oct2020

2024-10-26

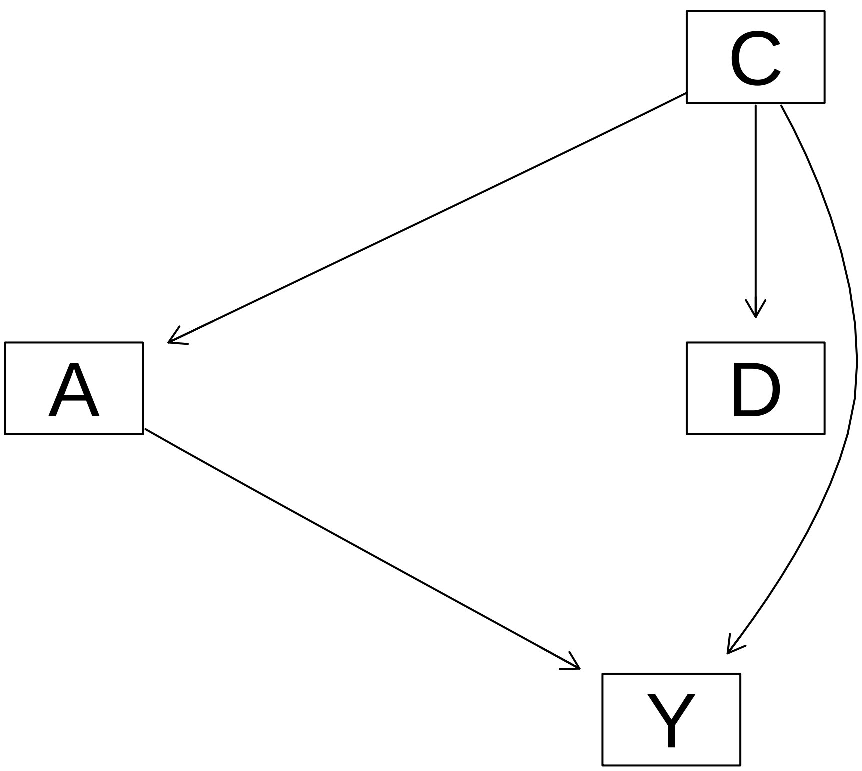
1. Graphical Models (6 p)

```
set.seed(123)
library("bnlearn")
library("gRain")
```

```
## Loading required package: gRbase

##
## Attaching package: 'gRbase'

## The following objects are masked from 'package:bnlearn':
##
##     ancestors, children, nodes, parents
```

```
dag = model2network("[C][D|C][A|C][Y|A:C]") # Construct dag
graphviz.plot(dag)
```

```
## Loading required namespace: Rgraphviz
```




```
results = matrix(NA, 1000, 4)
resC = 0
```

```r
resD = 0

for (i in 1:1000) {
  temp_dag = dag

  ### Sample C
  cptC = runif(2)
  cptC = cptC/sum(cptC) # normalize
  dim(cptC) = c(2)
  dimnames(cptC) = list(c("1", "0"))

  ### Sample A
  cptA = runif(4)
  dim(cptA) = c(2,2)
  cptA = prop.table(cptA, 2) # normalize
  dimnames(cptA) = list("A" = c("1", "0"), "C" = c("1", "0"))

  ### Sample D
  cptD = runif(4)
  dim(cptD) = c(2,2)
  cptD = prop.table(cptD, 2)# normalize
  dimnames(cptD) = list("D" = c("1", "0"), "C" = c("1", "0"))

  ### Sample Y
  cptY = runif(8)
  dim(cptY) = c(2,2,2)
  cptY = prop.table(cptY, 2:3) # normalize
  dimnames(cptY) = list("Y" = c("1", "0"), "C" = c("1", "0"), "A" = c("1", "0"))

  customfit = custom.fit(temp_dag, list(A = cptA, D = cptD, C = cptC, Y = cptY))
  grain_fit = as.grain(customfit)
  grain = compile(grain_fit)
  grain

  # convert to grain objects and set evidence
  pac11 = setEvidence(grain, nodes = c("A", "C"), states = c("1","1"))
  pos_pac11 = querygrain(pac11, nodes = "Y")$Y[1]
  pac10 = setEvidence(grain, nodes = c("A", "C"), states = c("1","0"))
  pos_pac10 = querygrain(pac10, nodes = "Y")$Y[1]
  pac01 = setEvidence(grain, nodes = c("A", "C"), states = c("0","1"))
  pos_pac01 = querygrain(pac01, nodes = "Y")$Y[1]
  pac00 = setEvidence(grain, nodes = c("A", "C"), states = c("0","0"))
  pos_pac00 = querygrain(pac00, nodes = "Y")$Y[1]

  pad11 = setEvidence(grain, nodes = c("A", "D"), states = c("1","1"))
  pos_pad11 = querygrain(pad11, nodes = "Y")$Y[1]
  pad10 = setEvidence(grain, nodes = c("A", "D"), states = c("1","0"))
  pos_pad10 = querygrain(pad10, nodes = "Y")$Y[1]
  pad01 = setEvidence(grain, nodes = c("A", "D"), states = c("0","1"))
  pos_pad01 = querygrain(pad01, nodes = "Y")$Y[1]
  pad00 = setEvidence(grain, nodes = c("A", "D"), states = c("0","0"))
  pos_pad00 = querygrain(pad00, nodes = "Y")$Y[1]
```

```r
  # p(y|a, c) is non-decreasing
  nondecC = ( (pos_pac11 >= pos_pac10) & (pos_pac01 >= pos_pac00))
  # p(y|a, c) is non-increasing
  nonincC = (pos_pac11 <= pos_pac10 & pos_pac01 <= pos_pac00)

  # p(y|a, d) is non-decreasing
  nondecD = (pos_pad11 >= pos_pad10 & pos_pad01 >= pos_pad00)
  # p(y|a, d) is non-increasing
  nonincD = (pos_pad11 <= pos_pad10 & pos_pad01 <= pos_pad00)

  if((nondecC == 1 | nonincC == 1) & (nondecD == 0 & nonincD == 0)) {
    resC = resC + 1
  }

  if((nondecD == 1 | nonincD == 1) & (nondecC == 0 & nonincC == 0)) {
    resD = resD + 1
  }

  results[i,] = c(nondecC, nonincC, nondecD, nonincD)
}

resC
```

```
## [1] 0
```

```r
resD
```

```
## [1] 0
```

```r
# monotone in C but not D
colSums(results[which(results[,1]==TRUE & results[,2]==FALSE & results[,3]==FALSE & results[,4]==FALSE)
```

```
## [1] 0 0 0 0
```

```r
colSums(results[which(results[,1]==FALSE & results[,2]==TRUE & results[,3]==FALSE & results[,4]==FALSE)
```

```
## [1] 0 0 0 0
```

```r
# monotone in D but not C
colSums(results[which(results[,1]==FALSE & results[,2]==FALSE & results[,3]==TRUE & results[,4]==FALSE)
```

```
## [1] 0 0 0 0
```

```r
colSums(results[which(results[,1]==FALSE & results[,2]==FALSE & results[,3]==FALSE & results[,4]==TRUE)
```

```
## [1] 0 0 0 0
```

