Lab3

Hugo Axandersson, Wuhao Wang & Kangbo Chen

2022/9/29

Contribution

Every question was discussed in group and the code was combined.

Environment A

Part A.1: Implement the greedy and ϵ-greedy policies

The following functions was written to implement the greedy and ϵ-greedy algorithm.

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

  # Get a greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.
  max_act = which(max(q_table[x,y,]) == q_table[x,y,])
  return(ifelse(length(max_act) ==  1, max_act, sample(max_act, 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}.
  return (ifelse(runif(1)>epsilon, GreedyPolicy(x, y), sample(1:4,1)))
}
```

Part A.2: Implement the Q-learning algorithm

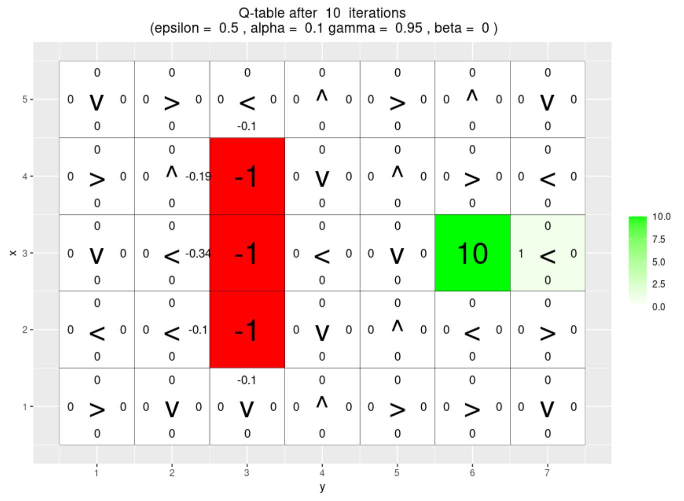
The Q-learning algorithm was implemented with the following code:

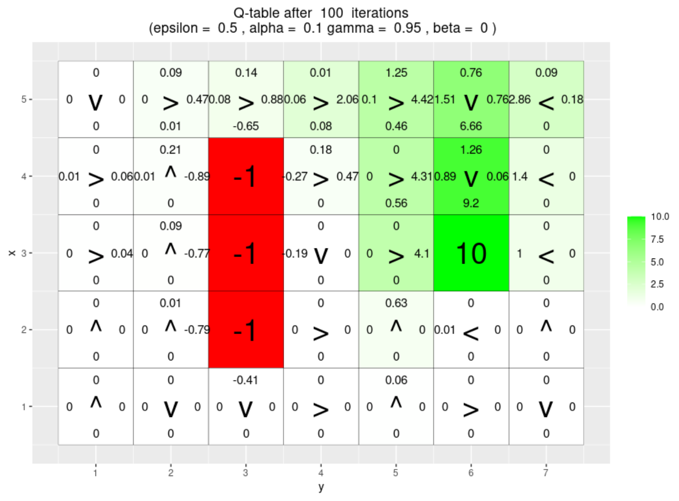
```r
q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                       beta = 0){
```

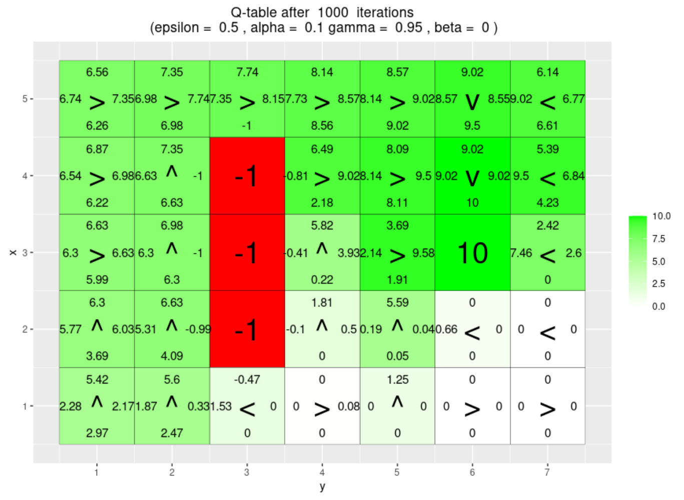
```
  # Perform one episode of Q-learning. The agent should move around in the
  # environment using the given transition model and update the Q-table.
  # The episode ends when the agent reaches a terminal state.
  #
  # Args:
  #   start_state: array with two entries, describing the starting position 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 <<-.

  state = start_state
  episode_correction = 0
  repeat{
    # Follow policy, execute action, get reward.
    x = state[1]
    y = state[2]
    act = EpsilonGreedyPolicy(x, y, epsilon)
    new_state = transition_model(x, y, act, beta)
    x_new = new_state[1]
    y_new = new_state[2]

    reward = reward_map[x_new, y_new]
    # Q-table update.
    temp_diff = reward + gamma * max(q_table[x_new,y_new,]) - q_table[x, y, act]
    q_table[x, y, act] <<- q_table[x, y, act] + alpha * temp_diff

