Advanced Machine Learning

Hugo Morvan (hugmo418)

2024-10-03

Lab 3: Reinforcement Learning

The purpose of the lab is to put in practice some of the concepts covered in the lectures.

1. Q-Learning.

The file RL Lab1.R in the course website contains a template of the Qlearning algorithm. You are asked to complete the implementation. We will work with a gridworld environment consisting of HxW tiles laid out in a 2-dimensional grid. An agent acts by moving up, down, left or right in the grid-world. This corresponds to the following Markov decision process:

- State space: $S = (x, y)|x \in 1, ..., H, y \in 1, ..., W$.
- Action space: $A = up, down, left, right$.

Additionally, we assume state space to be fully observable. The reward function is a deterministic function of the state and does not depend on the actions taken by the agent. We assume the agent gets the reward as soon as it moves to a state. The transition model is defined by the agent moving in the direction chosen with probability $(1 - \beta)$. The agent might also slip and end up moving in the direction to the left or right of its chosen action, each with probability $\beta/2$. The transition model is unknown to the agent, forcing us to resort to model-free solutions. The environment is episodic and all states with a non-zero reward are terminal. Throughout this lab we use integer representations of the different actions: Up=1, right=2, down=3 and left=4.

2. Environment A.

For our first environment, we will use H = 5 and W = 7. This environment includes a reward of 10 in state (3,6) and a reward of -1 in states (2,3), (3,3) and (4,3). We specify the rewards using a reward map in the form of a matrix with one entry for each state. States with no reward will simply have a matrix entry of 0. The agent starts each episode in the state (3,1). The function `vis_environment` in the file `RL_Lab1.R` is used to visualize the environment and learned action values and policy. You will not have to modify this function, but read the comments in it to familiarize with how it can be used.

When implementing Q-learning, the estimated values of Q(S, A) are commonly stored in a data-structured called Q-table. This is nothing but a tensor with one entry for each state-action pair. Since we have a HxW environment with four actions, we can use a 3D-tensor of dimensions HxWx4 to represent our Q-table. Initialize all Q-values to 0. Run the function `vis_environment` before proceeding further. Note that each non-terminal tile has four values. These represent the action values associated to the tile (state). Note also that each non-terminal tile has an arrow. This indicates the greedy policy for the tile (ties are broken at random).

Your are requested to carry out the following tasks.

- Implement the greedy and ϵ-greedy policies in the functions `GreedyPolicy` and `EpsilonGreedyPolicy` of the file `RL_Lab1.R`. The functions should break ties at random, i.e. they should sample uniformly from the set of actions with maximal Qvalue.

```
set.seed(123)
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.
  greedy_actions = which.max(q_table[x,y,])
  # In case of multiple max, sample at random.
  greedy_action = sample(greedy_actions, 1)
  return(greedy_action)

}

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

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

  # Your code here.
  if(runif(1) < epsilon){
    #Act randomly
    return(sample.int(4,1))
  }else{
    #Act greedy
    return(GreedyPolicy(x,y))
  }
}
```

- Implement the Q-learning algorithm in the function `q_learning` of the file `RL_Lab1.R`. The function should run one episode of the agent acting in the environment and update the Q-table accordingly. The function should return the episode reward and the sum of the temporal-difference correction terms $R + \gamma max_a Q(S', a) - Q(S, A)$ for all steps in the episode. Note that a `transition_model` taking β as input is already implemented for you in the function transition model.

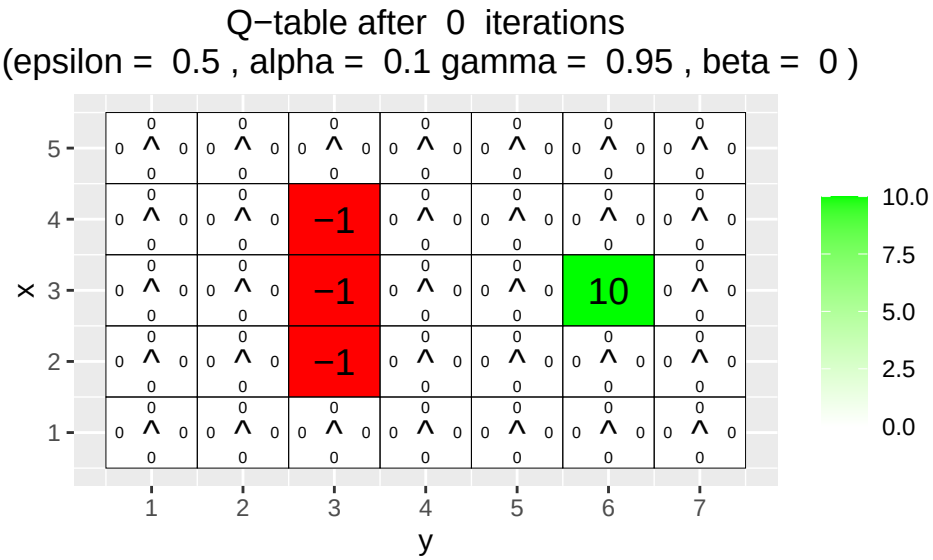
```
transition_model <- function(x, y, action, beta){
```

```
  # 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 randomly.
  #   alpha (optional): learning rate.
  #   gamma (optional): discount factor.
  #   beta (optional): slipping factor.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   reward: reward received in the episode.
  #   correction: sum of the temporal difference correction terms over the episode.
  #   q_table (global variable): Recall that R passes arguments by value. So, q_table being
  #   a global variable can be modified with the superassigment operator <<-.

  # Your code here.
  episode_correction = 0
  x = start_state[1]
  y = start_state[2]
  repeat{
    # Follow policy, execute action, get reward.
    action = EpsilonGreedyPolicy(x, y, epsilon) #follow policy
    new_state = transition_model(x, y, action, beta) #execute action
    new_x = new_state[1]
    new_y = new_state[2]
    reward = reward_map[new_x,new_y] #get reward
```

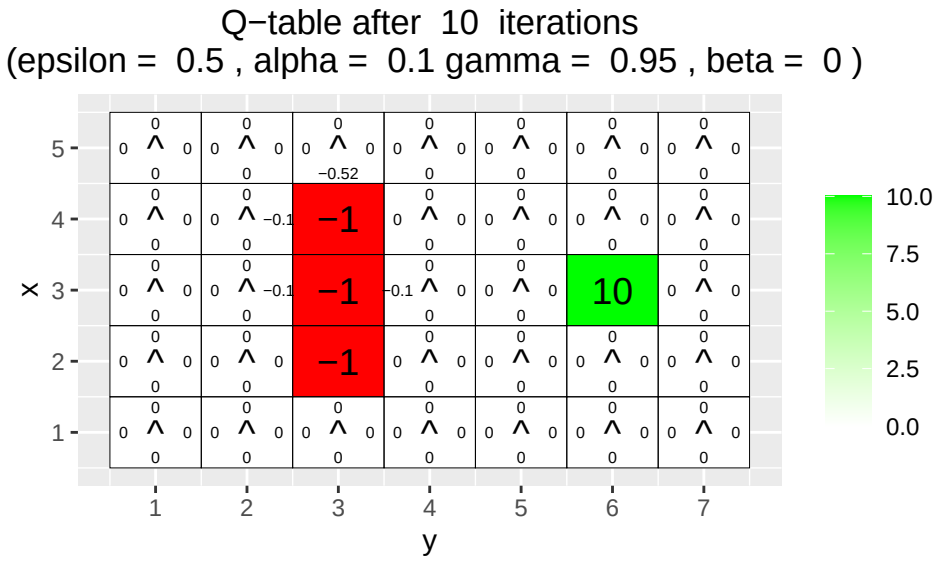
```
    # Q-table update.
    temporal_correction = reward + gamma*(max(q_table[new_x,new_y, ]))-q_table[x,y,action]

    q_table[x,y,action] <<- q_table[x,y,action] + alpha*temporal_correction
    episode_correction = episode_correction + temporal_correction
    #S = S'
    x = new_x
    y = new_y

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

}
```