2. Reinforcement Learning (7 p)

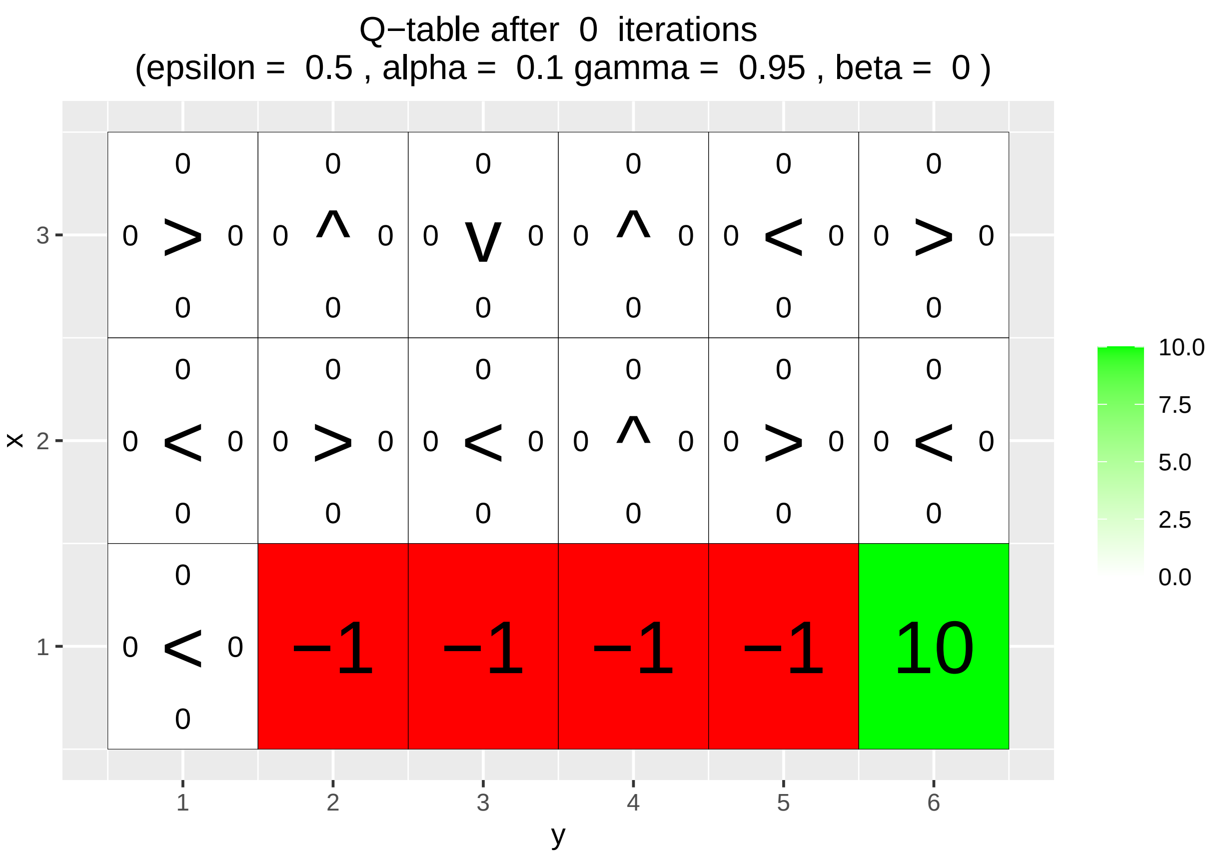
```r
set.seed(1234)
library(ggplot2)

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

```r
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.
  #
```

```r
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  q_values = q_table[x, y, ]

  # Find all actions with the maximum Q-value
  max_actions = which(q_values == max(q_values))
  if (length(max_actions) == 1) {
