# Lab3-Personal report

Arash

2023-09-19

## Question 2.1

We are given an MDP template for a grid world, and we are asked to complete it. Notice, a seed is set to be able to reproduce the report.

```
# set.seed(1378)
set.seed(1111)
```

```
library(ggplot2)

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

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

  # Visualize an environment with rewards.
  # Q-values for all actions are displayed on the edges of each tile.
  # The (greedy) policy for each state is also displayed.
  #
  # Args:
  #   iterations, epsilon, alpha, gamma, beta (optional): for the figure title.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #   H, W (global variables): environment dimensions.

  df <- expand.grid(x=1:H,y=1:W)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,1],NA),df$x,df$y)
  df$val1 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,2],NA),df$x,df$y)
  df$val2 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,3],NA),df$x,df$y)
  df$val3 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,4],NA),df$x,df$y)
  df$val4 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y)
    ifelse(reward_map[x,y] == 0,arrows[GreedyPolicy(x,y)],reward_map[x,y]),df$x,df$y)
  df$val5 <- as.vector(foo)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,max(q_table[x,y,]),
                                     ifelse(reward_map[x,y]<0,NA,reward_map[x,y])),df$x,df$y)
  df$val6 <- as.vector(foo)
```

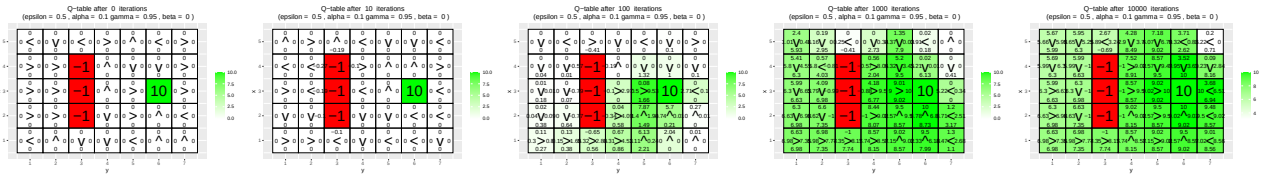
```
  print(ggplot(df,aes(x = y,y = x)) +
          scale_fill_gradient(low = "white", high = "green", na.value = "red", name = "") +
          geom_tile(aes(fill=val6)) +
          geom_text(aes(label = val1),size = 4,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 4,nudge_x = .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}.
  max_value <- max(q_table[x, y, ])
  action <- which(q_table[x, y, ] == max_value)
  if(length(action) != 1)
    action <- sample(action, 1)

  return(action)

}

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

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

  if(runif(1) < epsilon){
    action <- sample(1:4, 1)
  }else{
    action <- GreedyPolicy(x, y)
  }
  return(action)
```

```
}

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

  # Computes the new state after given action is taken. The agent will follow the action
  # with probability (1-beta) and slip to the right or left with probability beta/2 each.
  #
  # Args:
  #   x, y: state coordinates.
  #   action: which action the agent takes (in {1,2,3,4}).
  #   beta: probability of the agent slipping to the side when trying to move.
  #   H, W (global variables): environment dimensions.
  #
  # Returns:
  #   The new state after the action has been taken.

  delta <- sample(-1:1, size = 1, prob = c(0.5*beta,1-beta,0.5*beta))
  final_action <- ((action + delta + 3) %% 4) + 1
  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 <<-.

  current_state <- start_state
  episode_correction <- 0
  repeat{
    x <- current_state[1]
    y <- current_state[2]
    # Follow policy, execute action, get reward.
    a <- EpsilonGreedyPolicy(x, y, epsilon)
```

```
    next_state <- transition_model(x, y, a, beta)
    reward <- reward_map[next_state[1], next_state[2]]
    # Q-table update.
    correction <- reward + gamma * max(q_table[next_state[1], next_state[2], ]) -  q_table[x, y, a]
    q_table[x, y, a] <<- q_table[x, y, a] + alpha * correction
    episode_correction <- episode_correction + correction
    current_state <- next_state
    if(reward!=0)
      # End episode.
      return (c(reward,episode_correction))
  }

}
```