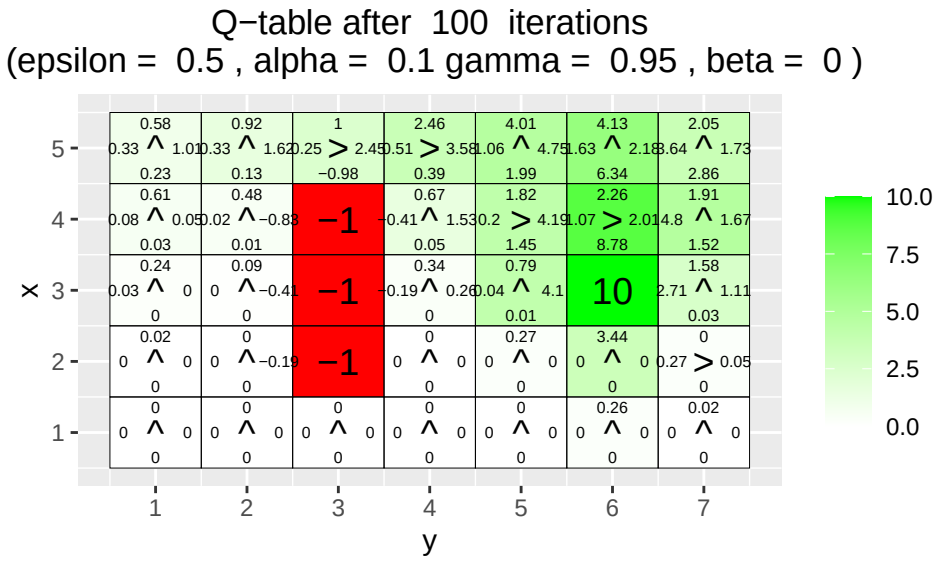
- Run 10000 episodes of Q-learning with $\epsilon = 0.5$, $\beta = 0$, $\alpha = 0.1$ and $\gamma = 0.95$. To do so, simply run the code provided in the file `RL_Lab1.R`. The code visualizes the Q-table and a greedy policy derived from it after episodes 10, 100, 1000 and 10000.

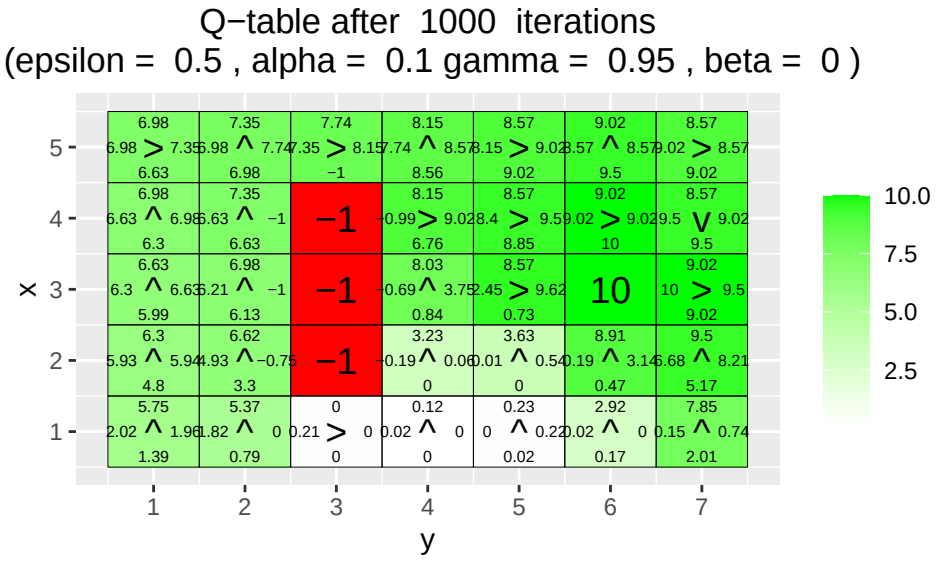
```
set.seed(123)
# Environment A (learning)

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()
```