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

}
```

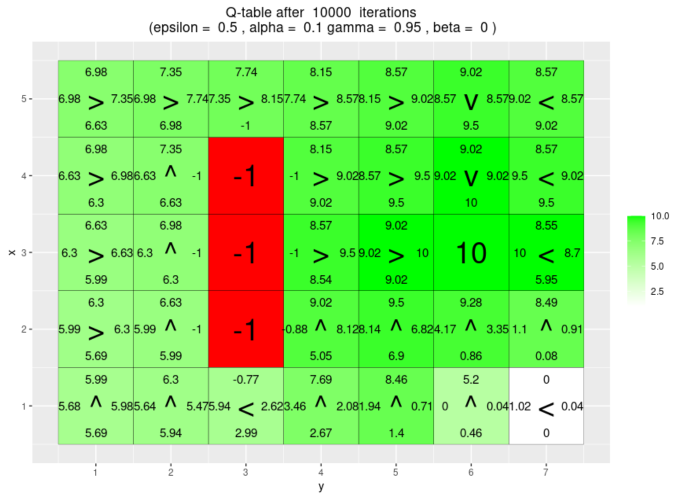
Part A.3: Visualize the Q-table and a greedy policy derived from it after episodes 10, 100, 1000 and 10000.

The Q-learning was run for 10 000 episodes with $\epsilon = 0.5$, $\beta = 0$, $\alpha = 0.1$ and $\gamma = 0.95$. The Q-table was plotted along with arrows pointing using greedy policy for the episodes 10, 100, 1000 and 10 000.









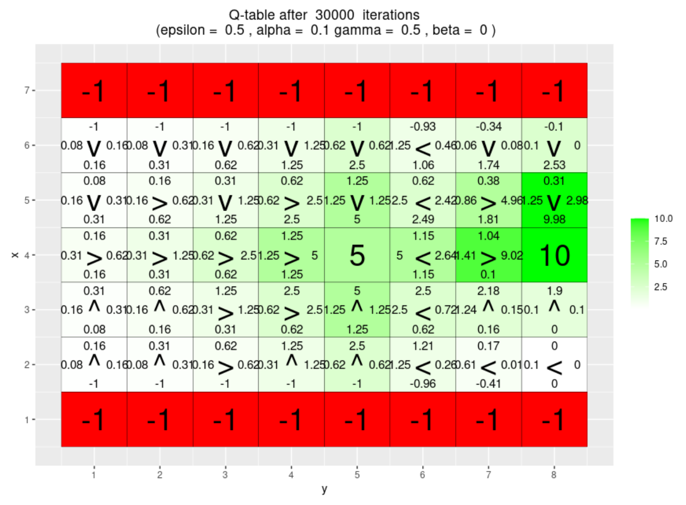
Part A.3.1: What has the agent learned after the first 10 episodes ? After 10 iterations very few values in the Q-table has been filled with the only ones being values for going directly to the endpoint from a nearby tile. This is because the algorithm hasn't been able to explore all that much already.

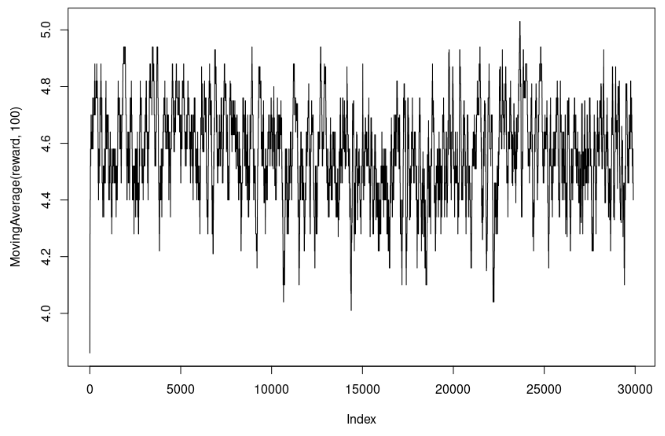
Part A.3.2: Is the final greedy policy (after 10000 episodes) optimal for all states, i.e. not only for the initial state ? Why / Why not ? Most of the tiles has an optimal policy with the exception being the tiles in the lower left that could have went below the "wall" to reach the end tile faster.

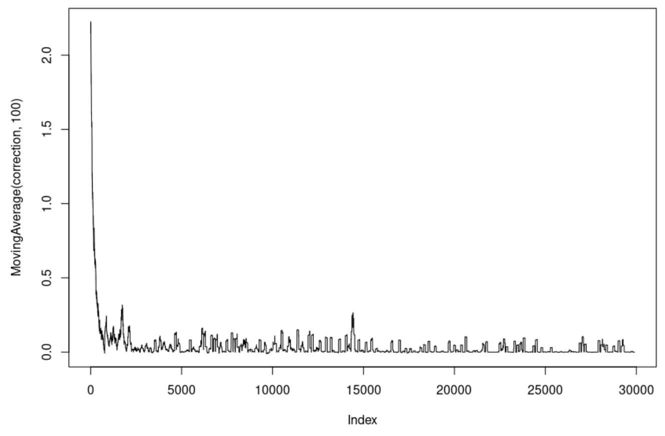
Part A.3.3: Do the learned values in the Q-table reflect the fact that there are multiple paths (above and below the negative rewards) to get to the positive reward ? If not, what could be done to make it happen ? The reason for going below the negative values might be because the algorithm