## Question 2.2

```
# 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()


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

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



What has the agent learned after the first 10 episodes ?

Depending on the run, it learnt very little about the q values of the states adjacent to the terminal states.

Is the final greedy policy (after 10000 episodes) optimal for all states, i.e. not only for the initial state ? Why / Why not ?

Depending on the run, the policy is mostly optimal. The agent does not take actions that results in a negative reward, and it tries to traverse from anywhere in the grid to reach the +10 reward.

The Policy around the initial state is also seems optimal as any directions towards up, down or right is optimal. The reason that the may cause to have equal q values for initial state is that the agent did not

explore enough around the initial state and since at the start of the episode the reward is equal to zero, and agent gets no information about the initial state.

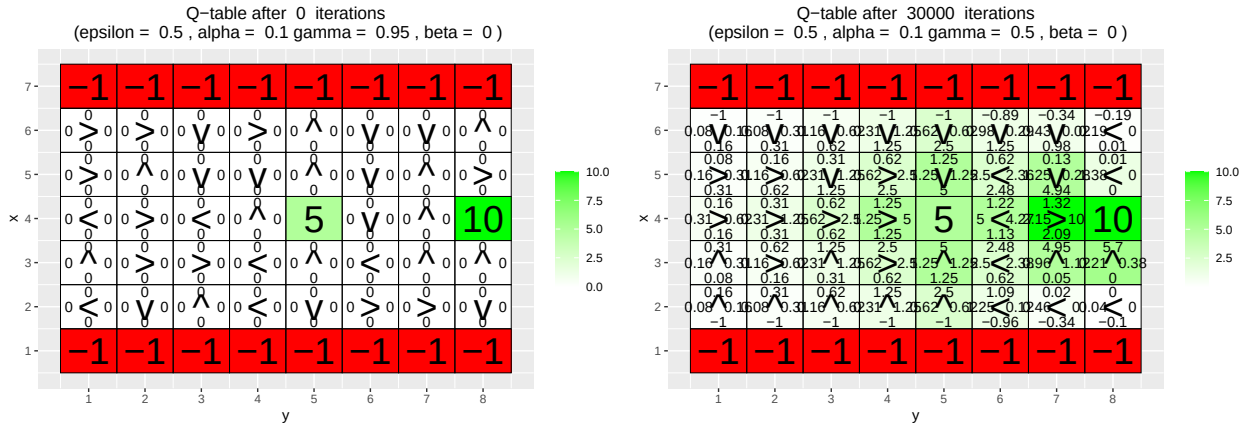
> Do the learned values in the Q-table reflect the fact that there are multiple paths (above and below the negative rewards) to get to the positive reward ? If not, what could be done to make it happen?

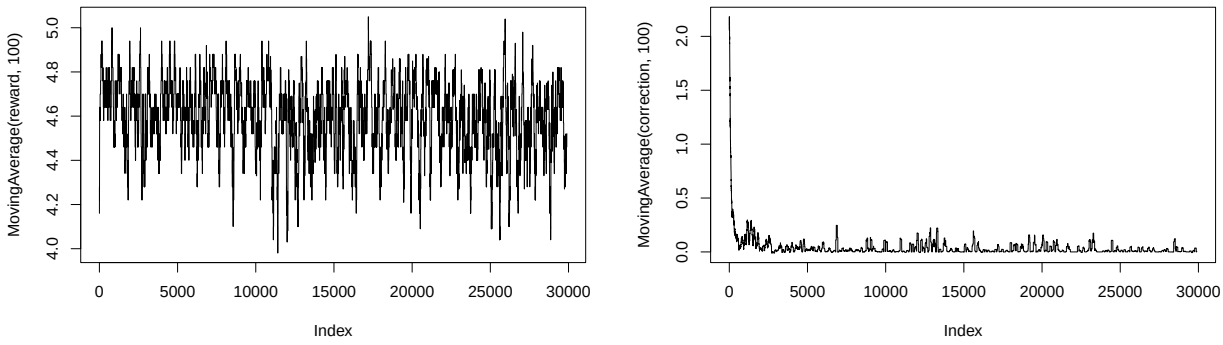
No. From each state around the negative rewards, agent can exactly have one path to get to the positive reward. The reason is that, MDP is deterministic and if agent act based on greedy policy, there will be a unique path for each given state to the positive reward.

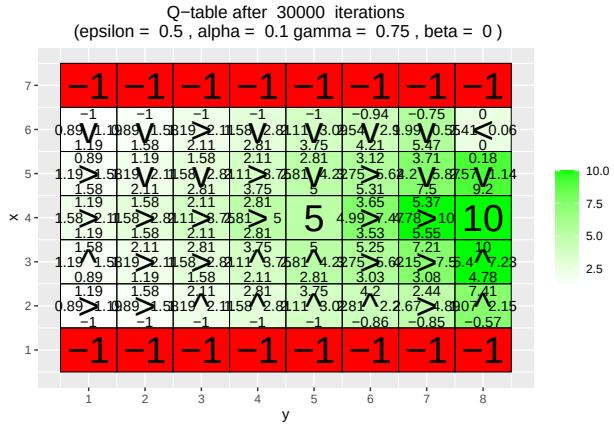
## Question 2.3

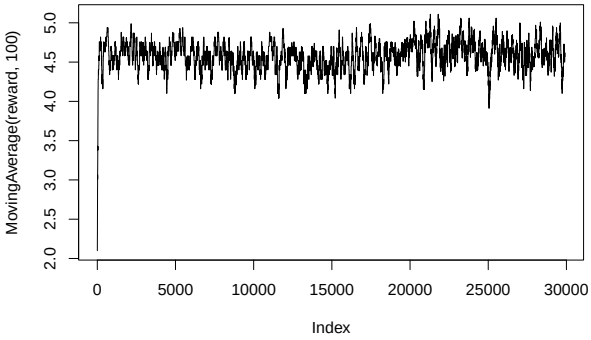
```
# Environment B (the effect of epsilon and gamma)

H <- 7
W <- 8

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

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

vis_environment()

MovingAverage <- function(x, n){

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

  return (rsum)
}

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

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

  vis_environment(i, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}

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

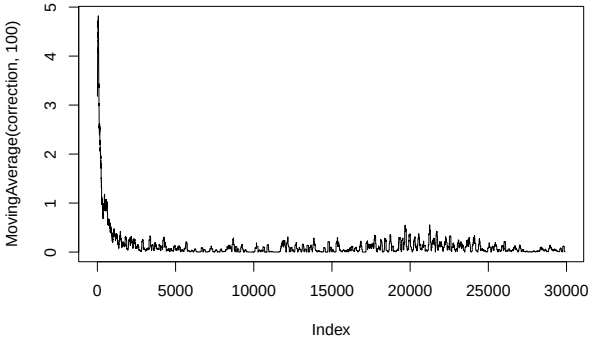
```
  reward <- NULL
  correction <- NULL

  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")
}
```

