# Implementing MAXQ and Qlearning methods in R and applying them to the taxi problem *

Mihai Groza
Concordia University
fill this

Khaled Fouda
Concordia University
khaledsfouda@gmail.com

December 6, 2020

**Abstract**

*last thing to do.*

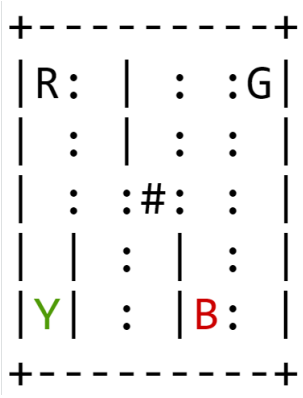



**Figure 1:** *generated by render(s)*

## I. Introduction

later.

## II. Environment

The taxi's - shown as # - goal is to first pick up the passenger from one of the four places (R,Y,G,B) where the current location of the passenger is shown in green. After picking up the customer, the next and final goal is to drive him to his destination which is one of the four places (R,Y,G,B) and the destination is shown in red.

The state is defined as a set (taxi_row, taxi_column, passenger_location, destination_location) where:
taxi_row and taxi_column take values from 1 to 5 as we have 5x5 grid,
passenger_location and destination_location take values [1,2,3,4] for [R,Y,G,B], furthermore, if the passenger is in the taxi then they takes a location 5.
That being said, we have $5 * 5 * 5 * 4 = 500$ different states.

The set of actions available is ( North, South, East, Wast ) with ids 1 to 6.

If the taxi successfully dropped off the passenger then the episode is over and they receive a reward of +20, if they dropped the passenger at a wrong location or before picking them up first then they receive a reward of -10 and continue the episode to drop the passenger at the right location (ie. no change in the state). Similarly if they attempted to pick the passenger at the wrong place. Otherwise, they receive a reward of -1 for each step taken. Note that hitting the wall

*Reinforcement learning undergraduate class project

results in a reward of -1 and no change in the state.

I have implemented the following functions for the environment:

- render(state) : returns nothing and prints out the environment at the current state.
- encode(s) : maps each state to an integer between 1 and 500.
- decode(i): The inverse of encode(s). it takes an integer between 1 and 500 and returns the corresponding state.
- loc.indx(i): maps the four pick/drop locations (1,2,3,4) to a (row,column) set.
- hitting.wallQ(r,c,a): returns True if taking the action (a) at the location (r,c) would results in hitting the wall.
- step(s,a) returns (s',r) at state s, it takes the action a, observes the reward r and the next state s'

## III. MAXQ method

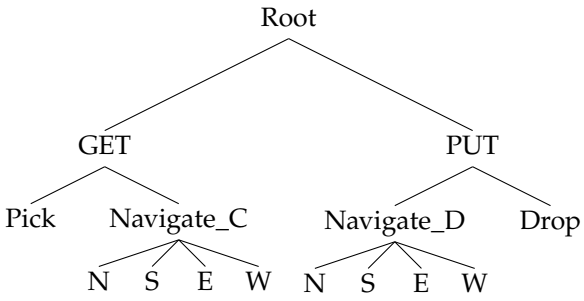
In our problem, the agent's goal is not only reaching the destination but is to also pick up the customer first. From this we define the following sequence of subtasks needed to reach the goal.

**Reach the customer ⇒ Pick them up ⇒ Drive to the destination ⇒ drop them off**

MAXQ algorithm is one of the hierarchical reinforcement learning methods which guarantees ultimate goal satisfaction only if a sequence of sub-goals was satisfied.
One the other hand, the algorithm's performance depends on how the structure of hierarchy is defined. MAXQ0 provides no way of learning that structure.
In MAXQ there are two types of actions: a primitive action which is the action that change the state of the environment and composite action which is a sub-task. For our Implementations, we defined the following actions:

- Primitive actions:
  1. Move one step north (N)
  2. Move one step south (S)
  3. Move one step east (E)
  4. Move one step south (W)
  5. pick-up the passenger
  6. drop-off the passenger
- subtasks actions:
  7. Root - explained below -
  8. reach the customer and pick them up (GET)
  9. drive to the destination and drop them off (PUT)
  10. drive to the customer (Navigate_C)
  11. drive to the destination (Navigate_D)

and Their hierarchical structure is defined as follows:




Where - starting at root - the agent has to recursively go down the graph satisfying each composite action. The root is satisfied if both get and put are satisfied and hence the episode ends. At each subtask i, the agent needs to repeatedly choose an action until the task is completed and hence we go back to the parent subtask.
For each action, a list of Terminal states is defined. If an action is at a terminal state, it no longer needs move down the tree and it returns back to it's parent subtask. For example,

at Navigate_C, it repeatedly choose one of the four actions [N,S,E,W] until it reaches the location of the customer. When that happens, we say that Navigate_C reached a terminal state and it stops executing it's subtask and it reports back to it's parent "GET" who either choose an action "Pick" or "Navigate_C" or if it's at a terminal state, it reports back to root and so on. The only terminal state for root is when both "GET" and "PUT" reached their terminal states and the episode is over.