very early on (see iteration 100 and 1000) randomly (thanks to ϵ-greedy) went for the top path this resulted in a bit higher q-values which later on snowballed into a policy that prioritized the top path (because the bottom path was more rarely explored). This could maybe be remedied a bit by using a higher ϵ value to incentivize more exploration below the negative values or with more iterations.

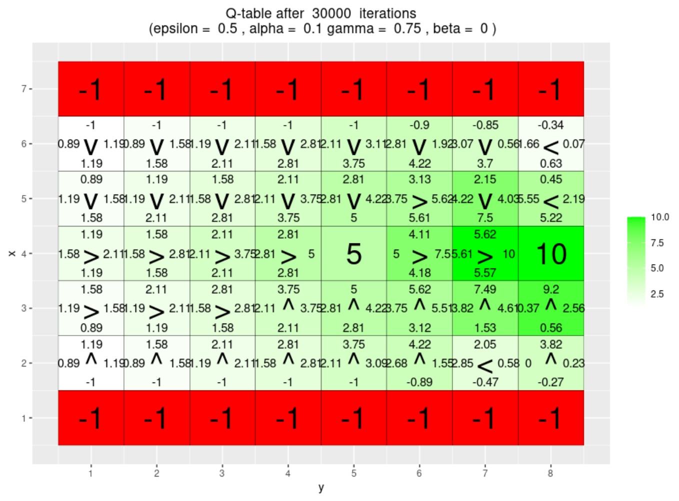
Environment B

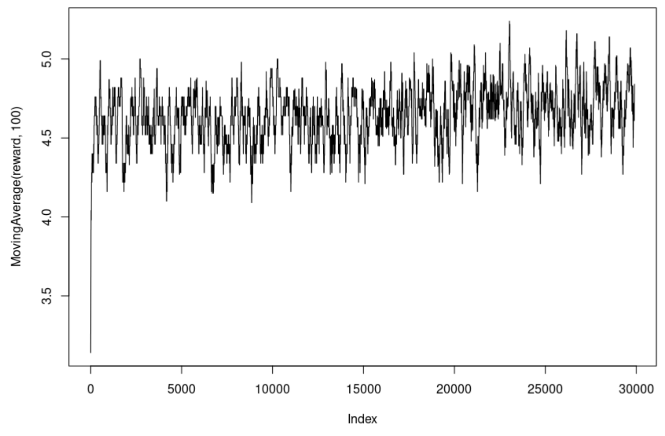
The following plots resulted from using ϵ = 0.5, γ =0.5, 0.75, 0.95. The moving averages was also created for the corrections and rewards.

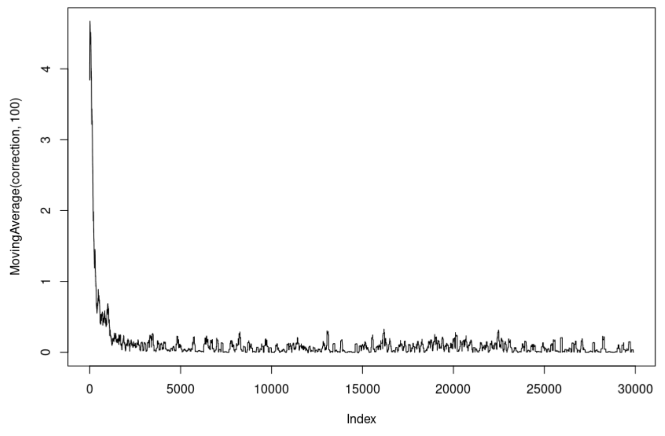

Q-table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.5 , beta = 0)
x
y

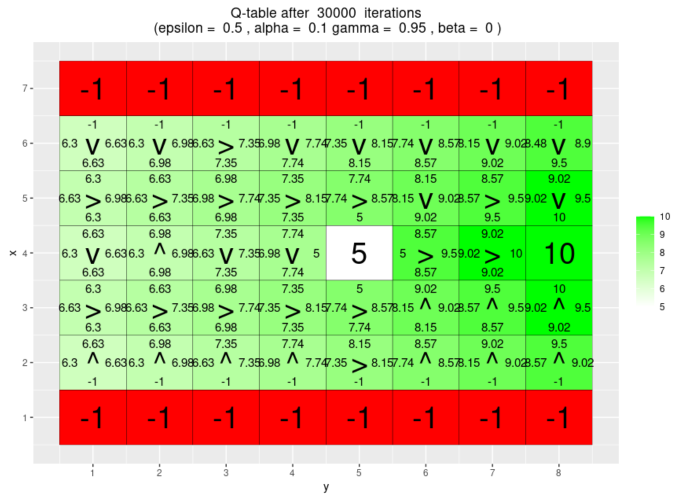

MovingAverage(reward, 100)
Index

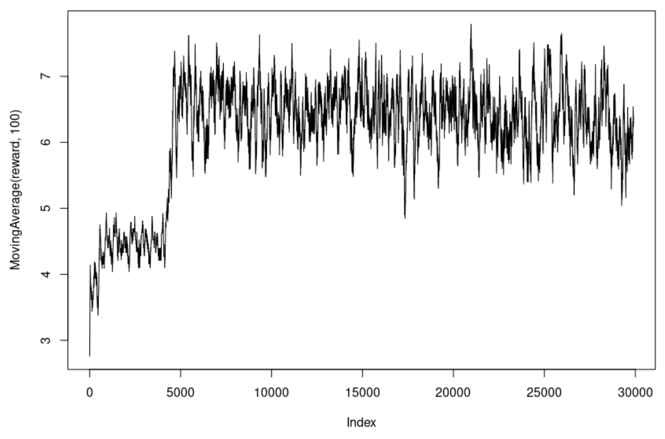

MovingAverage(correction, 100)
Index

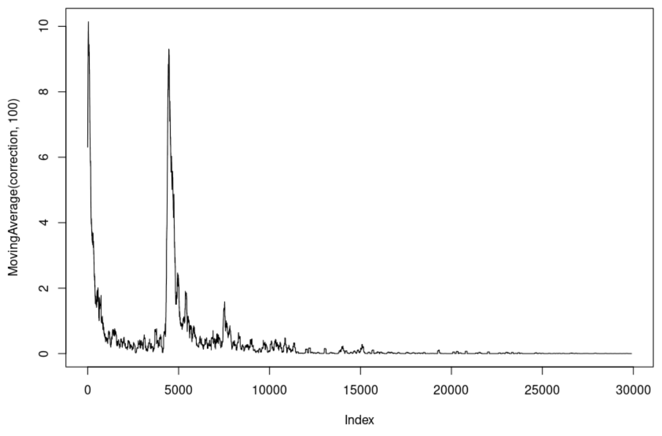

Q-table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y


MovingAverage(reward, 100)
Index


MovingAverage(correction, 100)
Index

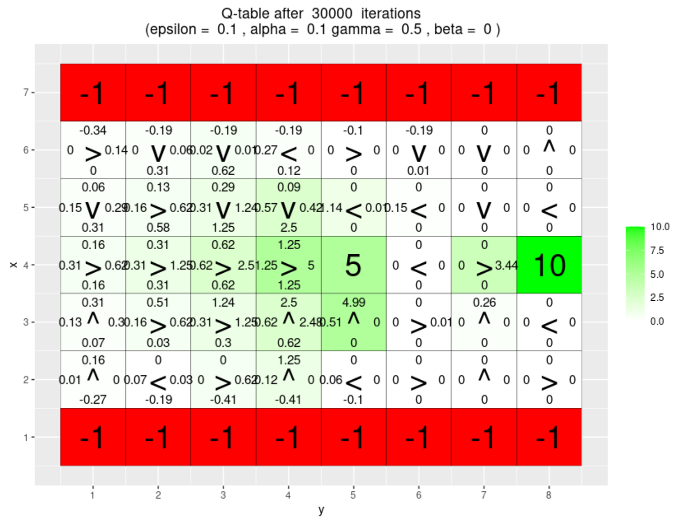


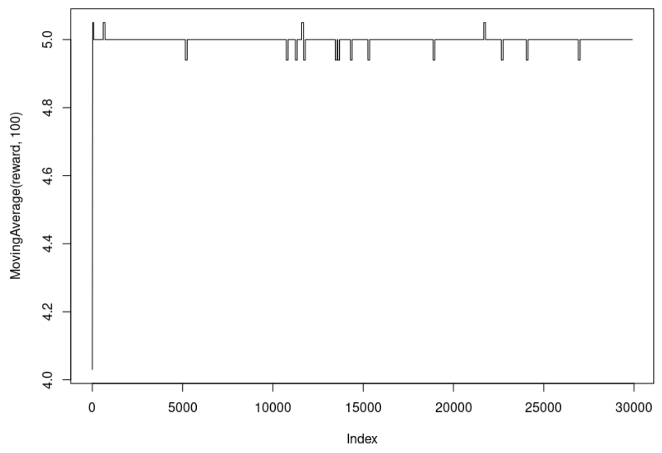


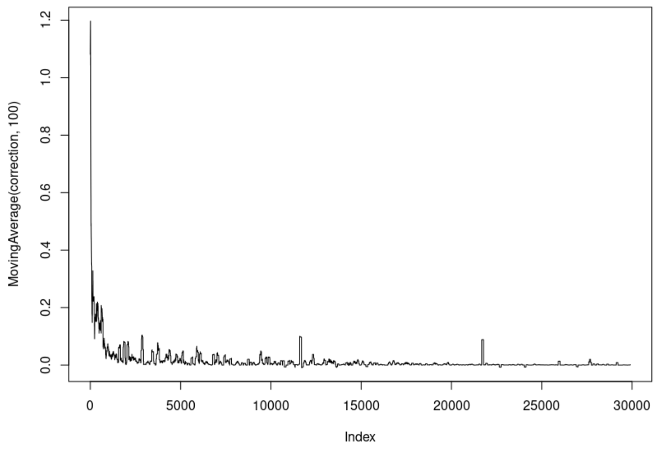


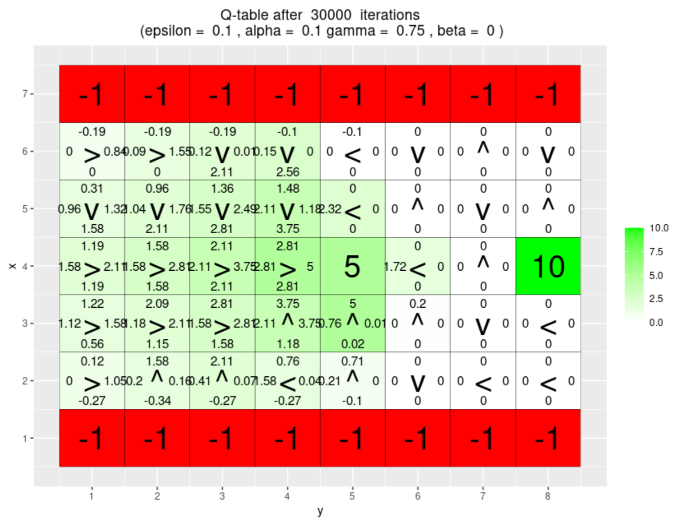