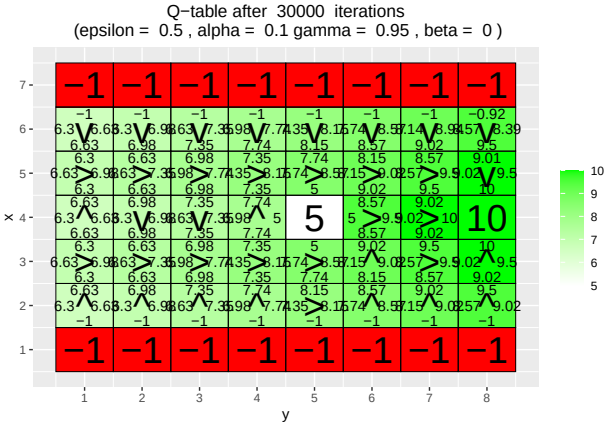


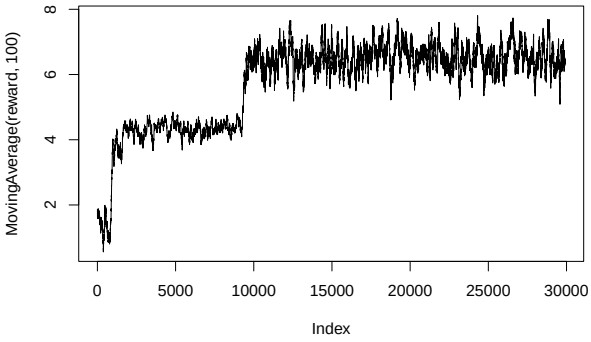



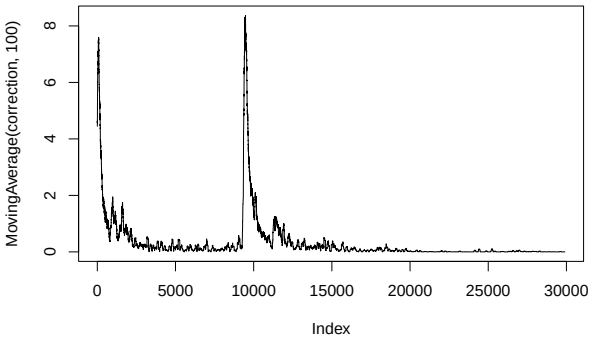

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0 )
x
y

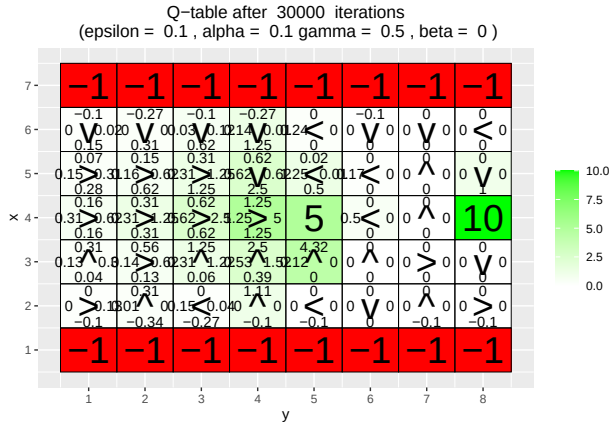

MovingAverage(reward, 100)
Index

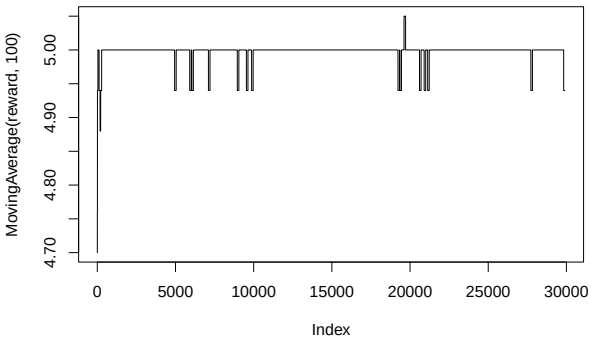

MovingAverage(correction, 100)
Index

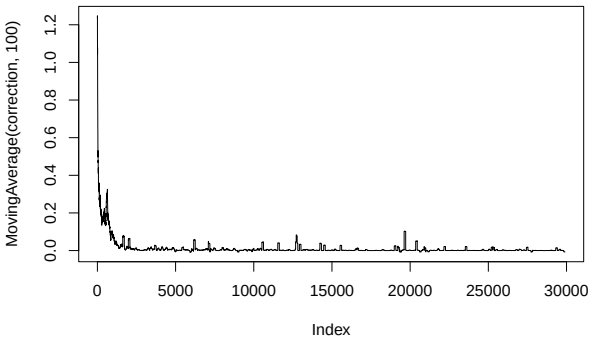

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0 )
x
y

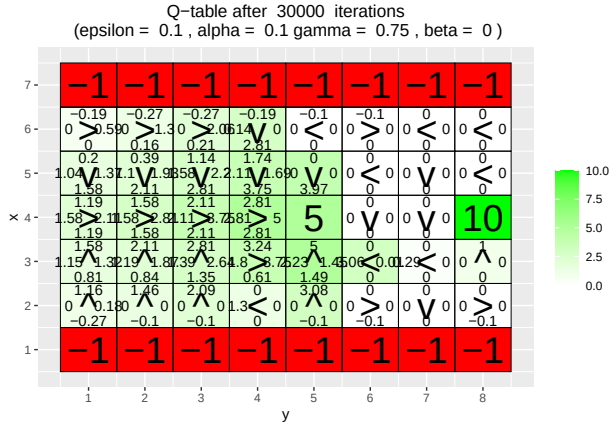

MovingAverage(reward, 100)
Index

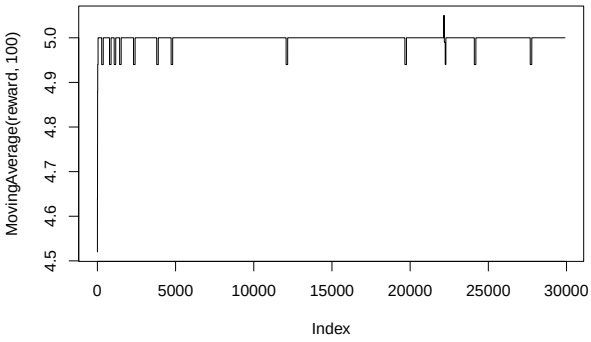

MovingAverage(correction, 100)
Index

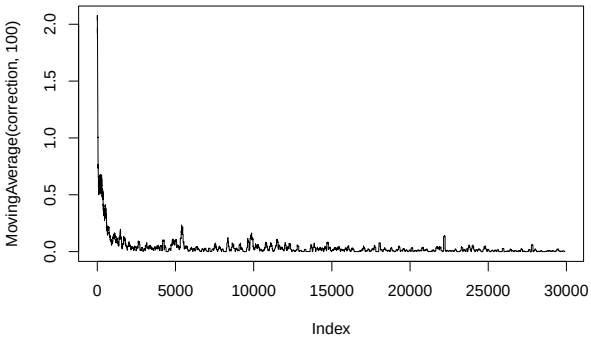

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.5 , beta = 0 )
x
y
10.0
7.5
5.0
2.5
0.0