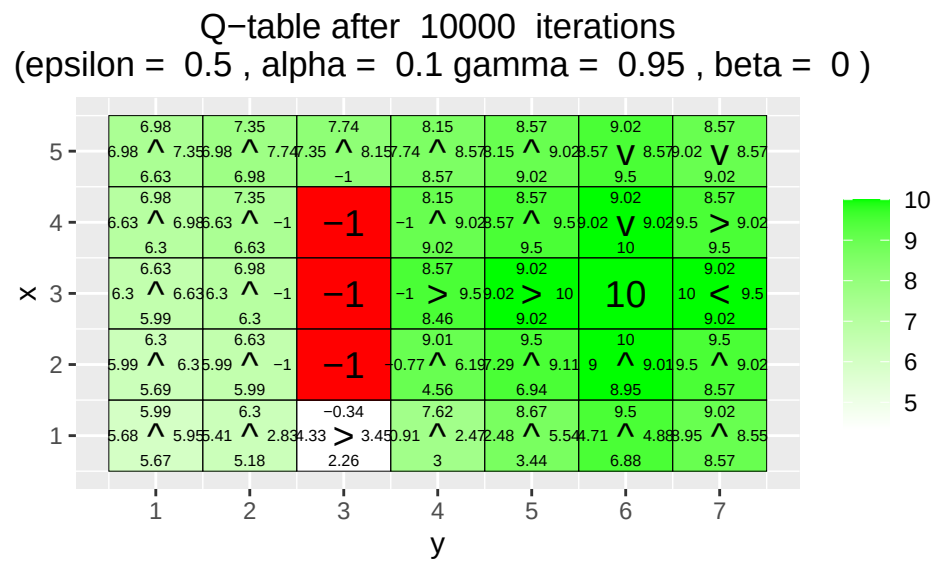


```
for(i in 1:10000){
  foo <- q_learning(start_state = c(3,1))

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









Answer the following questions:

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

After the first 10 episodes, the agent has not learned much: it has not found the 10pts goal and has only updated 3 Q-values, probably after encountering the -1 cells.

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

The final greedy policy after 10000 episodes is not optimal as it can be seen that the states below the negative rewards have not been optimized and some cells on the outside of the environment are pointing outwards. This is because the agent has not had time to explore fully the environment and come up with the optimal policy for all the states.

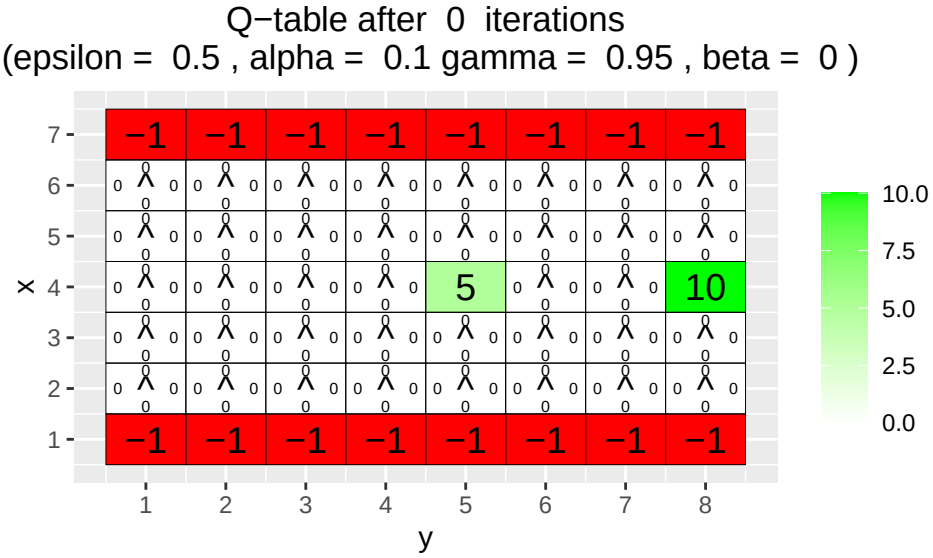
- Do the learned values in the Q-table reflect the fact that there are multiple paths (above and below the negative rewards) to get to the positive reward ? If not, what could be done to make it happen ?

The Q-table does not reflect the fact that there are multiple paths to get to the positive reward as it can be seen that the values are different between the top path and the bottom path. To make it happen, we could increase the exploring of the agent by increasing the epsilon parameter, which is responsible for the probability of acting randomly.

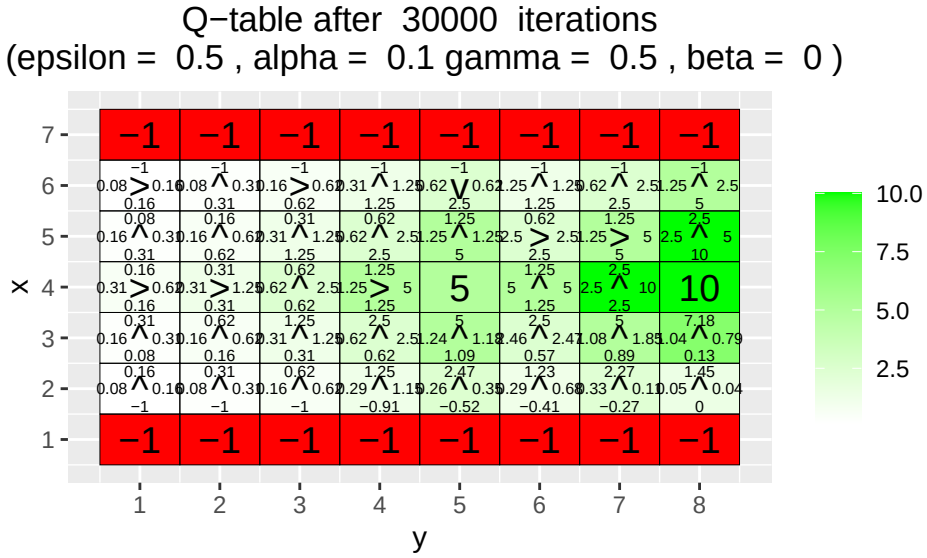
3. Environment B.

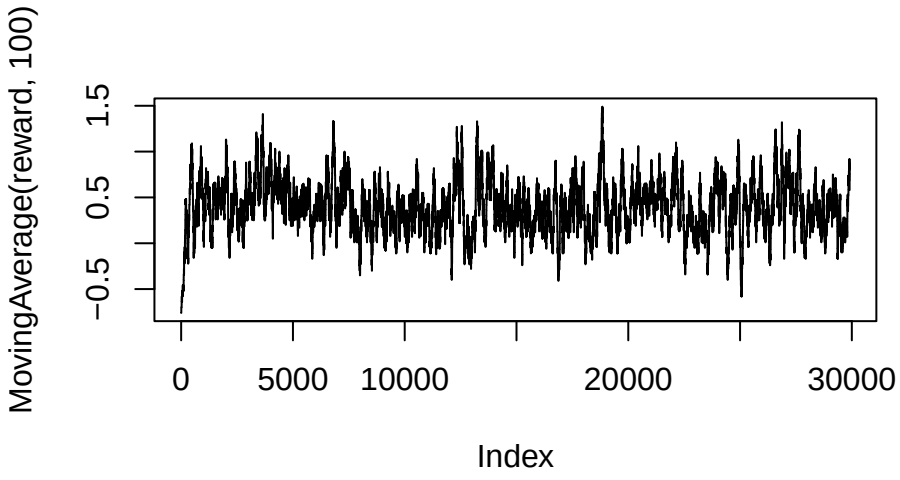
This is a 7x8 environment where the top and bottom rows have negative rewards. In this environment, the agent starts each episode in the state (4, 1). There are two positive rewards, of 5 and 10. The reward of 5 is easily reachable, but the agent has to navigate around the first reward in order to find the reward worth 10. Your task is to investigate how the ϵ and γ parameters affect the learned policy by running 30000 episodes of Q-learning with $\epsilon = 0.1, 0.5$, $\gamma = 0.5, 0.75, 0.95$, $\beta = 0$ and $\alpha = 0.1$. To do so, simply run the code provided in the file RL Lab1.R and explain your observations.

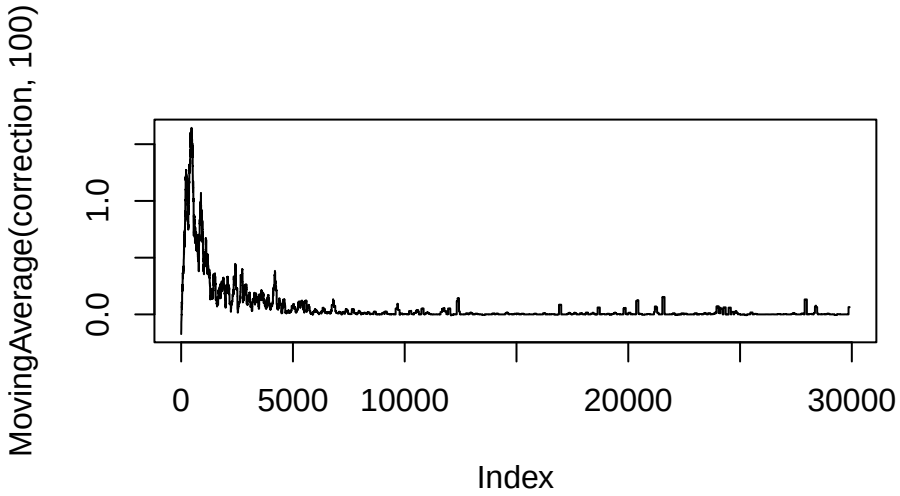
```
# Environment B (the effect of epsilon and gamma)