As we can see the moving averages was very similar for γ=0.5 and 0.75. There was some differences for the policies for the tiles very close to the 10 tile where γ=0.75 prioritized the 10 tile more. However the resulting greedy policy will never reach the 10 tile because all path from the start tile goes to the 5 tile (both for γ=0.5 and 0.75). The moving averages and the greedy policy for the γ=0.95 was different from the others. The greedy policy never went to the 5 tile and always went for the 10 tile. The moving average for the reward had a uptick after around 5000 iterations this might be the point were the algorithm "found" the 10 tile and the greedy policy started to point there. The same thing can be noticed from the moving average of the correction where we see a high peak around the same iteration which probably means a lot of the Q-table was updated to now point to the "new found" 10 tile.

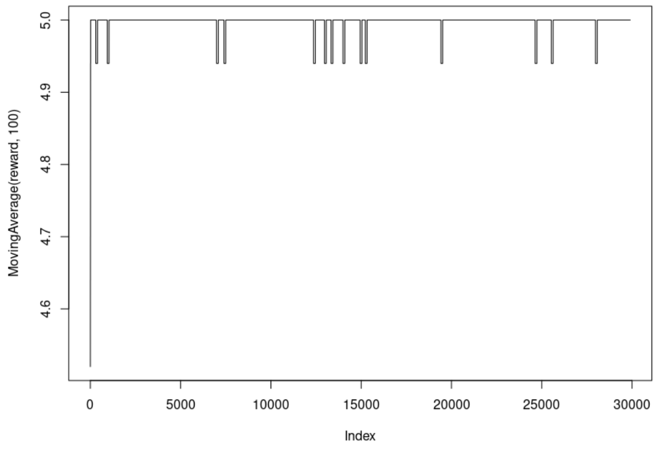
The following plots resulted from using ϵ = 0.1, γ=0.5, 0.75, 0.95. The moving averages was also created for the corrections and rewards.

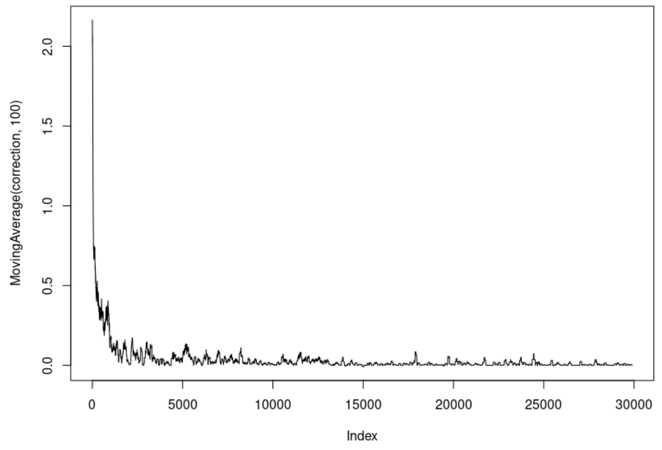

Q-table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.5 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0

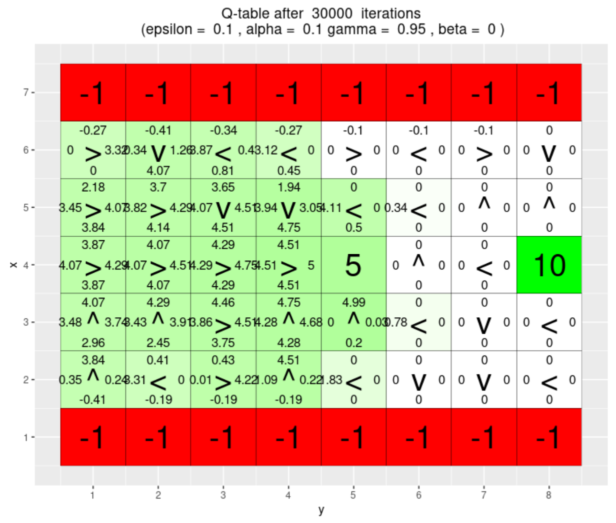

MovingAverage(reward, 100)
Index

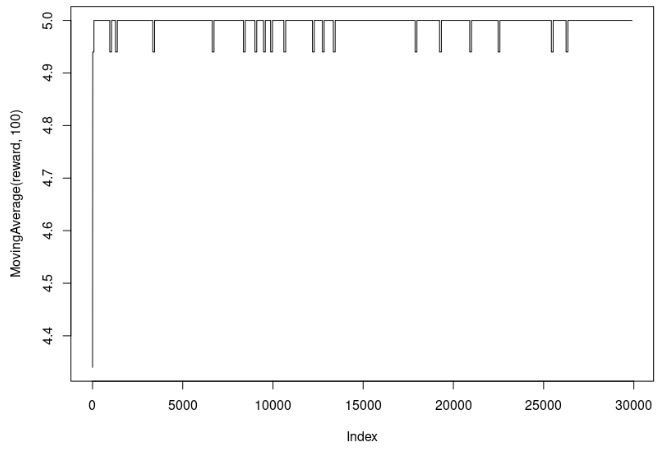

MovingAverage(correction, 100)
Index

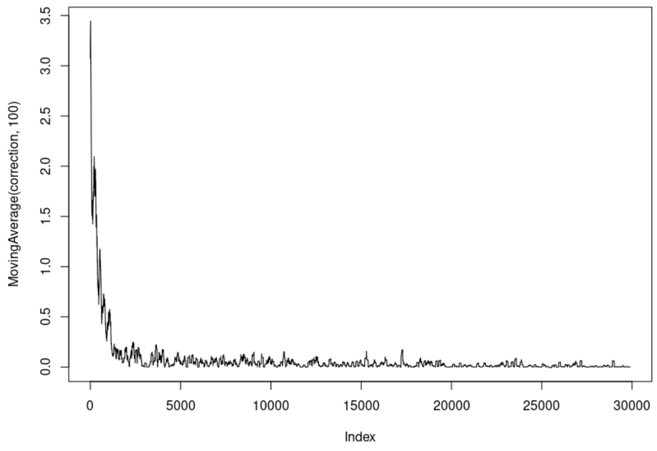

Q-table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
Index


MovingAverage(correction, 100)
Index



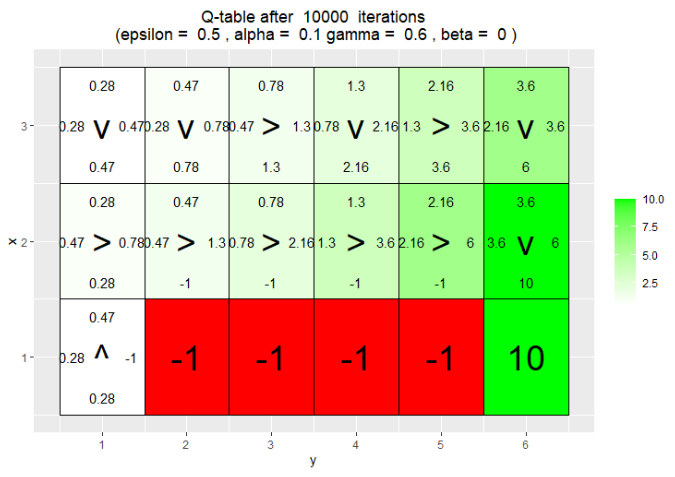


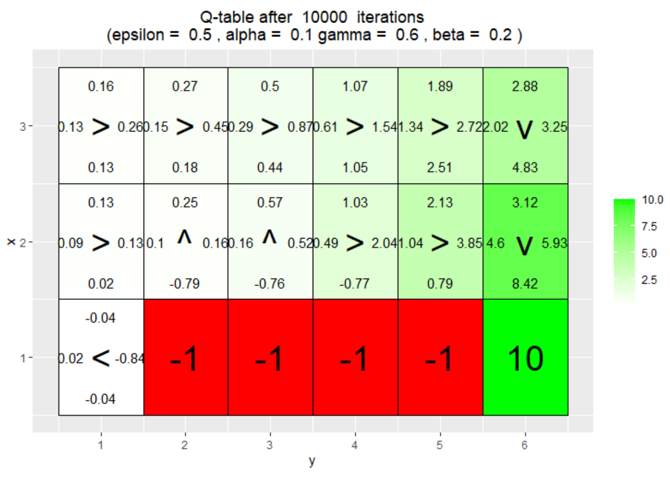


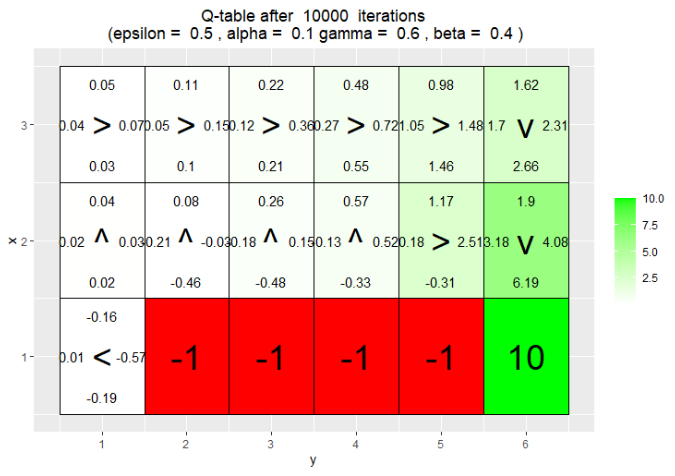