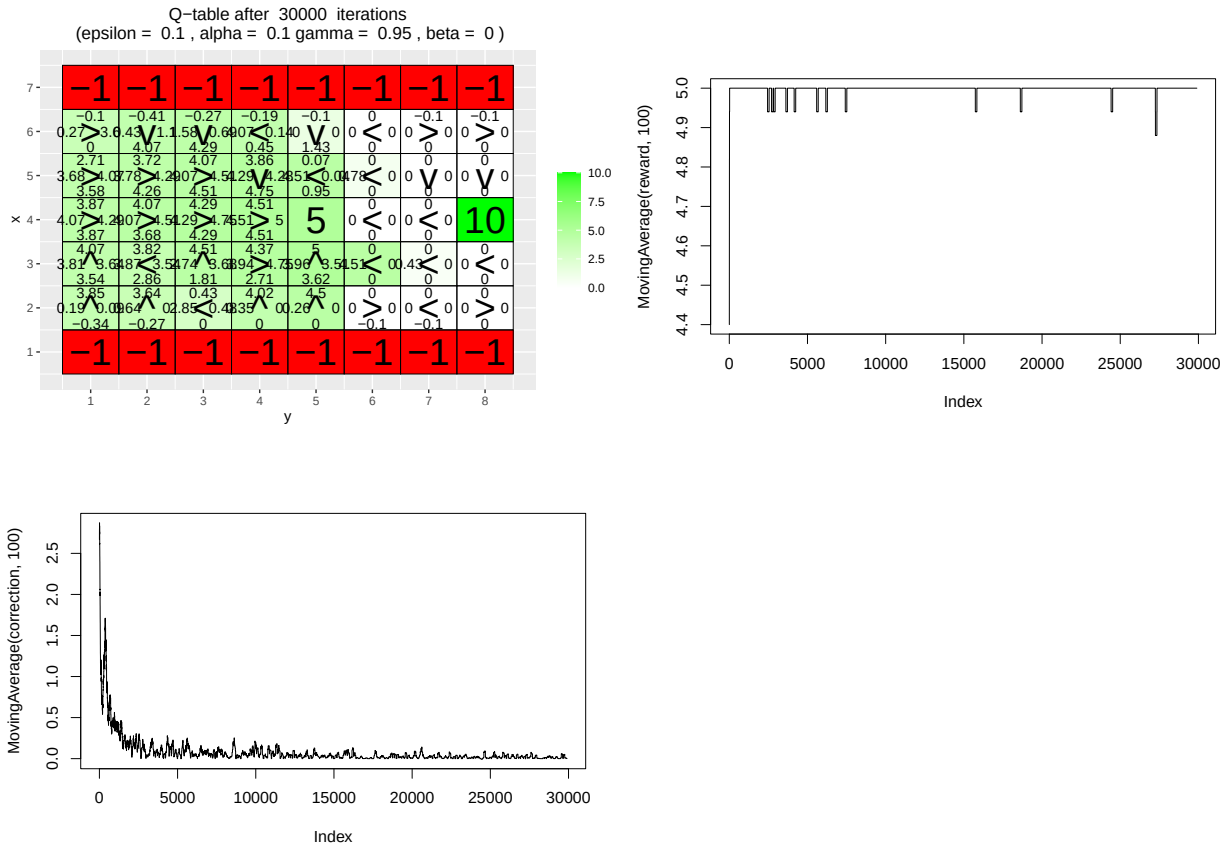

MovingAverage(reward, 100)
Index


MovingAverage(correction, 100)
Index


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


MovingAverage(reward, 100)
Index


MovingAverage(correction, 100)
Index




Based on the experiments above, we can see the effect of the $\epsilon$ and $\gamma$ on the agent and the policy that it learns.

**Regarding the $\gamma$:**

In the experiment with $\epsilon = 0.5$, we can observe that as we increase the the discounting factor, the agent tends to learn the policy that results in a higher expected return in a long-term. If we put $\gamma = 0.95$, it will result in a agent that tries to avoid not only the negative rewards, but also the positive 5 reward in the middle. In contrast, if $\gamma = 0.5$ or even $\gamma = 0.75$, the agent will get to the +5 reward if it is to the left, below or above that state.

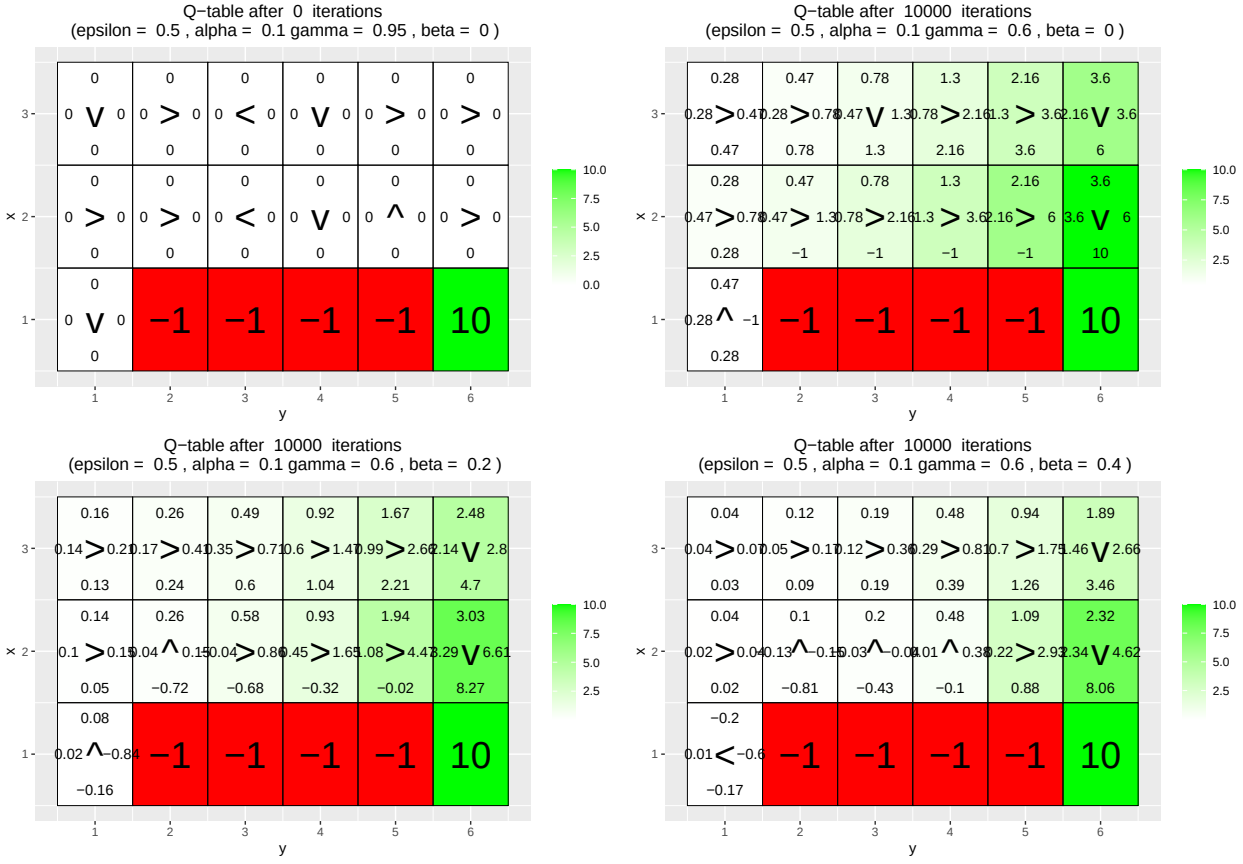
**Regarding the $\epsilon$:**

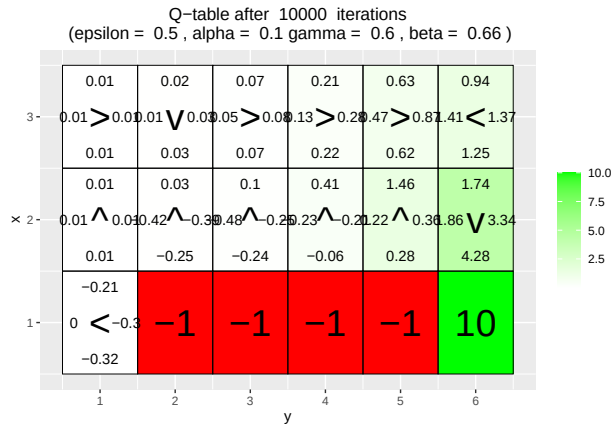
Although we now know the effect of $\gamma$ on the learnt policy, if agent does not explore the environment enough, it may result in a sub-optimal policy. The policies with all the different values of $\gamma$ are resembling the optimal policies, but they do not have an optimal policy for some states close to the lower border. Also, the lack of exploration results in a Q table that does not have optimal values for the states close to the lower border.

## Question 2.4

```
# 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)
}
```






In case of a stochastic environment, the policy tries to generate an action that avoid agent to fall in to the negative rewards on the bottom edge of the map.