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()
```

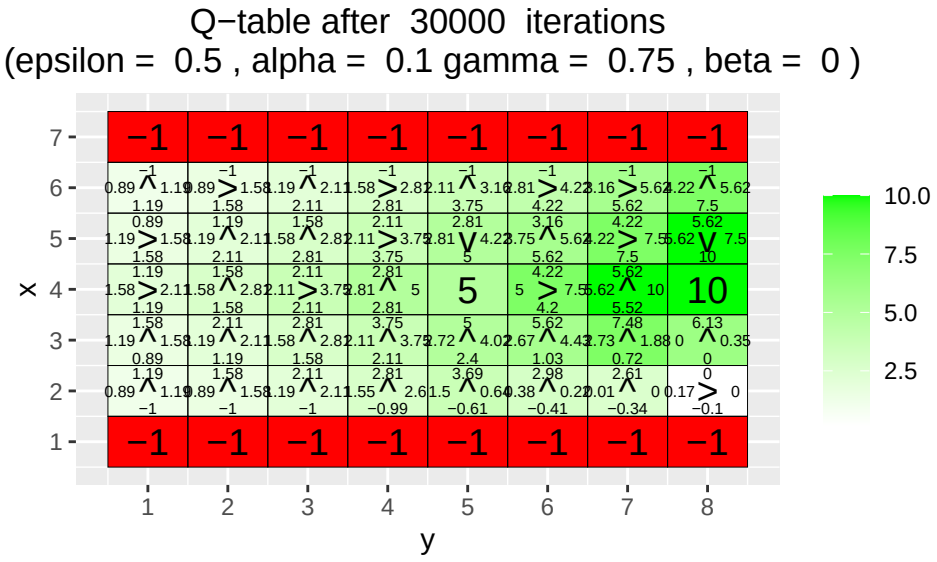


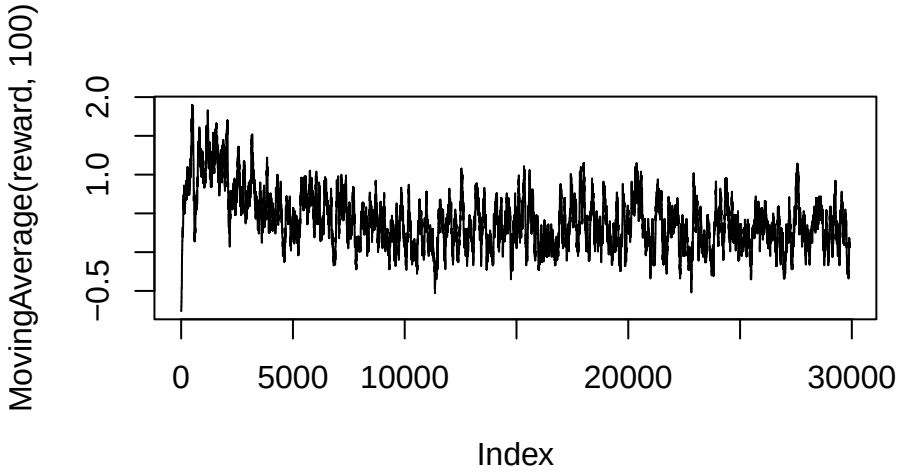
```
MovingAverage <- function(x, n){