As can be seen from the moving averages for the rewards the iterations almost always end up in the 5 tile for all γ. The moving averages for the correction indicates that the algorithm converged pretty quickly (after around 2500 iteration) for all γ. The low ϵ hinders exploration by a lot as the algorithm will almost always follow the highest q-value path. The only γ that "found" the 10 tile was γ=0.5 (which was probably a combination of luck and the fact that all values in the q-table are so low). As we can see from the previous $\epsilon = 0.5$, γ=0.95 case that γ-value did prefer a path to the 10 tile. Therefore in the $\epsilon = 0.1$, γ=0.95 case the algorithm would probably have made the optimal path to the 10 tile if it was lucky enough to "find" the 10 tile.

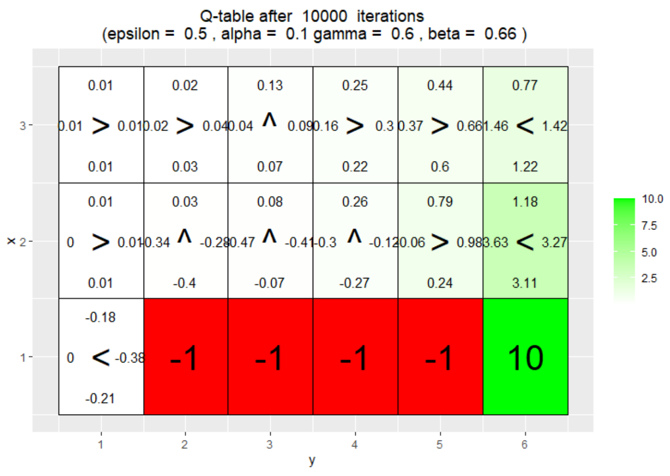
Environment C

The following plots resulted from β=0, 0.2, 0.4 and 0.66.







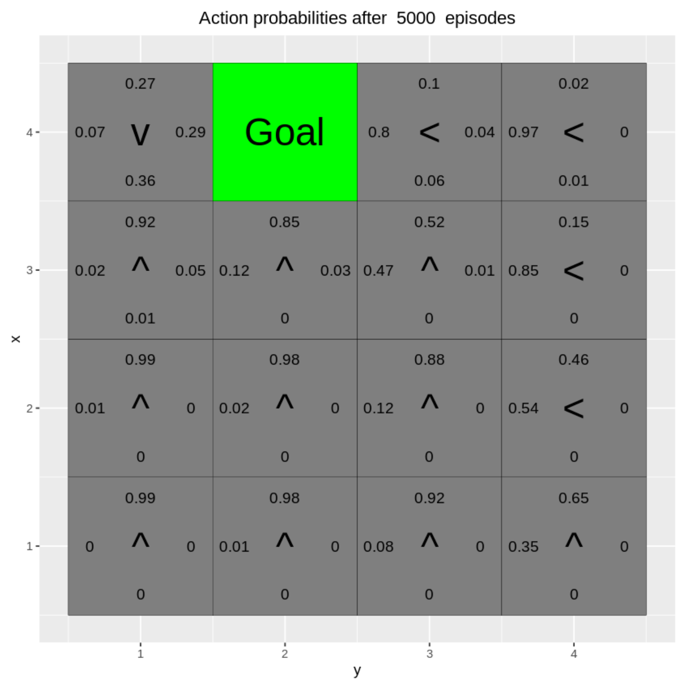


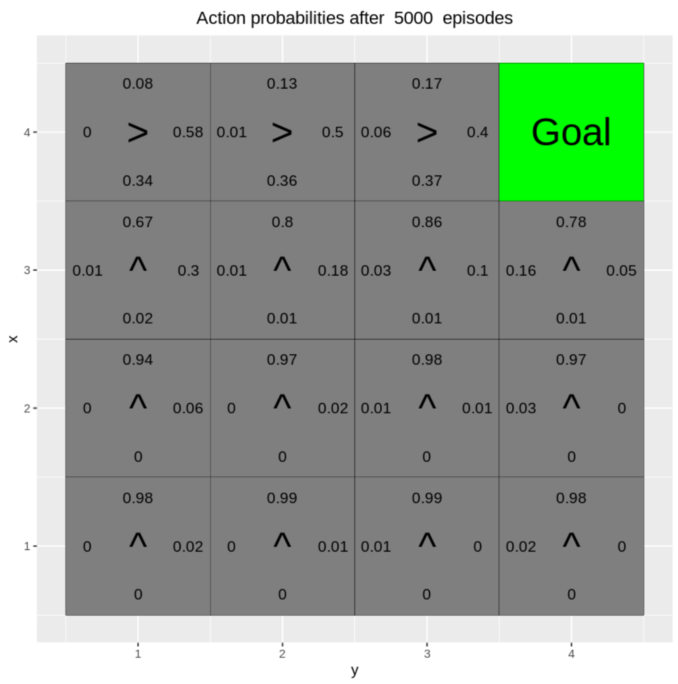
The $\beta = 0$ case is the simplest. The agent always walk its intended way and never slips. It therefore makes sense that all arrows for $\beta = 0$ points to the fastest way to the 10 tile. For $\beta = 0.2$ the agent moves away from the -1 tiles at the start. This is the case because it's not worth to go for the fastest path early on if you risk slipping into a -1 tile. The $\beta = 0.4$ case is very similar to the $\beta = 0.2$ case but this time two more arrows point away from the -1 tiles and avoids the fastest path.