For instance, when $\beta = 0.2$ or $\beta = 0.4$, agent decides to go up in states (2,2) and (2,3) and even (2,4). In case of $\beta = 0.6$, the agent even tries to go to the right even when $y = 6$ so that it may by chance goes to the goal state.

In general, as the $\beta$ parameter gets bigger, the agent can not get a good signal (it gets a noisy signal) for the chosen actions in each state;therefore, The policy may not be optimal (specially if $\beta$ is big enough).

## Question 2.5

```
reticulate::use_condaenv("tbi26")
```

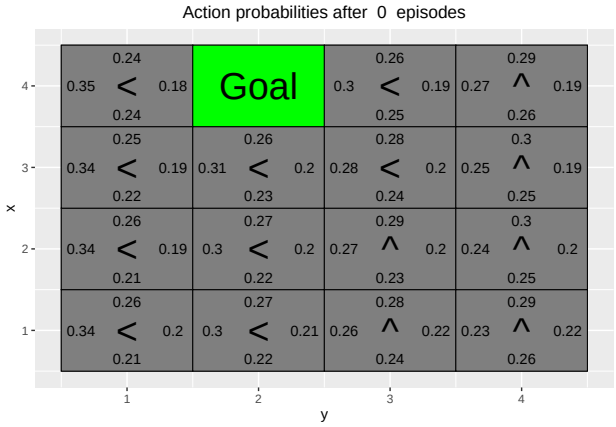
```
# By Jose M. Peña and Joel Oskarsson.
# For teaching purposes.
# jose.m.pena@liu.se.

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

# install.packages("keras")
library(keras)

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

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.
  #
```

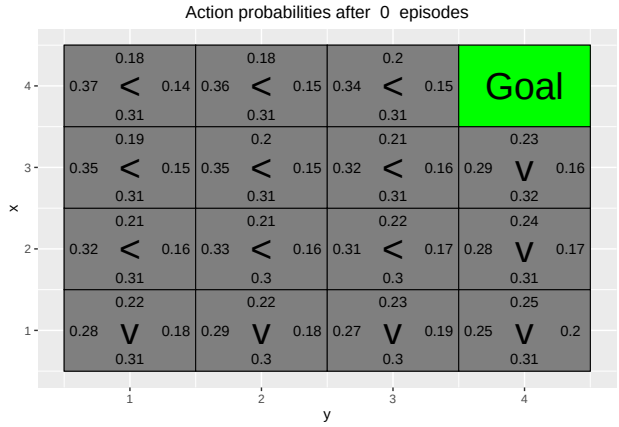
```r
  # Args:
  #   goal: goal coordinates, array with 2 entries.
  #   episodes, epsilon, alpha, gamma, beta (optional): for the figure title.
  #   H, W (global variables): environment dimensions.

  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}).
```

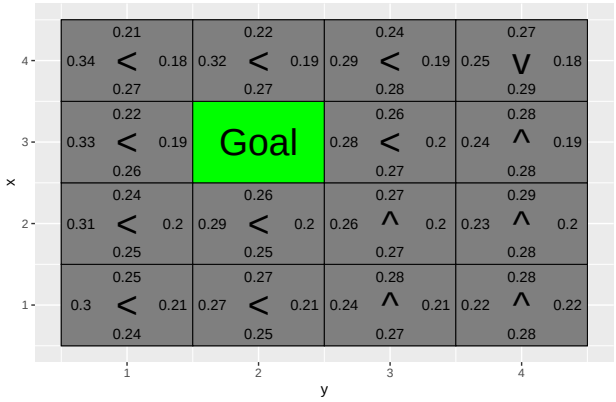
```
  #   beta: probability of the agent slipping to the side when trying to move.
  #   H, W (global variables): environment dimensions.
  #
  # Returns:
  #   The new state after the action has been taken.

  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){
```

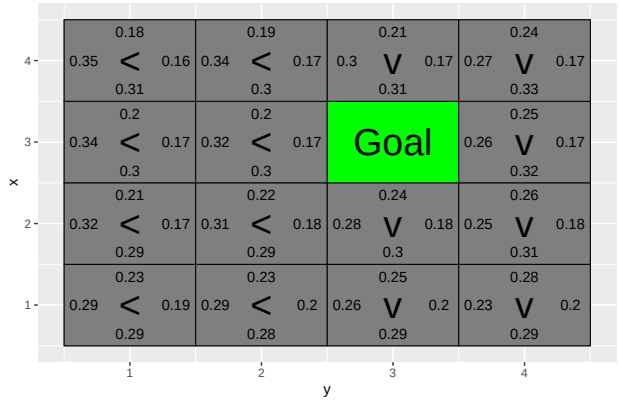
```r
  # Train the policy network on a rolled out trajectory.
  #
  # Args:
  #   states: array of states visited throughout the trajectory.
  #   actions: array of actions taken throughout the trajectory.
  #   goal: goal coordinates, array with 2 entries.
  #   gamma: discount factor.

  # 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)
```

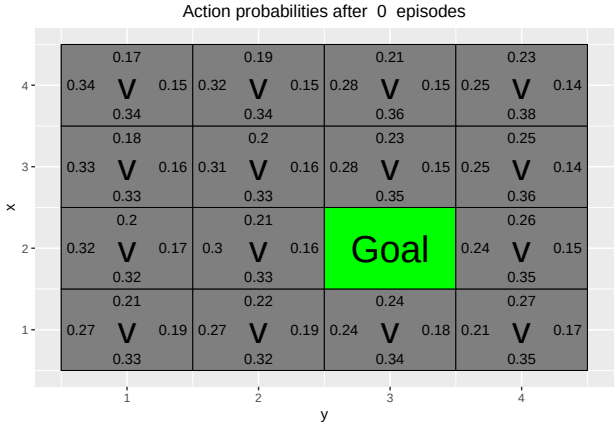
```
    cur_pos <- new_pos

    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)

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)
```

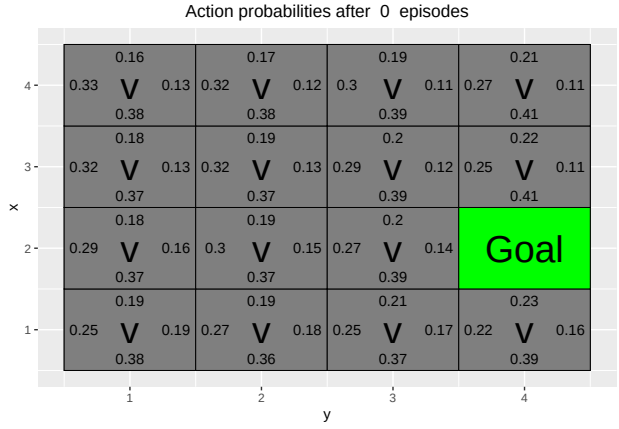

Action probabilities after 0 episodes
0.24 0.35 < 0.18 0.24
Goal
0.26 0.3 < 0.19 0.25
0.29 0.27 ^ 0.19 0.26
0.25 0.34 < 0.19 0.22
0.26 0.31 < 0.2 0.23
0.28 0.28 < 0.2 0.24
0.3 0.25 ^ 0.19 0.25
0.26 0.34 < 0.19 0.21
0.27 0.3 < 0.2 0.22
0.29 0.27 ^ 0.2 0.23
0.3 0.24 ^ 0.2 0.25
0.26 0.34 < 0.2 0.21
0.27 0.3 < 0.21 0.22
0.28 0.26 ^ 0.22 0.24
0.29 0.23 ^ 0.22 0.26
x
y
1
2
3
4