    return(max_actions)
  } else {
    return(sample(max_actions, 1))
  }

}

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

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

  # Your code here.
  if (runif(1) < epsilon) {
    return (sample(1:4,1))
  } else {
    return (GreedyPolicy(x,y))
  }
}

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)))
```

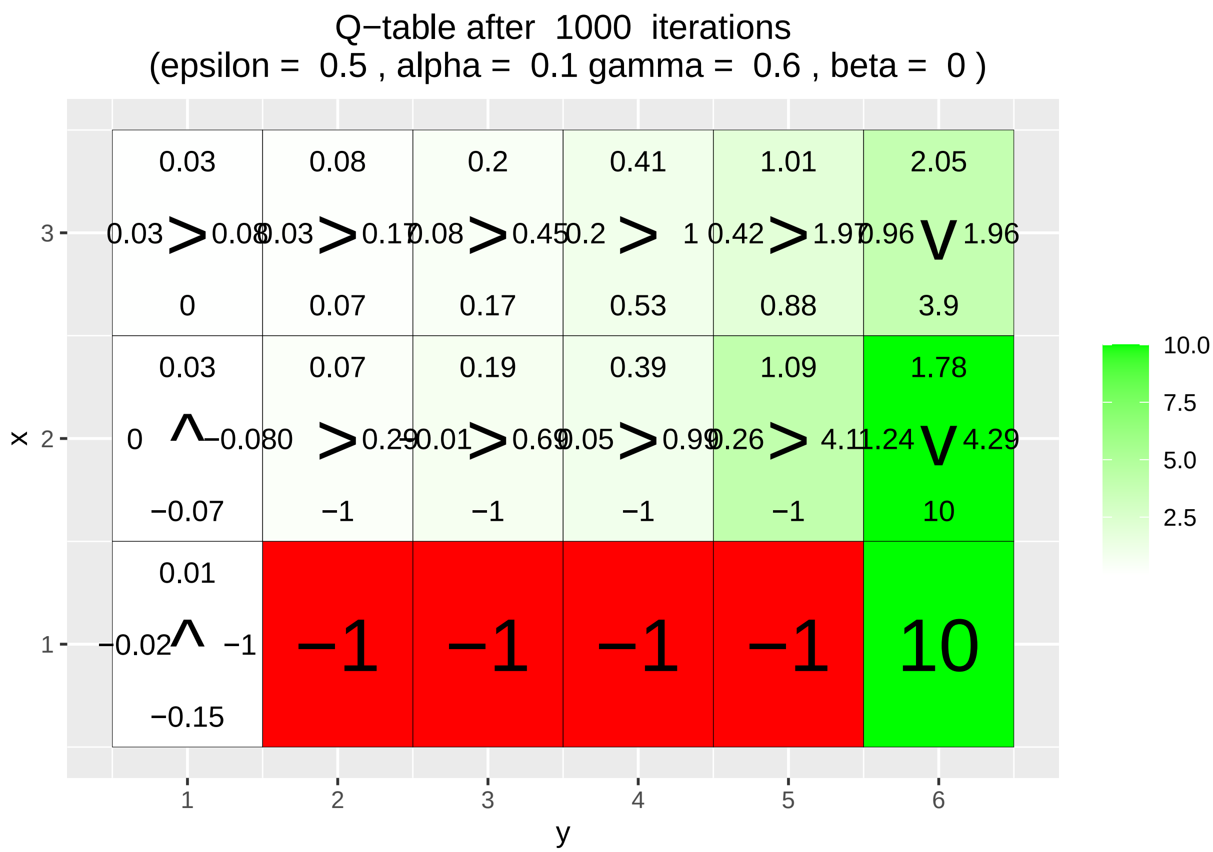
```r
  return (foo)
}