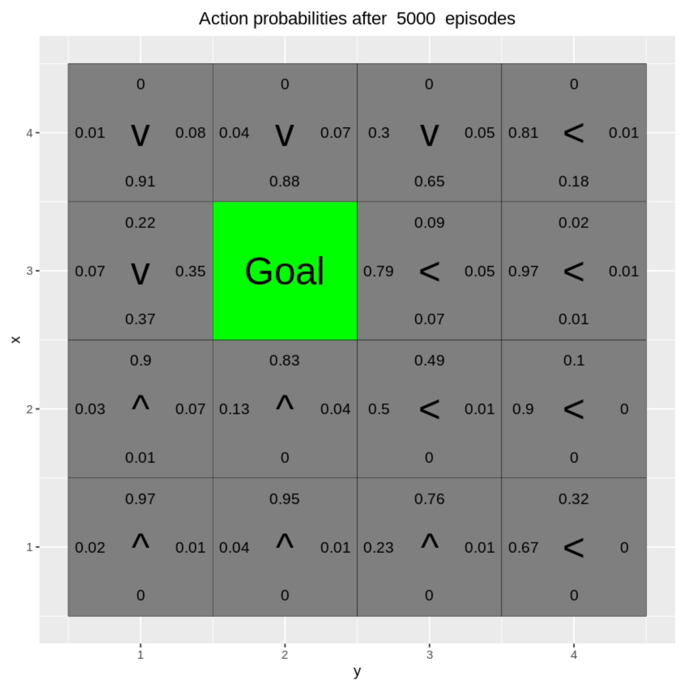
For $\beta = 0.66$ the arrows start pointing a little weirdly (towards the -1 tiles and away from the fastest path). This is probably the case because there is so much randomness in which way to go (the agent is equally likely to go in one of three different paths). It therefore doesn't matter as much which way the agent goes as it is unlikely that it will actually move in the intended directions. This makes it so that the q-values "doesn't matter" as much and are therefore lower. This can also be seen from the other β as higher β means higher q-values in the q-table as you have more certainty that a direction is more optimal than another. Even if some of the arrows for the $\beta = 0.66$ case seem to be pointing in a suboptimal direction due to all the randomness it is still clear that the arrows seem to have some general trend away from the -1 tiles and towards the 10 tile, the algorithm tries to maximize the expected value of the direction. With more iterations the $\beta = 0.66$ case will probably have converged to a better policy.

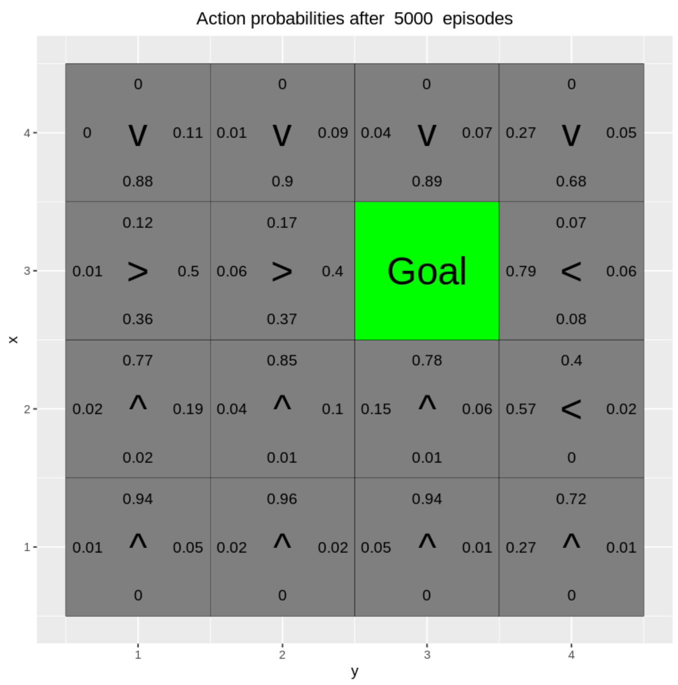
Environment D

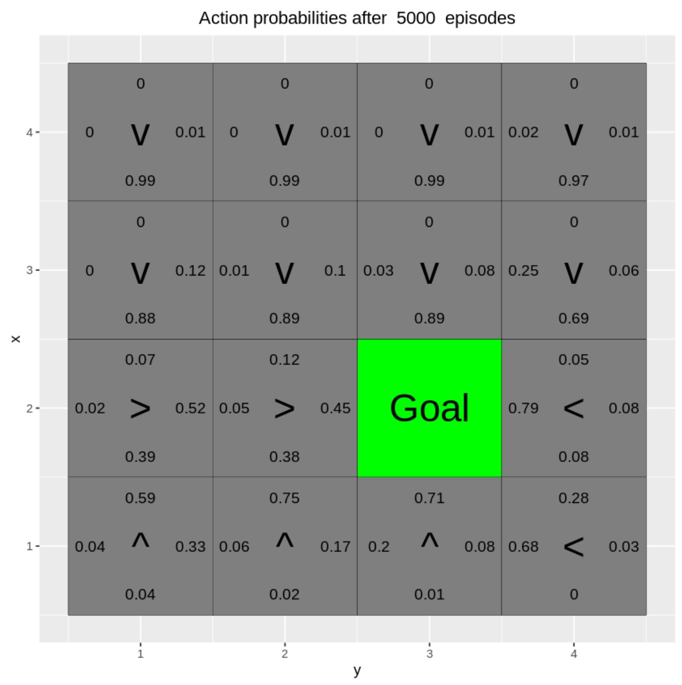
The different goal tiles resulted in the following plots.

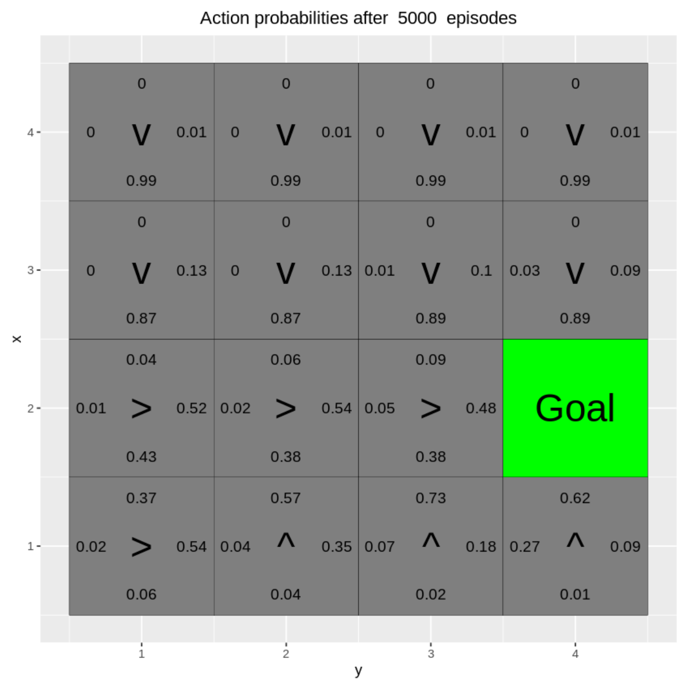

Action probabilities after 5000 episodes
Goal
x
y

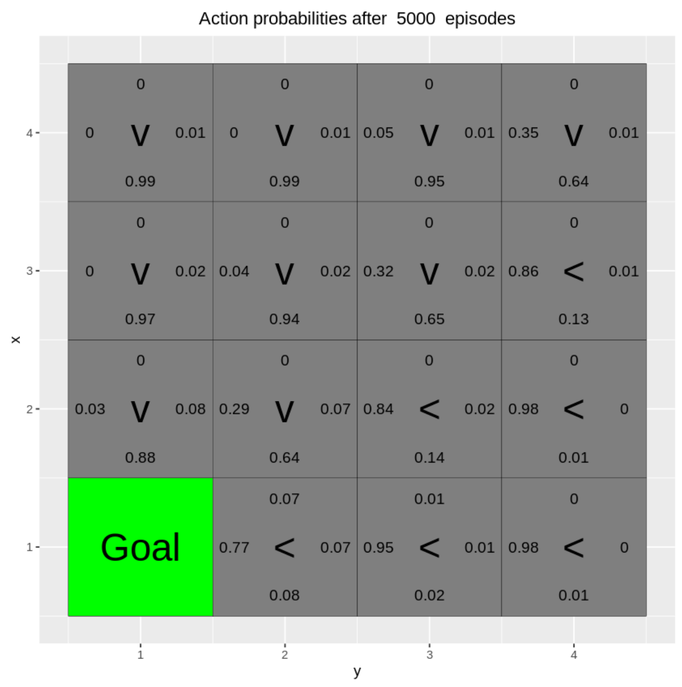

Action probabilities after 5000 episodes
Goal
x