  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")
}
```

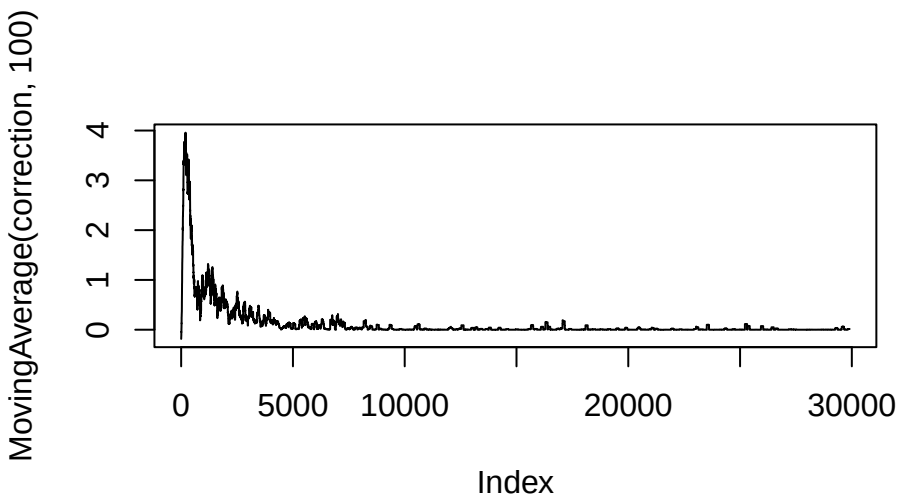


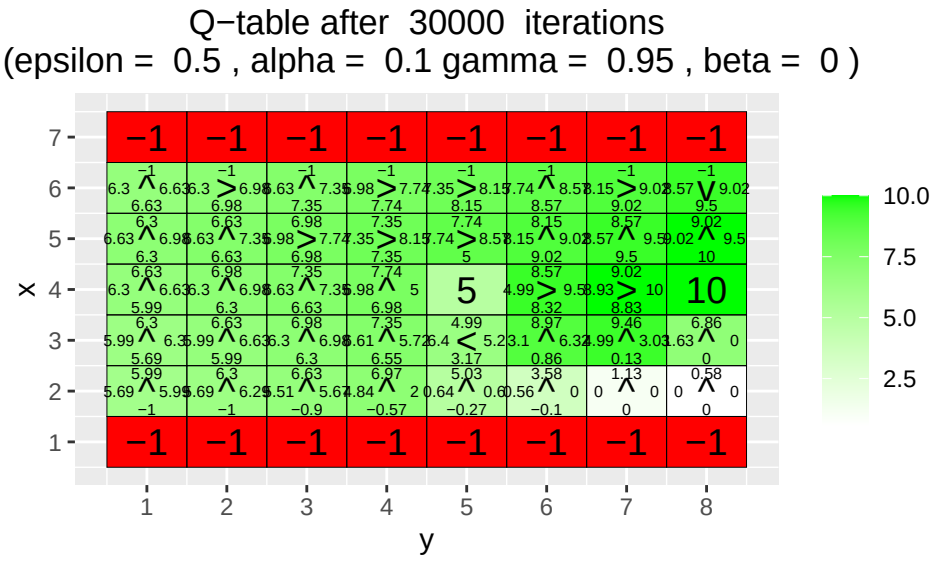

MovingAverage(reward, 100)
−0.5
0.5
1.5
0
5000
10000
20000
30000
Index

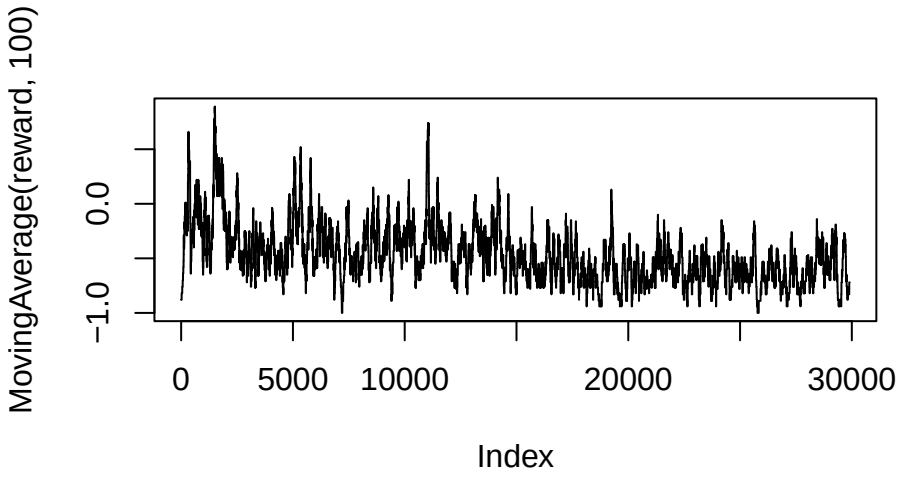

MovingAverage(correction, 100)
0.0
1.0
0
5000
10000
20000
30000
Index

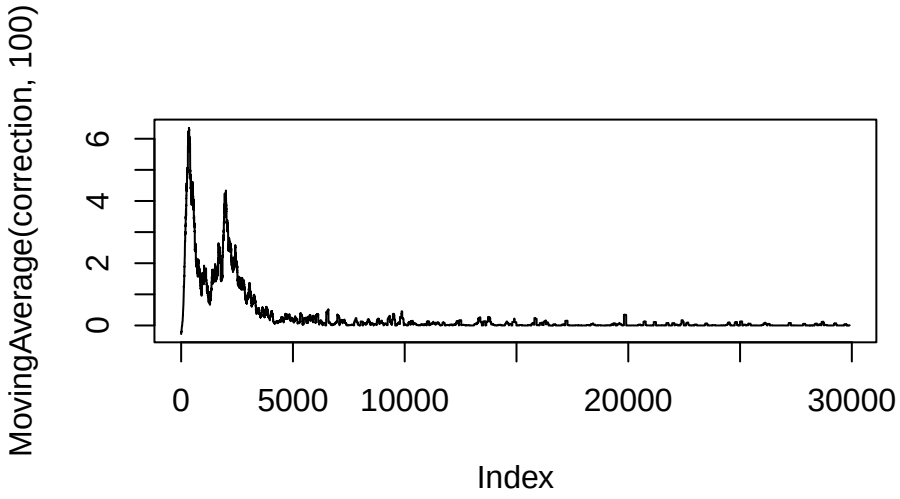