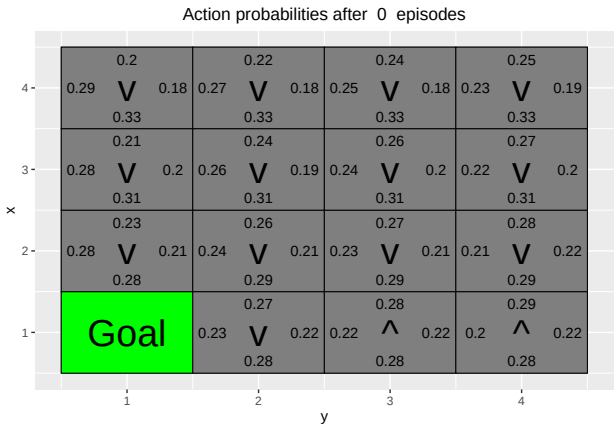

Action probabilities after 0 episodes
0.18 0.37 < 0.14 0.31
0.18 0.36 < 0.15 0.31
0.2 0.34 < 0.15 0.31
Goal
0.19 0.35 < 0.15 0.31
0.2 0.35 < 0.15 0.31
0.21 0.32 < 0.16 0.31
0.23 0.29 v 0.16 0.32
0.21 0.32 < 0.16 0.31
0.21 0.33 < 0.16 0.3
0.22 0.31 < 0.17 0.3
0.24 0.28 v 0.17 0.31
0.22 0.28 v 0.18 0.31
0.22 0.29 v 0.18 0.3
0.23 0.27 v 0.19 0.3
0.25 0.25 v 0.2 0.31
x
y
1
2
3
4

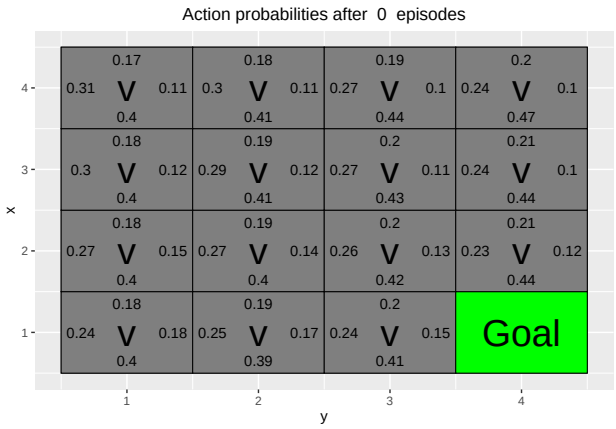

Action probabilities after 0 episodes
0.21 0.34 < 0.18 0.27
0.22 0.32 < 0.19 0.27
0.24 0.29 < 0.19 0.28
0.27 0.25 v 0.18 0.29
0.22 0.33 < 0.19 0.26
Goal
0.26 0.28 < 0.2 0.27
0.28 0.24 ^ 0.19 0.28
0.24 0.31 < 0.2 0.25
0.26 0.29 < 0.2 0.25
0.27 0.26 ^ 0.2 0.27
0.29 0.23 ^ 0.2 0.28
0.25 0.3 < 0.21 0.24
0.27 0.27 < 0.21 0.25
0.28 0.24 ^ 0.21 0.27
0.28 0.22 ^ 0.22 0.28
x
y
1
2
3
4

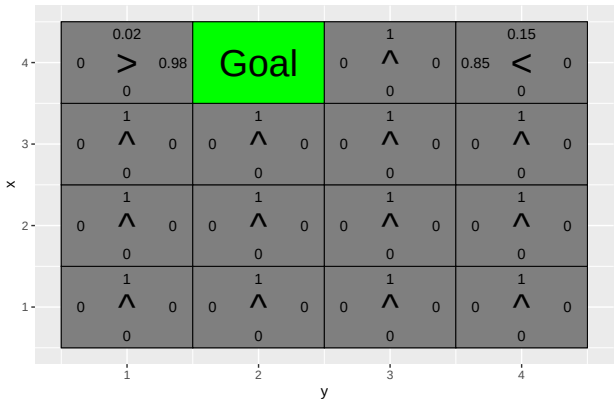

Action probabilities after 0 episodes
0.18 0.35 < 0.16 0.31
0.19 0.34 < 0.17 0.3
0.21 0.3 v 0.17 0.31
0.24 0.27 v 0.17 0.33
0.2 0.34 < 0.17 0.3
0.2 0.32 < 0.17 0.3
Goal
0.25 0.26 v 0.17 0.32
0.21 0.32 < 0.17 0.29
0.22 0.31 < 0.18 0.29
0.24 0.28 v 0.18 0.3
0.26 0.25 v 0.18 0.31
0.23 0.29 < 0.19 0.29
0.23 0.29 < 0.2 0.28
0.25 0.26 v 0.2 0.29
0.28 0.23 v 0.2 0.29
x
y
1
2
3
4

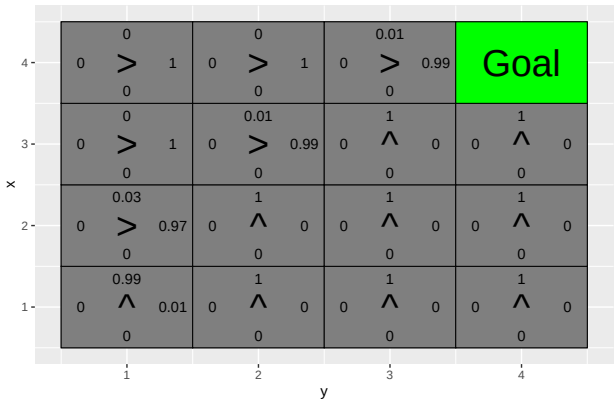

Action probabilities after 0 episodes
0.17 0.34 v 0.15 0.34
0.19 0.32 v 0.15 0.34
0.21 0.28 v 0.15 0.36
0.23 0.25 v 0.14 0.38
0.18 0.33 v 0.16 0.33
0.2 0.31 v 0.16 0.33
0.23 0.28 v 0.15 0.35
0.25 0.25 v 0.14 0.36
0.2 0.32 v 0.17 0.32
0.21 0.3 v 0.16 0.33
Goal
0.26 0.24 v 0.15 0.35
0.21 0.27 v 0.19 0.33
0.22 0.27 v 0.19 0.32
0.24 0.24 v 0.18 0.34
0.27 0.21 v 0.17 0.35
x
y
1
2
3
4