y

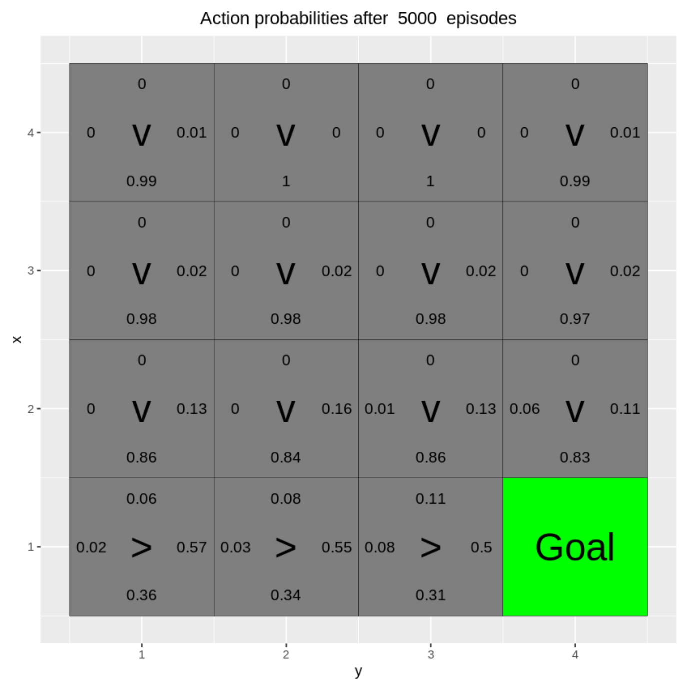

Action probabilities after 5000 episodes
Goal
x
y


Action probabilities after 5000 episodes
Goal
x
y









Part D.1: Has the agent learned a good policy? Why / Why not ?

All arrows (with the exception of one) for all goal tiles pointed in an optimal direction to reach the end goal the fastest. This indicates that the agent learned a good policy

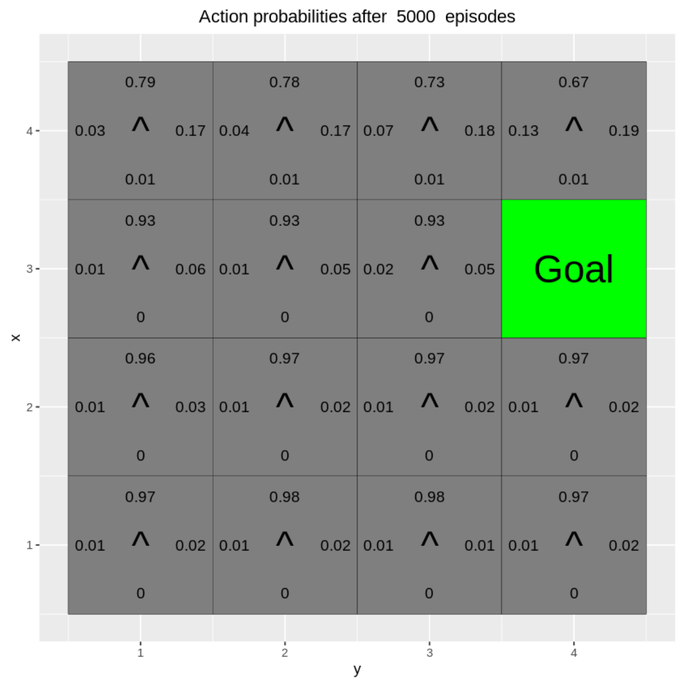
Part D.2: Could you have used the Q-learning algorithm to solve this task ?

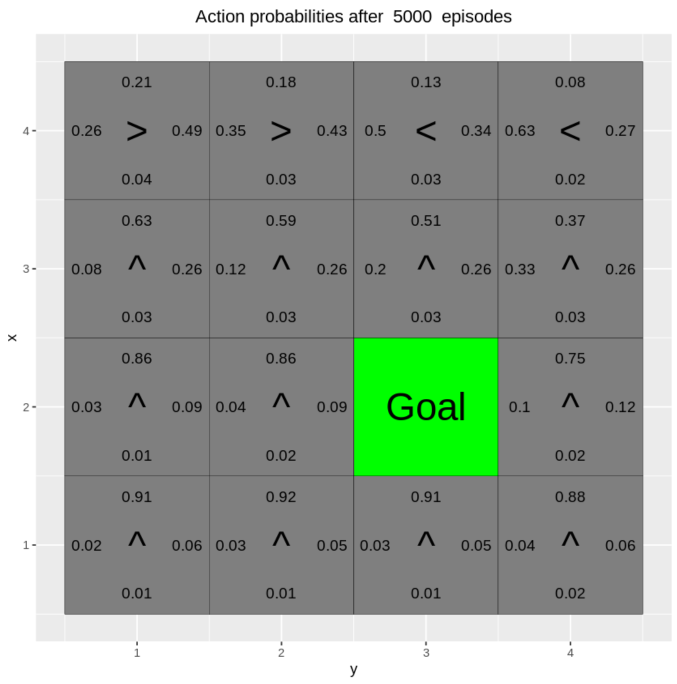
Q-learning would have had very poor results for this task as Q-learning only learns a static environment with a static end goal.

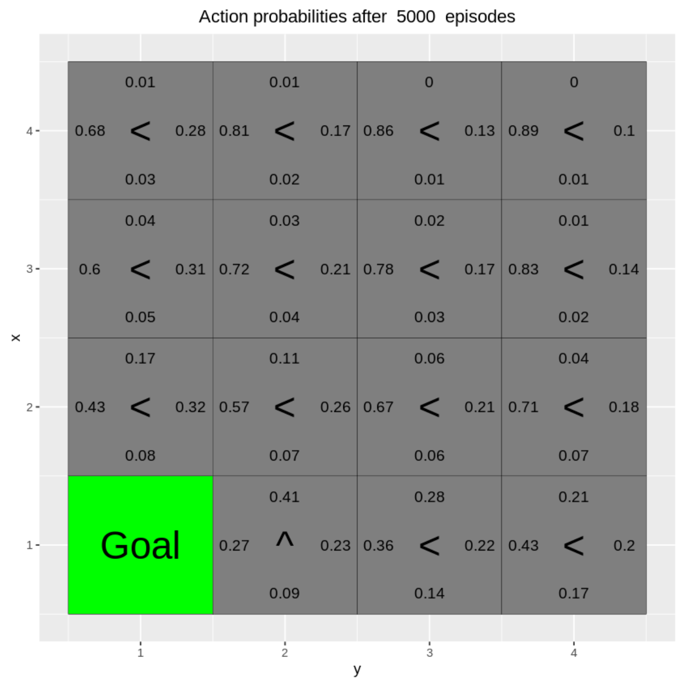
This is because Q-learning uses a single static Q-table. This table only depends on the goal nodes it trains on. Once it has been trained it will with greedy policy always result in a single deterministic optimal policy that does not depend on changing environment or goal position. It might be possible that the trained policy passes through all tiles and therefore can handle all goal positions. That policy will for our case however not be optimal for all goal positions (as you have to take a detour to go through all tiles).

Environment E

Different goal tiles resulted in the following plots.







Part E.1: Has the agent learned a good policy? Why / Why not ?

The learned policy is very bad. Very few arrows make a path towards the goal tile.