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5


MovingAverage(reward, 100)
Index

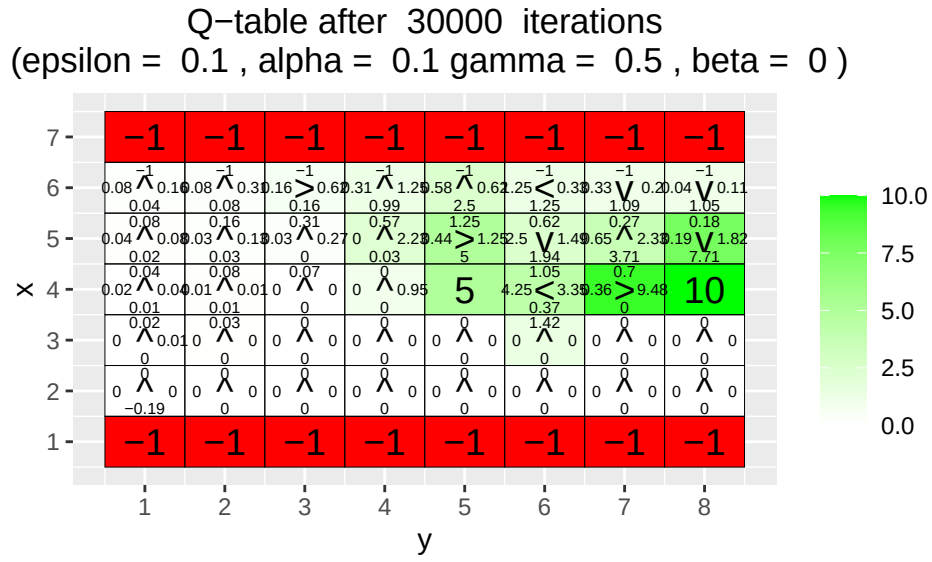

MovingAverage(correction, 100)
0 1 2 3 4
0 5000 10000 20000 30000
Index

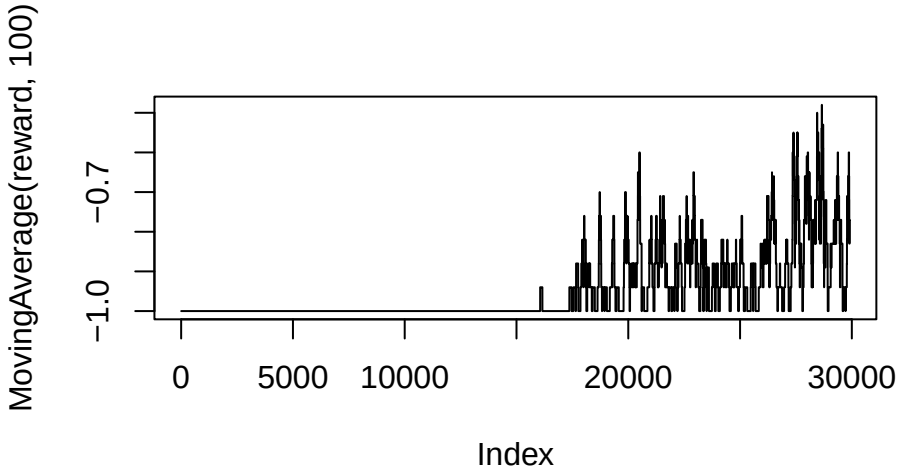

Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5

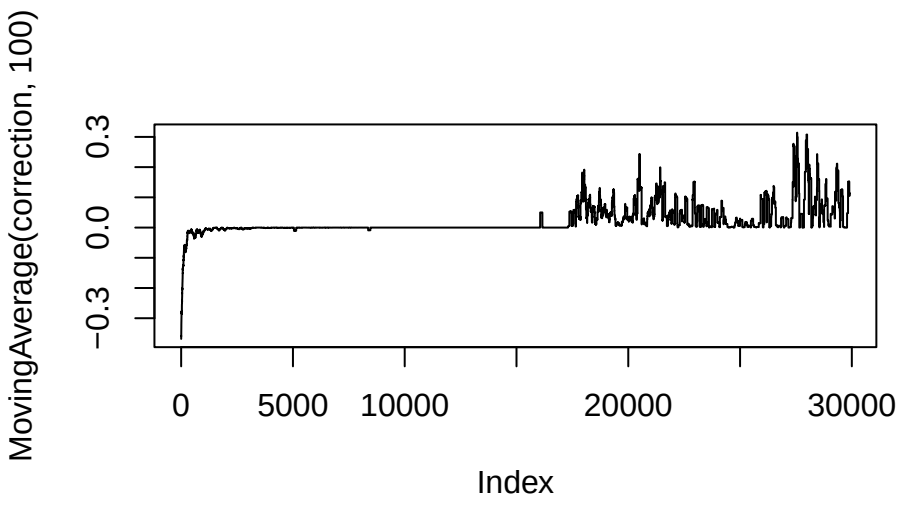


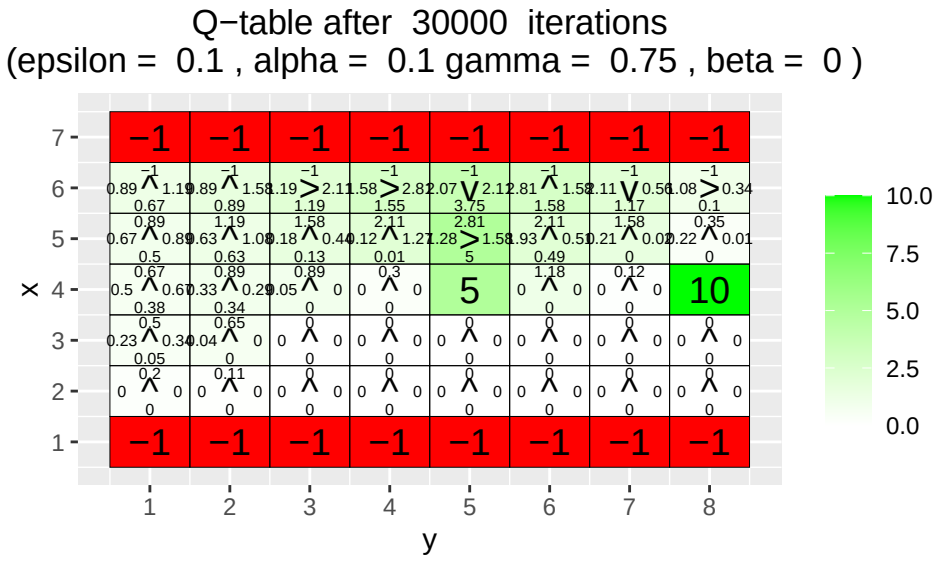


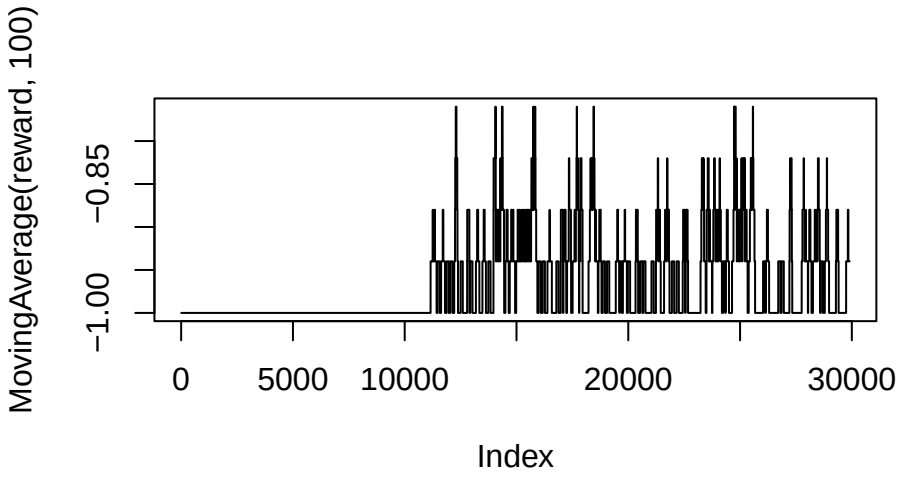