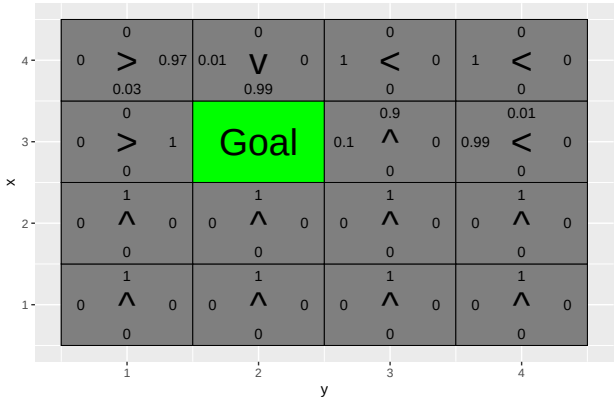

Action probabilities after 0 episodes
0.16 0.33 v 0.13 0.38
0.17 0.32 v 0.12 0.38
0.19 0.3 v 0.11 0.39
0.21 0.27 v 0.11 0.41
0.18 0.32 v 0.13 0.37
0.19 0.32 v 0.13 0.37
0.2 0.29 v 0.12 0.39
0.22 0.25 v 0.11 0.41
0.18 0.29 v 0.16 0.37
0.19 0.3 v 0.15 0.37
0.2 0.27 v 0.14 0.39
Goal
0.19 0.25 v 0.19 0.38
0.19 0.27 v 0.18 0.36
0.21 0.25 v 0.17 0.37
0.23 0.22 v 0.16 0.39
x
y
1
2
3
4


Action probabilities after 0 episodes
0.2 0.29 0.33 0.18 0.22 0.27 0.33 0.18 0.24 0.25 0.33 0.18 0.25 0.23 0.33 0.19
0.21 0.28 0.31 0.2 0.24 0.26 0.31 0.19 0.26 0.24 0.31 0.2 0.27 0.22 0.31 0.2
0.23 0.28 0.28 0.21 0.26 0.24 0.29 0.21 0.27 0.23 0.29 0.21 0.28 0.21 0.29 0.22
Goal
0.27 0.23 0.28 0.22 0.28 0.22 0.28 0.22 0.29 0.2 0.28 0.22
x
y

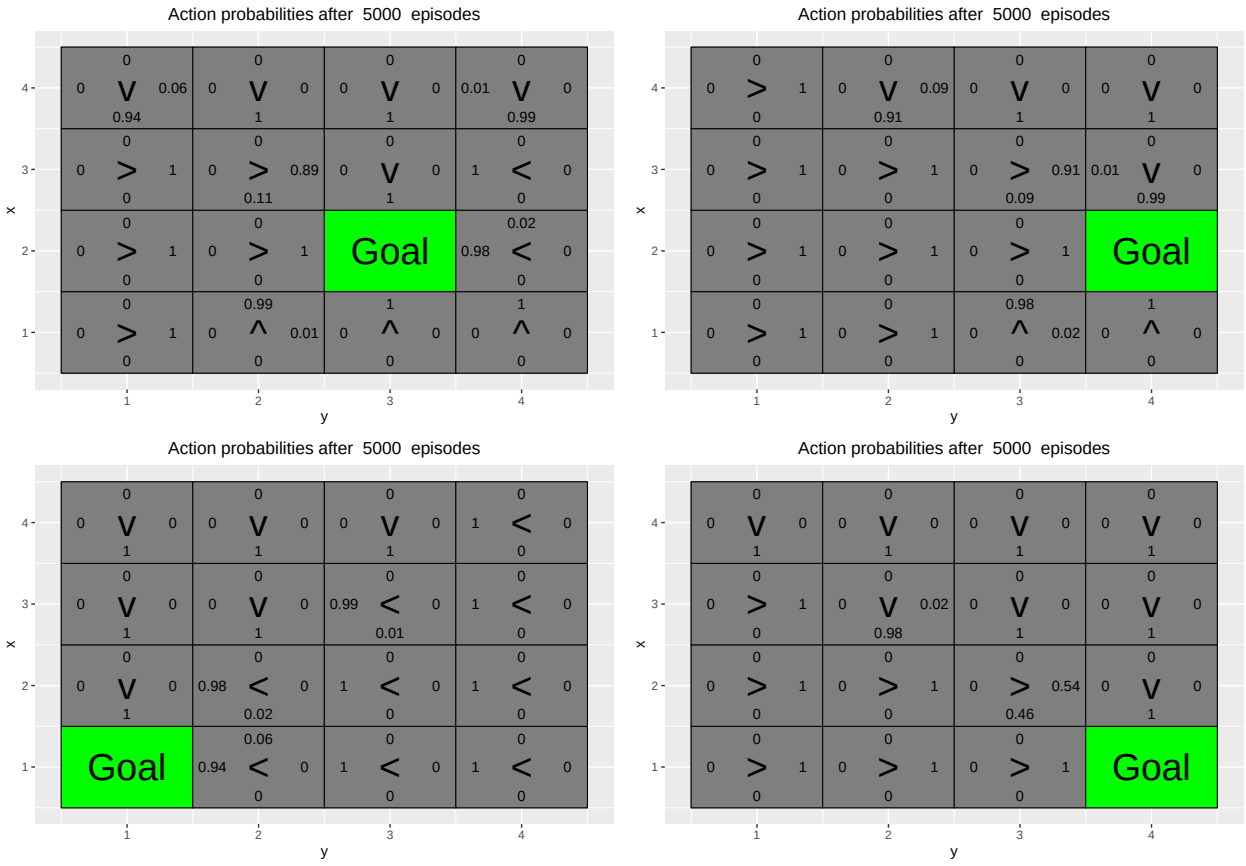

Action probabilities after 0 episodes
0.17 0.31 0.4 0.11 0.18 0.3 0.41 0.11 0.19 0.27 0.44 0.1 0.2 0.24 0.47 0.1
0.18 0.3 0.4 0.12 0.19 0.29 0.41 0.12 0.2 0.27 0.43 0.11 0.21 0.24 0.44 0.1
0.18 0.27 0.4 0.15 0.19 0.27 0.4 0.14 0.2 0.26 0.42 0.13 0.21 0.23 0.44 0.12
0.18 0.24 0.4 0.18 0.19 0.25 0.39 0.17 0.2 0.24 0.41 0.15
Goal
x
y