Part E.2: If the results obtained for environments D and E differ, explain why.

The reason why the policy turned out so bad for Environment E was probably because the algorithm only trained on goal tiles on the top row and therefore doesn't know what the optimal way to act for goal tiles on the other rows is. Environment D however trained on goal tiles on all different rows and columns so it was able to interpolate the optimal path for those goal tiles to a policy for goal tiles for the whole grid.

Appendix

```
knitr::opts_chunk$set(echo = TRUE)
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_act = which(max(q_table[x,y,]) == q_table[x,y,])
  return(ifelse(length(max_act) ==  1, max_act, sample(max_act, 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}.
  return (ifelse(runif(1)>epsilon, GreedyPolicy(x, y), sample(1:4,1)))
}
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 <<-.

  state = start_state
  episode_correction = 0
  repeat{
    # Follow policy, execute action, get reward.
    x = state[1]
    y = state[2]
    act = EpsilonGreedyPolicy(x, y, epsilon)
    new_state = transition_model(x, y, act, beta)
    x_new = new_state[1]
    y_new = new_state[2]

    reward = reward_map[x_new, y_new]
    # Q-table update.
    temp_diff = reward + gamma * max(q_table[x_new,y_new,]) - q_table[x, y, act]
    q_table[x, y, act] <<- q_table[x, y, act] + alpha * temp_diff

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

}
```