q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                       beta = 0, tr = 1){

  # 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 <<-.

  # initialize Q
  Q = start_state
  x = Q[1]
  y = Q[2]
  episode_correction = 0
  new_action = EpsilonGreedyPolicy(x,y,epsilon*tr) # follow policy
  repeat{

    # Follow policy, execute action, get reward.
    action = new_action # follow policy
    next_state = transition_model(x,y,action,beta) # excecute action
    reward = reward_map[next_state[1],next_state[2]] # get reward

    # Find next action
    new_action = EpsilonGreedyPolicy(next_state[1],next_state[2],epsilon*tr)

    # Q-table update.
    correction = reward + gamma * (q_table[next_state[1],next_state[2],new_action])-q_table[x,y,action]
    q_table[x,y,action] <<- q_table[x,y,action] + alpha * (correction*tr)
    episode_correction = episode_correction + correction

    x = next_state[1]
    y = next_state[2]

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

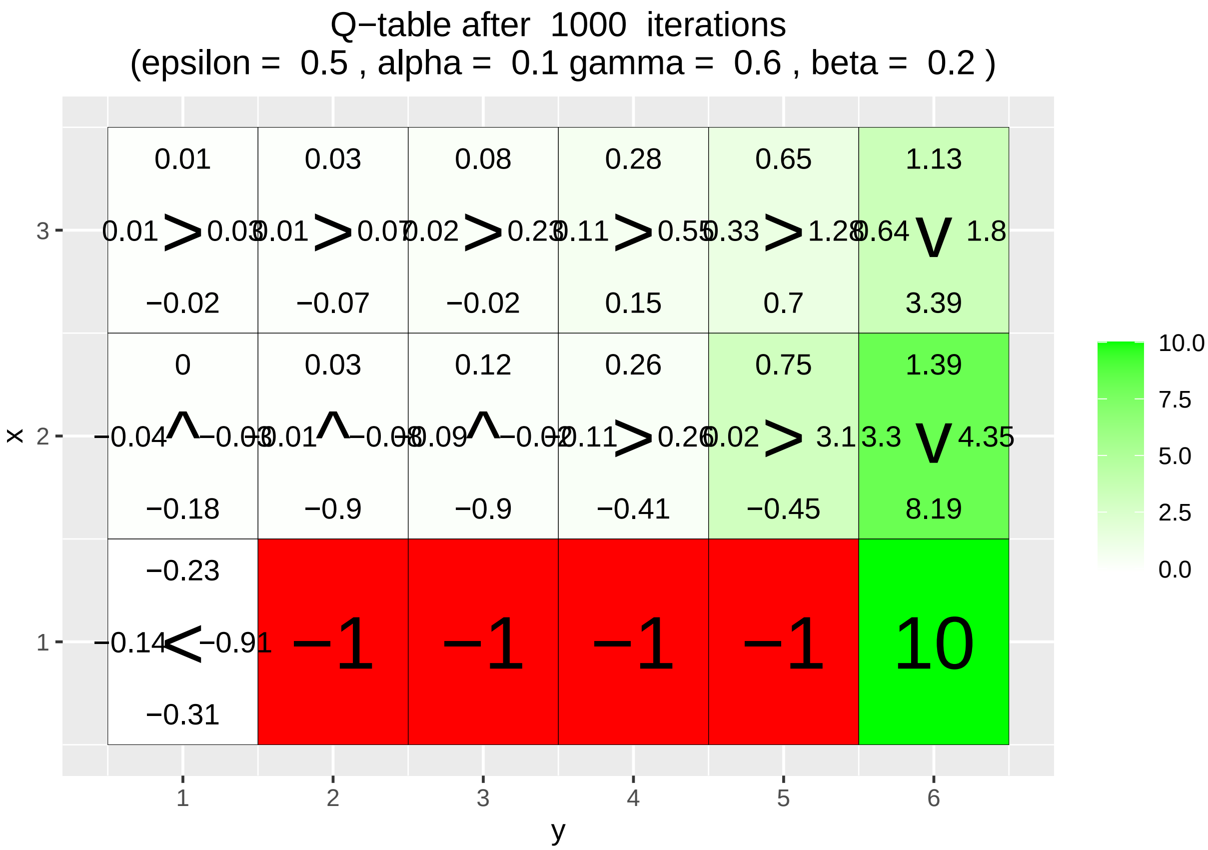
```r
}


MovingAverage <- function(x, n){

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

  return (rsum)
}
# ENV C
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()
```