Action probabilities after 5000 episodes
0.02 0 0 0.98
Goal
1 0 0 0
0.15 0.85 0 0
1 0 0 0
1 0 0 0
1 0 0 0
1 0 0 0
1 0 0 0
1 0 0 0
1 0 0 0
1 0 0 0
x
y


Action probabilities after 5000 episodes
0 0 0 1
0 0 0 1
0.01 0 0 0.99
Goal
0 0 0 1
0.01 0 0 0.99
1 0 0 0
1 0 0 0
0.03 0 0 0.97
1 0 0 0
1 0 0 0
1 0 0 0
0.99 0 0 0.01
1 0 0 0
1 0 0 0
1 0 0 0
x
y


Action probabilities after 5000 episodes
0 0 0.03 0 0.97
0 0 0.99 0.01 0 0
0 0 0 1 0
0 0 0 1 0
0.9 0 0 0.1 0
0.01 0 0 0.99 0
1 0 0 0
1 0 0 0
1 0 0 0
1 0 0 0
1 0 0 0
1 0 0 0
1 0 0 0
1 0 0 0
x
y


Action probabilities after 5000 episodes
0 0 0 1
0 0 0 0.98
0 0.01 0 0.99
1 0 0 0
0 0 0 1
0 0 0 1
Goal
0.52 0.48 0 0
0 0 0 1
1 0 0 0
1 0 0 0
0.99 0 0 0.01
1 0 0 0
1 0 0 0
1 0 0 0
x
y




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

Based on the plots, the learnt policies are almost optimal where some actions are incorrect which cause a loop. The reason for this is that, REINFORCE network is just a classification network that maps the state to the taken action with a sample weight that comes from the reward and the discounted return. Since the goal changes, the set of states and actions with high reward and discounted return (sample weights) may not be useful or even misleading for the newly generated train environments. However, the model manage to converge and maximize the goal function $J(\theta) = V_\pi(S_0)$ by getting more samples by interacting with the environment.

Note that REINFORCE tries to learn the policy directly.

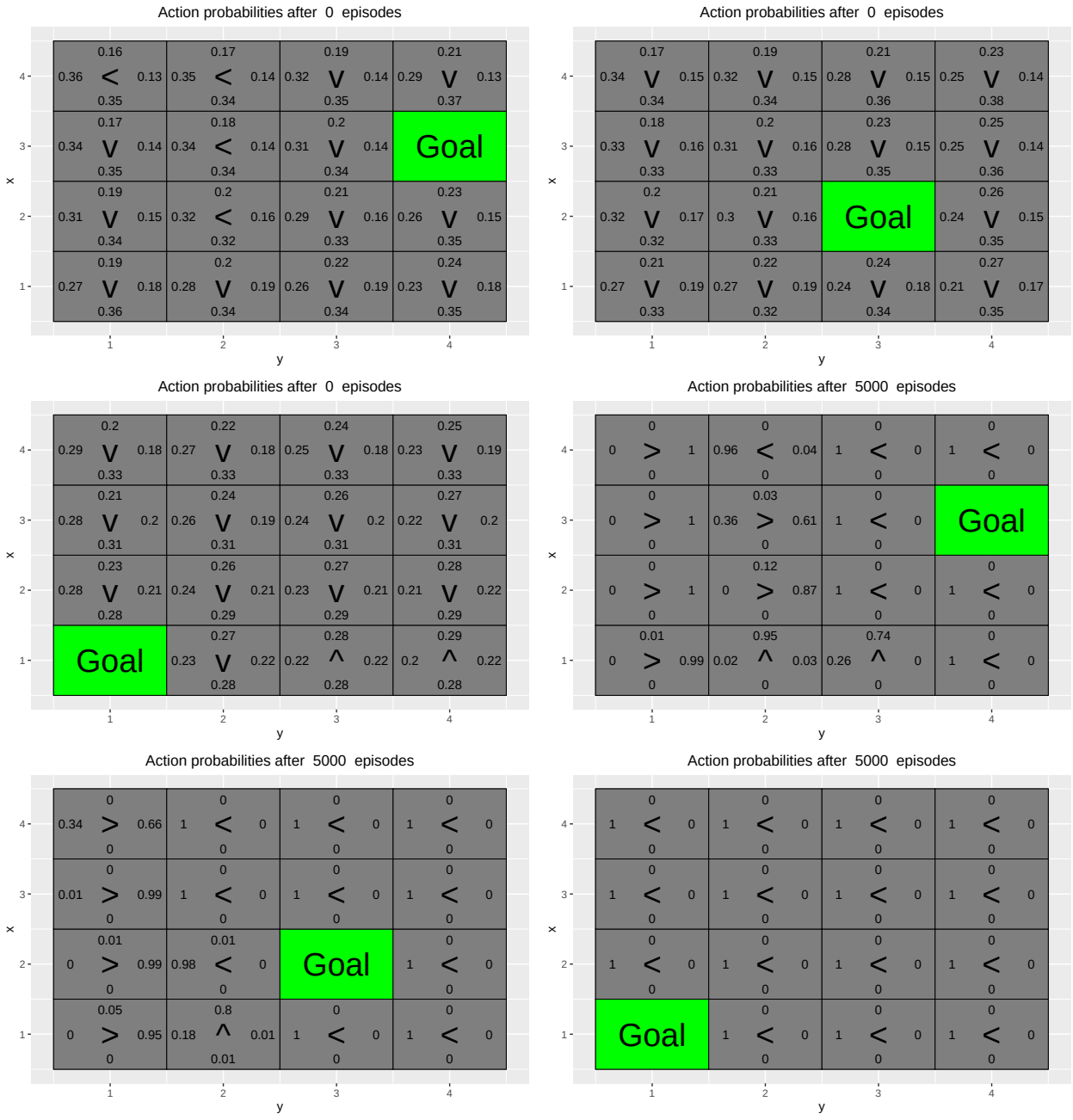
Could you have used the Q-learning algorithm to solve this task ?

No, since the Q Learning learns a parameterized Q table (state-action value table) based on the given environment. If the environment and the location of the goal changes, the learnt Q-values are not correct anymore, and agent cannot act based on the values and the policy derived from it.

## Question 2.7

```
# 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)
```




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

No. The agent learns a policy to get to the left side of the map, but in the validation maps which have different goals, agent would not preform good. The reason is that the trajectories with high sample weights

that the agent uses to learn cause the network to map any states to left action in most of the cases.

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

The reason is that, the way the environment changes now is different. In fact, the policy network classify most of the states to left action as the sample weights for those state-action pairs are big. But, when the goal moves to somewhere else on the map, the network would not generate a better action as it did not get enough state-action pairs with actions heading towards the new goal with big sample weights; in fact, the sample weight probably for any actions except left were low. However, in question D, the agent interacts with the training environment that is more close to the validation. In other words, in environment E the trajectories are biased in compare to the validation maps but in environment D the trajectories are unbiased.