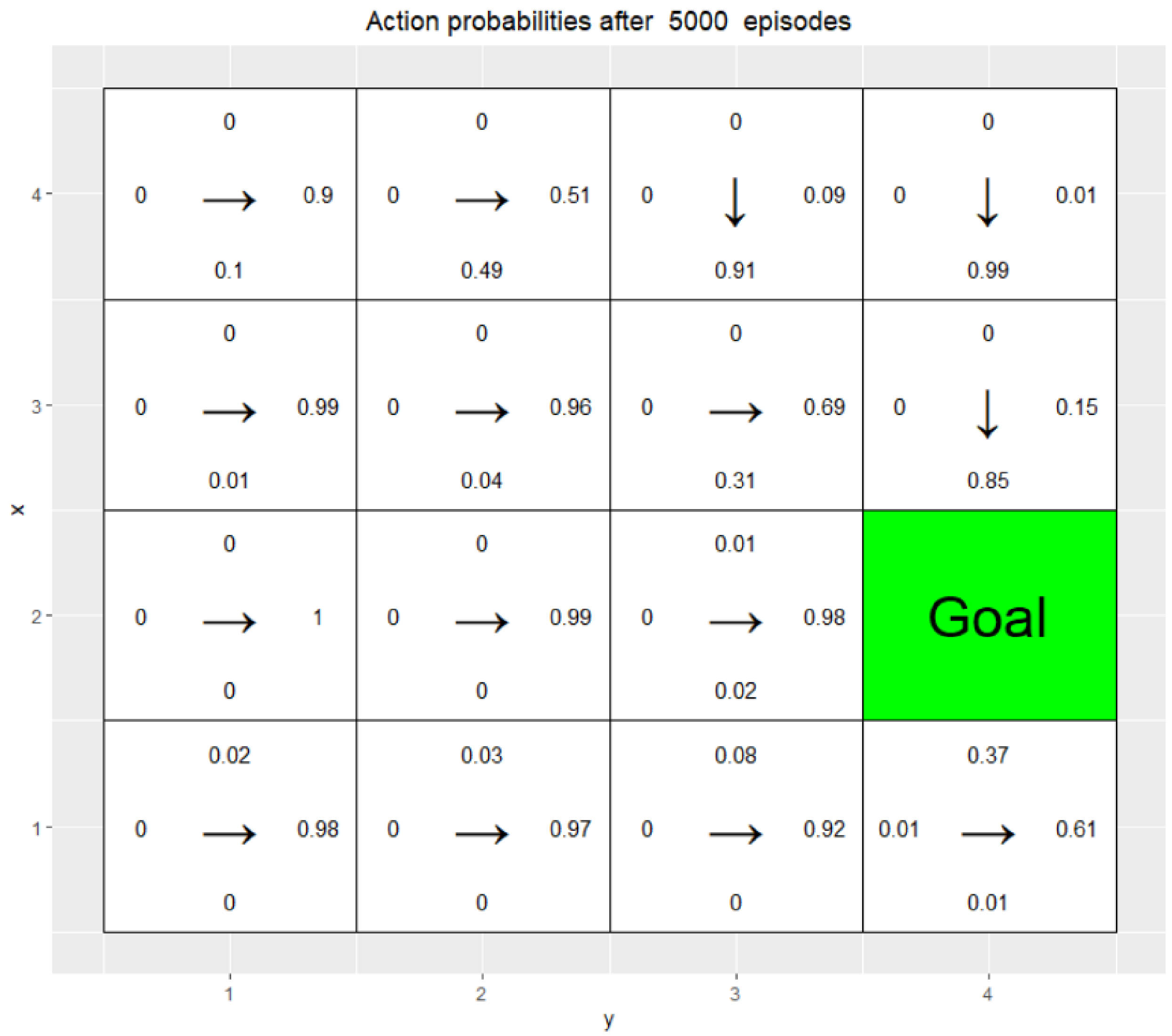

Figure 16: Reinforce with goal in $(2, 4)$

Environment E

- *Has the agent learned a good policy? Why / Why not?*

The policy is worse than in Environment D as there are a lot more cases where if the agent where placed in a random tile in the grid they would not end up at the goal. The reason for this is that it learns to just move generally from the top row to to the goal, which does not work as the goal can be in any of the bottom tiles.

Example of a bad policy where the agent would end up almost stuck at the right edge most of the time:

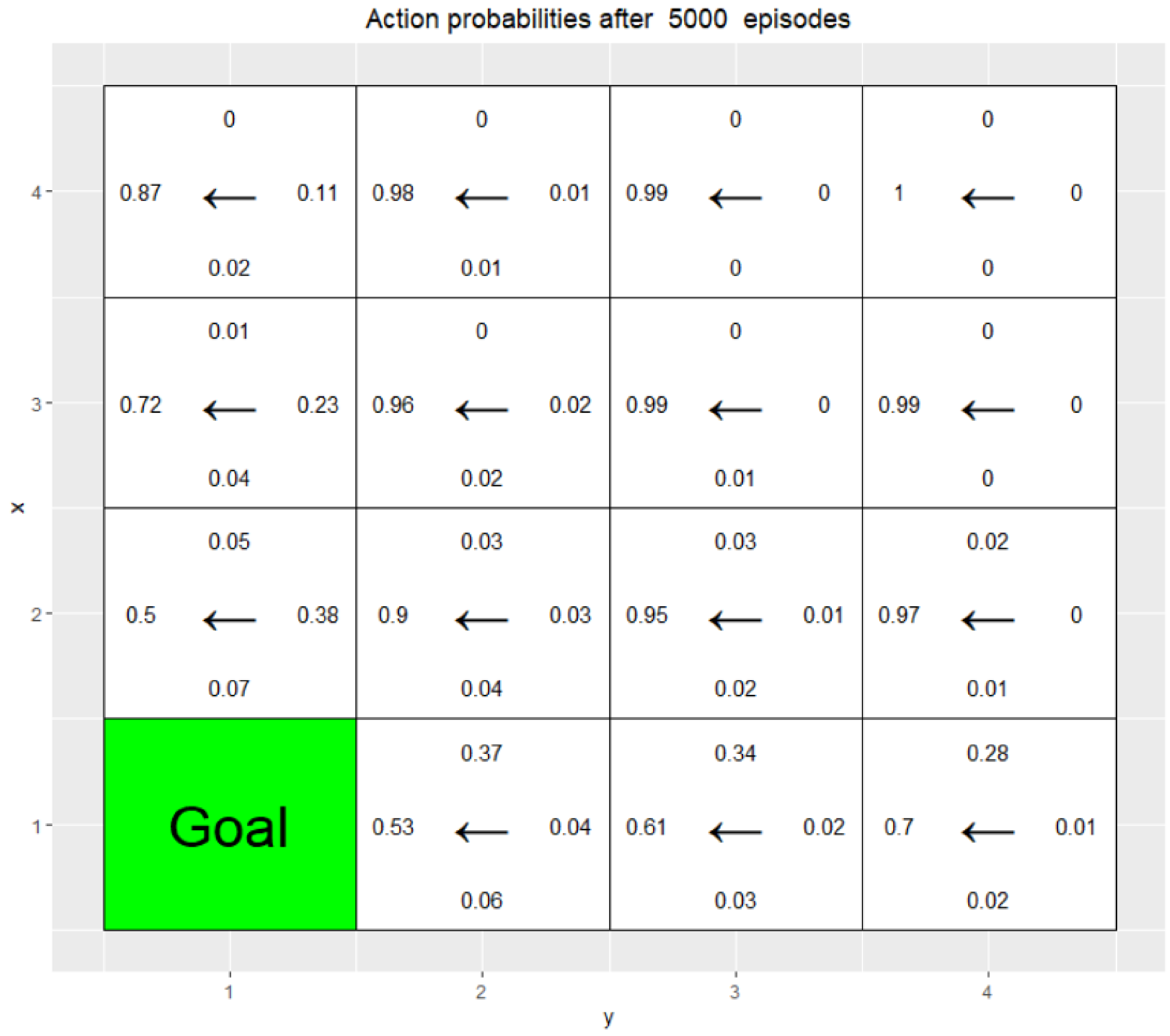

Figure 17: Reinforce with goal in $(1, 1)$

- *If the results obtained for environments D and E differ, explain why.*

The results differ for the reason stated in the question above.

Appendix for code

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

### MODIFIED BY AXEL HOLMBERG (axeho681@student.liu.se) AT "YOUR CODE HERE" ###

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

# 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("↑", "→", "↓", "←")
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}.

  # Your code here.

  max <-which(q_table[x,y,] == max(q_table[x,y,]))
  index <- sample(1:length(max),1)

  return(max[index])
}

EpsilonGreedyPolicy <- function(x, y, epsilon){

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

  # Your code here.


  if(runif(1,0,1)<epsilon) return(sample(1:4,1))

  action <- GreedyPolicy(x,y)

  return(action)
```

```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 <<-.

    # Your code here
    state <- start_state
    episode_correction <- 0
    repeat {
      # Follow policy, execute action, get reward.
      x <- state[1]
      y <- state[2]

     # action <- GreedyPolicy(x, y)
      action <- EpsilonGreedyPolicy(x, y,epsilon)

      new_state <- transition_model(x, y, action, beta)

      # Q-table update.
      reward <- reward_map[new_state[1], new_state[2]]

      tmp_episode_correction <- reward + gamma * max(q_table[new_state[1], new_state[2] , ]) - q_table[

      episode_correction <- episode_correction + tmp_episode_correction



      q_table[x, y, action] <<-
        q_table[x, y, action] + alpha * (tmp_episode_correction)


      state <- new_state


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

    }

  }

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

# 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))

vis_environment()
```

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

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

# Environment B (the effect of epsilon and gamma)

H <- 7
W <- 8

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

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)){
  #Comparing gamma
  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=paste("Reward, Episolon ", 0.5, " Gamma ", j))
  plot(MovingAverage(correction,100),type = "l", main=paste("Correction, Episolon ", 0.5, " Gamma ", j)
}

for(j in c(0.5,0.75,0.95)){
  #Comparing
  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=paste("Reward, Episolon ", 0.1, " Gamma ", j))
  plot(MovingAverage(correction,100),type = "l",main=paste("Correction, Episolon ", 0.1, " Gamma ", j))
}

# 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)
}
```

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

#####################################################################################################
# REINFORCE
#####################################################################################################

# install.packages("keras")
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("↑", "→", "↓", "←")
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)))
```

```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)
}

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
    }
  }

}

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

# Environment D (training with random goal positions)

H <- 4
W <- 4

# 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)
```

```r
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)
```