## value function Estimates

Let $(i, s)$ define the state of the environment at time $t$ where $i$ is the subtask running. Let also $a$ be the action (sub-subtask) taken by subtask i at time t. Define

$$C^{\pi}(i,s,a) = (1-\alpha)C^{\pi}(i,s,a) + \alpha * \gamma^{N} * V[i,s']$$

$$V^{\pi}(i,s) = \begin{cases} (1-\alpha) * V^{\pi}(i,s) + \alpha * Reward(s'|s,i) \\ \text{if i is primitive} \\ max_{a \in A(i)} V^{\pi}(a,s) + C^{\pi}(i,s,a) \\ \text{if i is a subtask} \end{cases}$$

where $\alpha$ is the learning rate, $\gamma$ is the discounting factor, $N$ is the number of times the subtask i executed actions before reaching the terminal state, and $A(i)$ is the set of available actions to subtask i.
$C^{\pi}$ is only defined for composite actions, i.e, for $i \in (7,8,9,10)$ and it's similar to the usual action value function $Q$ and it it takes into account the value function of the next state produced by a primitive action. the term $\gamma^{N}$ is adapted from Semi-Markov decision model and it helps prioritizing the actions that leads to smaller number of steps. However, in our experiment, we got better results setting this term 1.
The state value function for the primitive actions is a classic weighted average of rewards and for the composite actions is a maximization of a modified action value function. The state value function for the composite actions requires recursive calculations of the subsequent actions chosen.

```
EvaluateMaxNode <- function(i, s){
  # if i is primitive then do nothing
  if(any(i==1:6)){return()}
  actions <- Action.space(i)
  for(a in actions){
          EvaluateMaxNode(a, s)
  }
  V[i,encode(s)] <- max(V[actions,encode(s)] +
  C[[i]][encode(s),actions])
}
```

## Policy

The policy used is $\epsilon - greedy$. Only composite actions -subtasks- can take subsequent actions and it's the action the maximizes the modified action value function above with probability $1-\epsilon$ and a random action with probability $\epsilon$ with the restriction of belonging to the action space -defined below- of that subtask.

## Action Space

The action space for each action is the set of sub-nodes related to that action. Remember that only subtasks have action spaces since states with primitive actions take a predetermined actions defined by the environment above.

```
Action.space <- function(.,i){
  # expects i to be from 7 to 13
  if(i==7){return(c(8,9))
  }else if(i==8){return(c(5,10))
  }else if(i==9){return(c(6,11))
  }else if(i==10 || i==11){return(1:4)}
}
```

## Terminal States

Each subtask -subtree- runs recursively until it hits one of its terminal states. Terminal states do not necessarily represent the completion of that subtask, they can be states where the subtask is not needed at that time step. We defined the following terminal states:

- When either the subtask Get or Navigate_C is active and the customer being already picked up.
- When either the subtask Put or Navigate_D is active and the customer is not yet picked.
- When either of the navigation subtasks is active and the taxi has already arrived the destination.
- When the episode is over and the customer have made their trip successfully.

As for the primitive actions, they are always executed once and report back to their parent-node who then can make choice of whether it execute them again using the defined policy.

## Learning Algorithm

**Algorithm 1** MAXQ0(node i, state s)

**if** i is a primitive action **then**
  execute the pair $(i, s)$ and observe the resulting $(i, s')$
  $V(i, s) \leftarrow (1 - \alpha) * V(i, s) + \alpha * r$
  $return(1)$
**else**
  $count \leftarrow 0$
  **while** (i,s) not a terminal state **do**
    choose a according to the policy
    $N \leftarrow MAXQ0(a, s)$
    **if** episode is done **then**
      $return(count)$
    **end if**
    $EvaluateMaxNode(i, s)$
    $C(i, s, a) \leftarrow (1 - \alpha)C(i, s, a) + \alpha * \gamma^N * V[i, s']$
    $count \leftarrow count + N$
    $s \leftarrow s'$
  **end while**
**end if**

The last function when called with an initial $i$ = 7-Root- it simulates a full episode from the beginning until the successful drop off of the passenger. For larger scale problems and poorer design of the environment, the while loop might run forever before reaching a terminal state.

## IV. Qlearning method

## V. Results

## VI. Similar projects

## VII. Final thoughts

## References

[Figueredo and Wolf, 2009] Figueredo, A. J. and Wolf, P. S. A. (2009). Assortative pairing and life history strategy - a cross-cultural study. *Human Nature*, 20:317–330.