```
for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

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

  vis_environment(i, epsilon = 0.1, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}
```

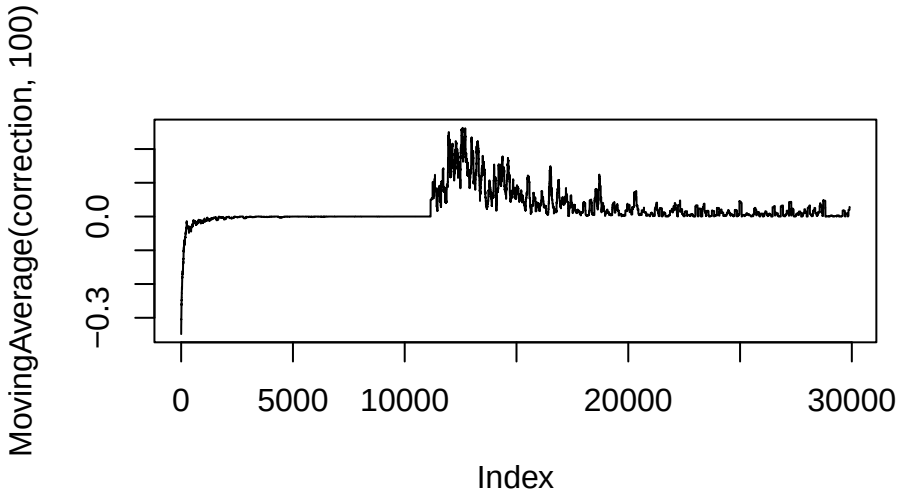

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.5 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0

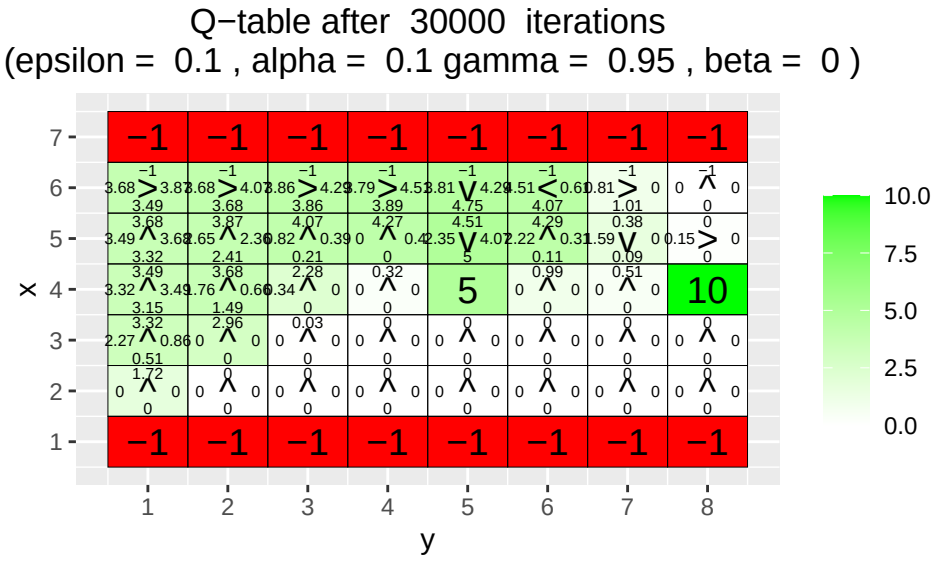

MovingAverage(reward, 100)
−1.0
−0.7
0
5000
10000
20000
30000
Index

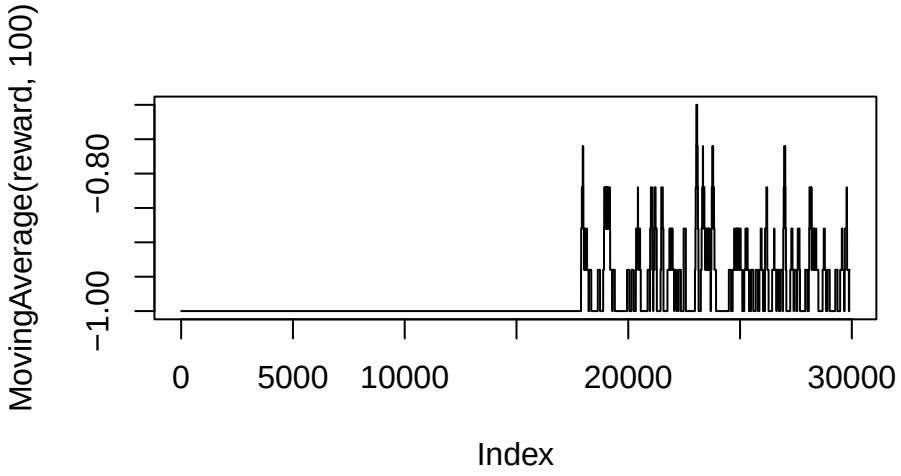

MovingAverage(correction, 100)
−0.3
0.0
0.3
0
5000
10000
20000
30000
Index

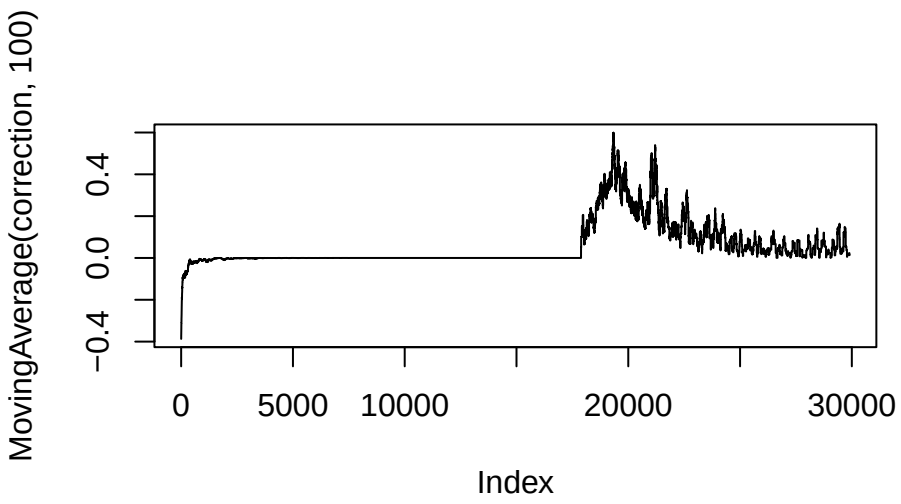

Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
−1.00
−0.85
0
5000
10000
20000
30000
Index


MovingAverage(correction, 100)
−0.3
0.0
0
5000
10000
20000
30000
Index


Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
−1.00
−0.80
0
5000
10000
20000
30000
Index



Effect of epsilon:

Epsilon is responsible for the exploration rate of the agent. If epsilon is small, the agent will prefer following the first good path it founds to reach the goal, and thus leaving a large part of the environment unexplored, as it can be seen on the plots with epsilon = 0.1. The reward is therefore smaller as the path chosen is not optimal. If epsilon is larger, like in the plots with epsilon = 0.5, the agent will explore the environment much more.

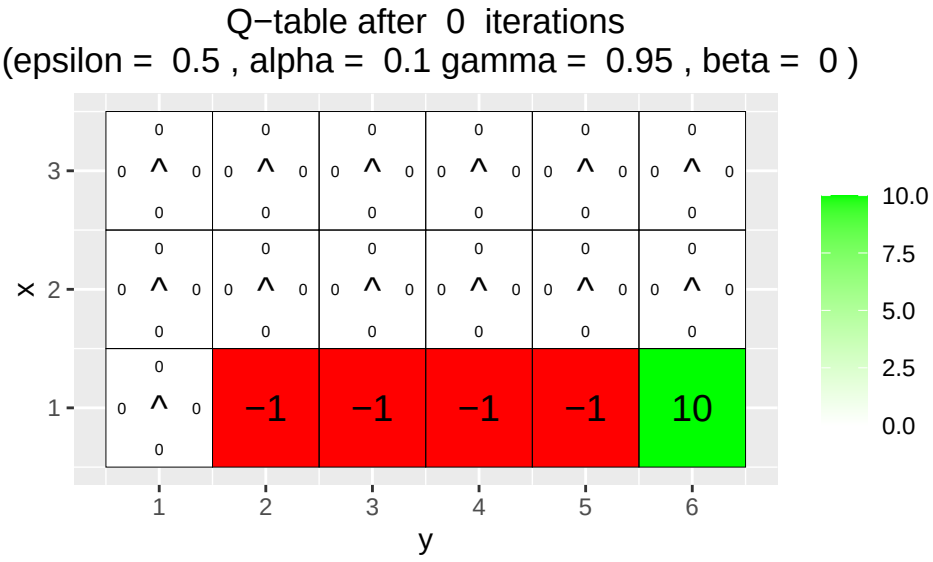
Effect of gamma:

Gamma describes the preference for the agent for present or future rewards. A large gamma gives a preference for future rewards while a small gamma gives a preference for immediate rewards. Short term reward can be observed on the graph with epsilon = 0.5 and gamma = 0.5 where the optimal policy goes towards the 5 reward while on the graph with epsilon = 0.5 and gamma = 0.95, the optimal policy goes around the 5 reward to go to the 10 reward instead.

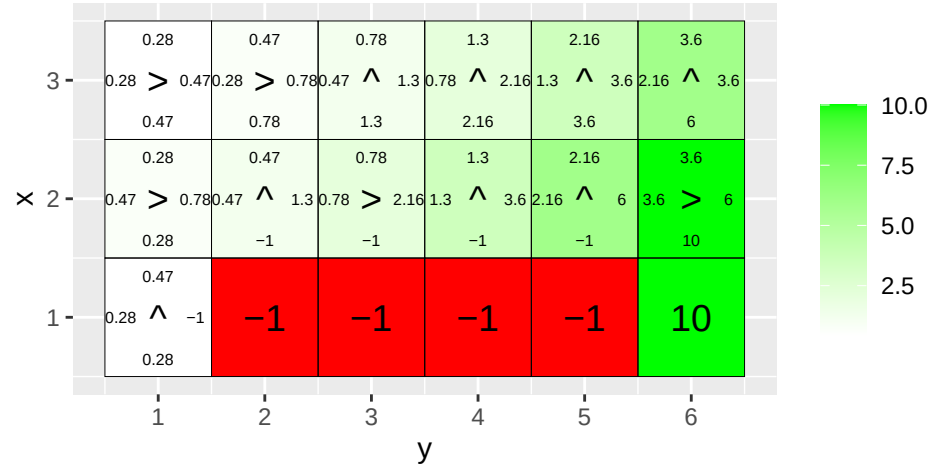
4. Environment C.

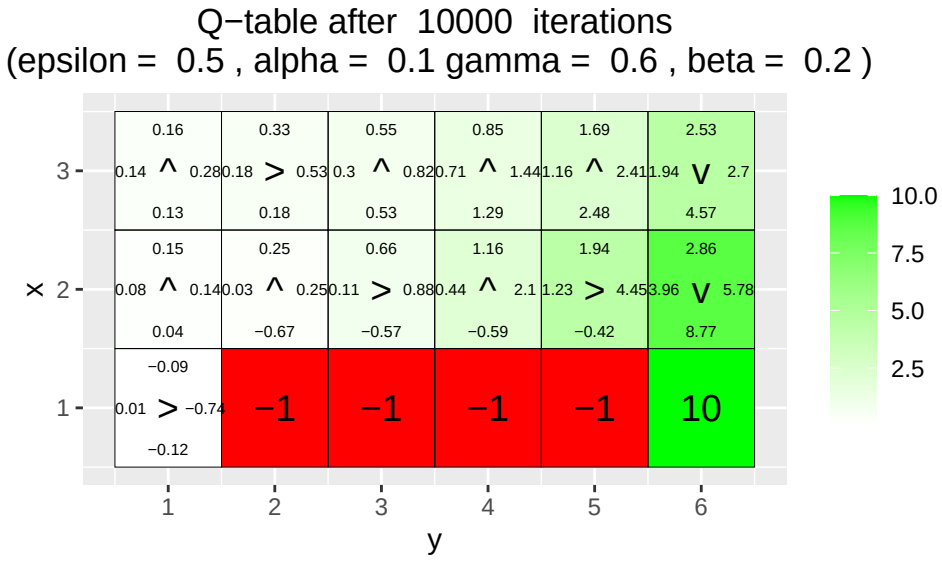
This is a smaller 3x6 environment. Here the agent starts each episode in the state (1,1). Your task is to investigate how the β parameter affects the learned policy by running 10000 episodes of Q-learning with β= 0, 0.2, 0.4, 0.66, $\epsilon = 0.5$, $\gamma = 0.6$ and $\alpha = 0.1$. To do so, simply run the code provided in the file RL Lab1.R and explain your observations.

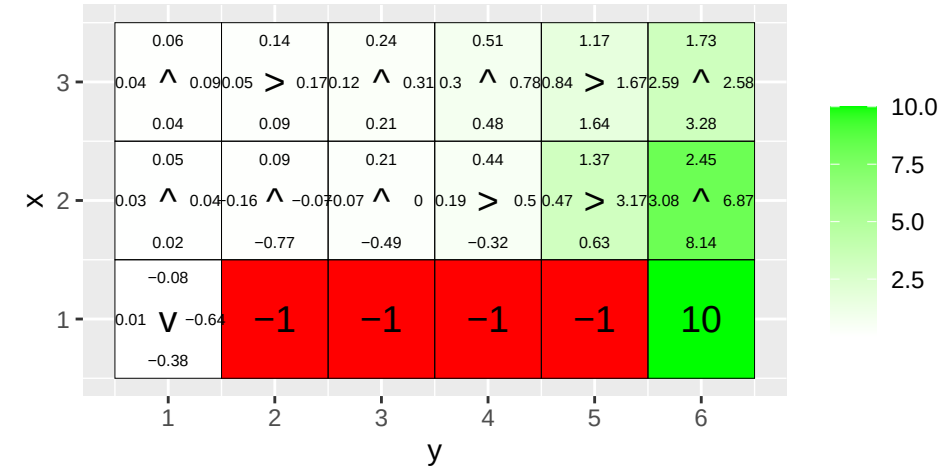
```
H <- 3
W <- 6