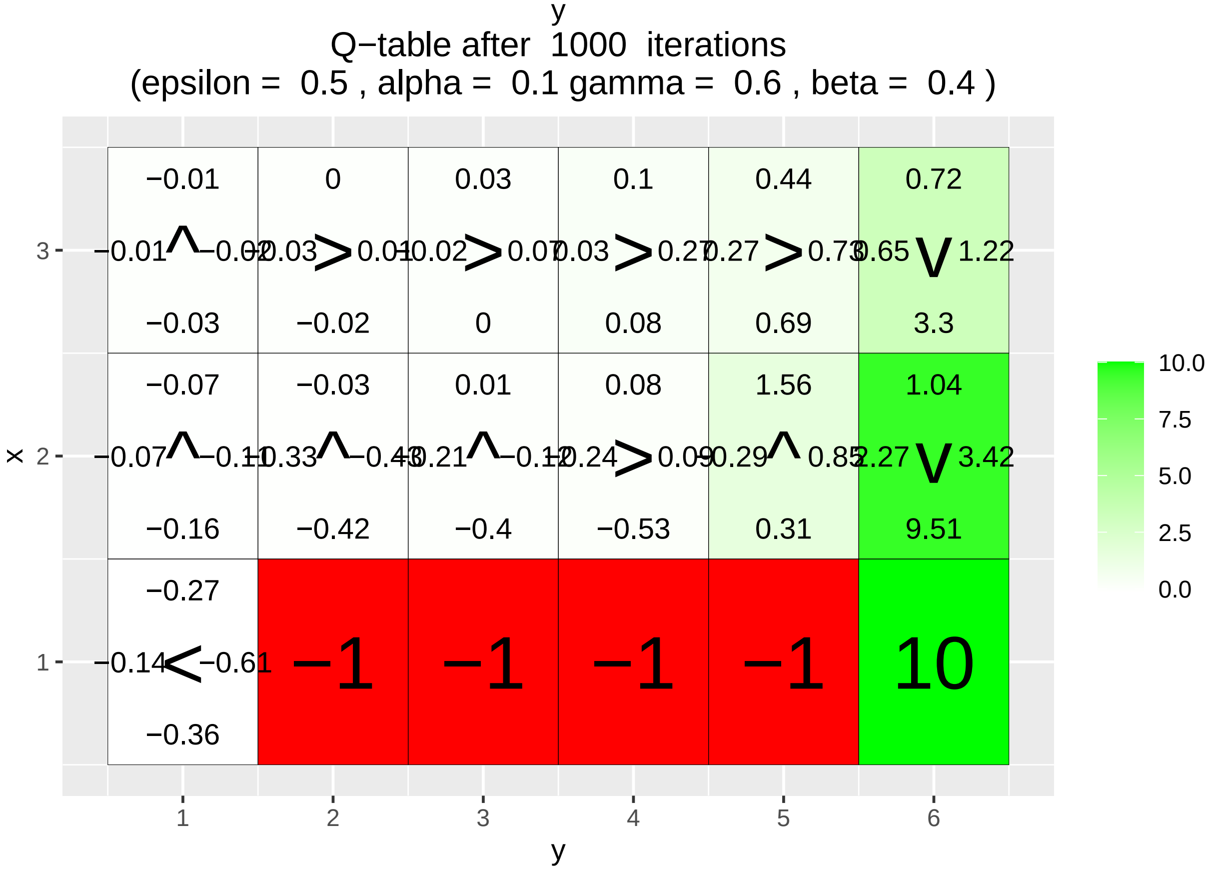


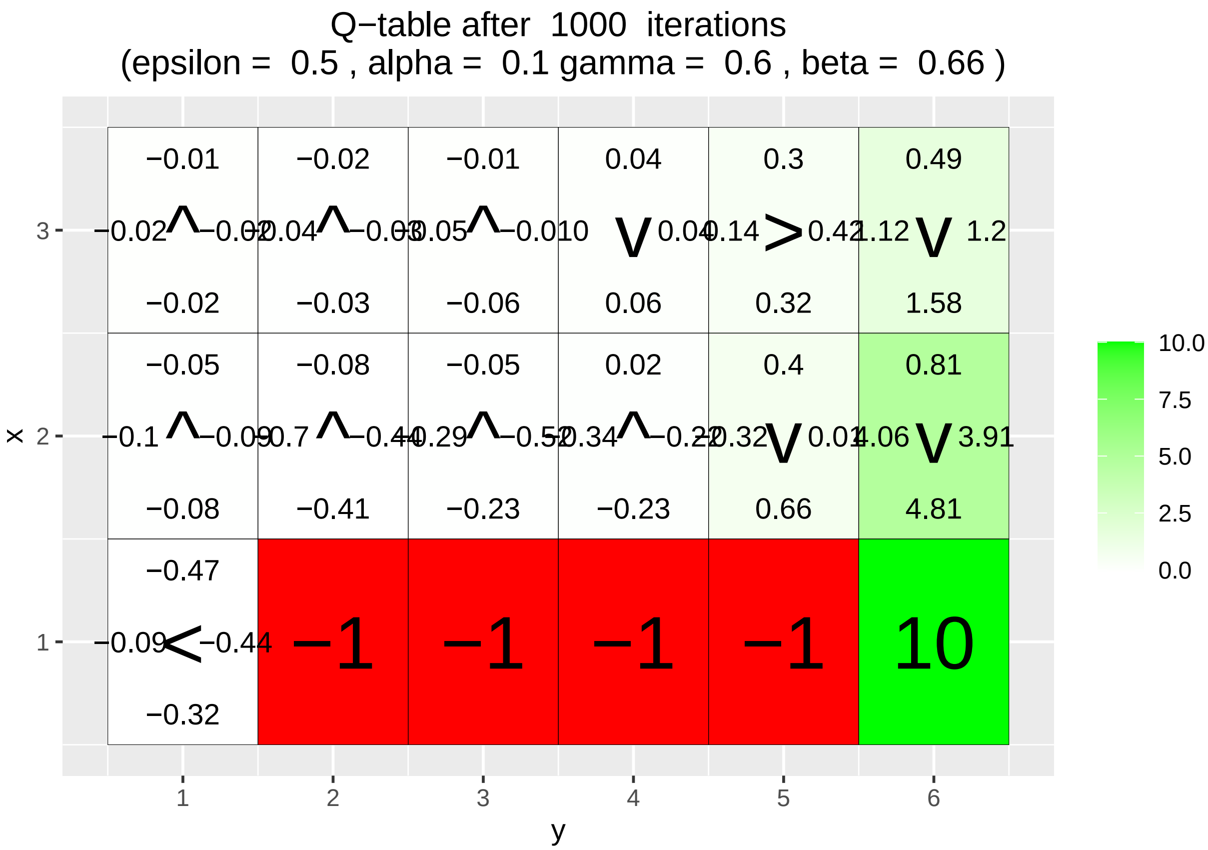

```r
reward  =NULL
for(j in c(0,0.2,0.4,0.66)){
  q_table <- array(0,dim = c(H,W,4))
```

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

  for(i in 1:1000) {
    foo <- q_learning(gamma = 0.6, beta = j, start_state = c(1,1), tr = 0)
    reward <- c(reward,foo[1])
  }

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




Q−table after 1000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.2)
x
y
−1
10


Q−table after 1000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.4)
x
y
−1
10


Q−table after 1000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.66)
x
y
10
10.0
7.5
5.0
2.5
0.0