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

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

vis_environment()
```

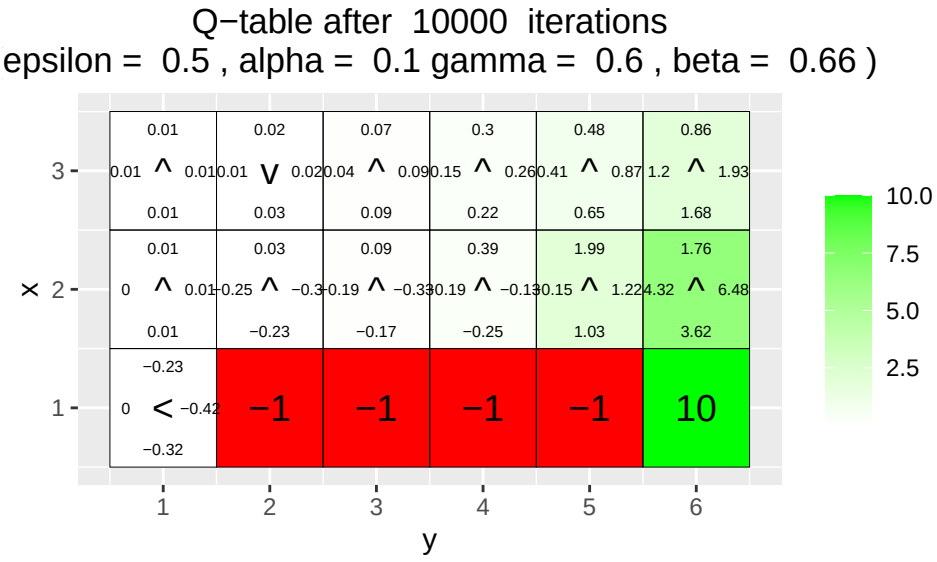



```
for(j in c(0,0.2,0.4,0.66)){
  q_table <- array(0,dim = c(H,W,4))

  for(i in 1:10000)
    foo <- q_learning(gamma = 0.6, beta = j, start_state = c(1,1))

  vis_environment(i, gamma = 0.6, beta = j)
}
```




Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.2)
0.16 0.14 ^ 0.28 0.13
0.33 0.18 > 0.53 0.18
0.55 0.3 ^ 0.82 0.53
0.85 0.71 ^ 1.44 1.29
1.69 1.16 ^ 2.41 2.48
2.53 1.94 v 2.7 4.57
0.15 0.08 ^ 0.14 0.04
0.25 0.03 ^ 0.25 −0.67
0.66 0.11 > 0.88 −0.57
1.16 0.44 ^ 2.1 −0.59
1.94 1.23 > 4.45 −0.42
2.86 3.96 v 5.78 8.77
−0.09 0.01 > −0.74 −0.12
−1
−1
−1
−1
10
x
y
10.0
7.5
5.0
2.5


Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.4)
0.06 0.04 ^ 0.09 0.04
0.14 0.05 > 0.17 0.09
0.24 0.12 ^ 0.31 0.21
0.51 0.3 ^ 0.78 0.48
1.17 0.84 > 1.67 1.64
1.73 2.59 ^ 2.58 3.28
0.05 0.03 ^ 0.04 0.02
0.09 −0.16 ^ −0.07 −0.77
0.21 0.07 ^ 0 −0.49
0.44 0.19 > 0.5 −0.32
1.37 0.47 > 3.17 0.63
2.45 3.08 ^ 6.87 8.14
−0.08 0.01 v −0.64 −0.38
−1
−1
−1
−1
10
x
y
10.0
7.5
5.0
2.5



The beta parameter is slipping factor, i.e. how often the agent will take an accidental side step. In this environment, the beta parameter will affect how much the agent is willing to be close to the -1 rewards. With beta=0, the agent does not slip and therefore does not care about being close to the -1 rewards and therefore goes alongside it to reach the 10 reward. With higher beta values, the agent tries to avoid getting close to the -1 rewards as there is a chance to accidentally walk into it, therefore the optimal policy is to take the long route by the top of the environment. The higher the beta, the more arrows are pointing perpendicular to the optimal direction, meaning that the agent knows that it will eventually take an accidental side step towards the optimal direction.

5. REINFORCE.

The file RL Lab2 Colab.ipynb in the course website contains an implementation of the REINFORCE algorithm. The file also contains the result of running the code, so that you do not have to run it. So, you do not need to run it if you do not want. Your task is to study the code and the results obtained, and answer some questions. We will work with a 4 x 4 grid. We want the agent to learn to navigate to a random goal position in the grid. The agent will start in a random position and it will be told the goal position. The agent receives a reward of 5 when it reaches the goal. Since the goal position can be any position, we need a way to tell the agent where the goal is. Since our agent does not have any memory mechanism, we provide the goal coordinates as part of the state at every time step, i.e. a state consists now of four coordinates: Two for the position of the agent, and two for the goal position. The actions of the agent can however only impact its own position, i.e. the actions do not modify the goal position. Note that the agent initially does not know that the last two coordinates of a state indicate the position with maximal reward, i.e. the goal position. It has to learn it. It also has to learn a policy to reach the goal position from the initial position. Moreover, the policy has to depend on the goal position, because it is chosen at random in each episode. Since we only have a single non-zero reward, we do not specify a reward map. Instead, the goal coordinates are passed to the functions that need to access the reward function.

6. Environment D.

In this task, we will use eight goal positions for training and, then, validate the learned policy on the remaining eight possible goal positions. The training and validation goal positions are stored in the lists `train_goals` and `val_goals` in the code in the file `RL_Lab2_Colab.ipynb`. The results provided in the file correspond to running the REINFORCE algorithm for 5000 episodes with $\beta = 0$ and $\gamma = 0.95$. Each

training episode uses a random goal position from `train_goals`. The initial position for the episode is also chosen at random. When training is completed, the code validates the learned policy for the goal positions in `val_goals`. This is done by with the help of the function vis prob, which shows the grid, goal position and learned policy. Note that each non-terminal tile has four values. These represent the action probabilities associated to the tile (state). Note also that each nonterminal tile has an arrow. This indicates the action with the largest probability for the tile (ties are broken at random).

Finally, answer the following questions:

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

It seems like the agent has learned a good policy as no matter where the goal is appearing, all the arrow are pointing towards it. This is because it has been trained on a good variation of goal positions, spreading the entire environment, thus it is able to learn that the goal position is where the agent should be going to.

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

Q-learning would have not be able to solve this task as it can only learn one optimal policy, which would not react well to a goal changing position.

7. Environment E.

In this task, the goals for training are all from the top row of the grid. The validation goals are three positions from the rows below. To solve this task, simply study the code and results provided in the file RL Lab2 Colab.ipynb and answer the following questions:

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

The agent has not learned a good policy. This is because the agent only sees goals in the first row of the environment during training, it is therefore not able to adapt to goals appearing in the lower two rows. It could be because the agent only learned to go up and from side to side to reach the goal, and is therefore not able to adapt to the goal behind at the bottom.

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

The results obtained in environments D and E differ because the agent is not trained in the same conditions. Hence, the generalization capabilities and behavior learned by the agent are different. It could be that in environment E, the y coordinate has no weight in the decision of the agent, while in environment D, it has some weight.