

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 4, 2020

Abstract

last thing to do.

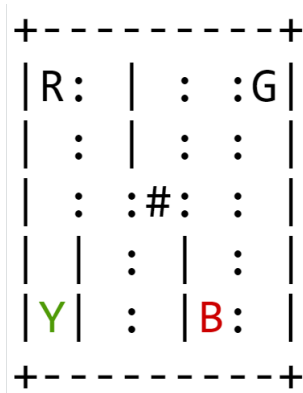


Figure 1: generated by render(s)

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, West) 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 \Rightarrow Pick them up \Rightarrow Drive to the destination \Rightarrow 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.

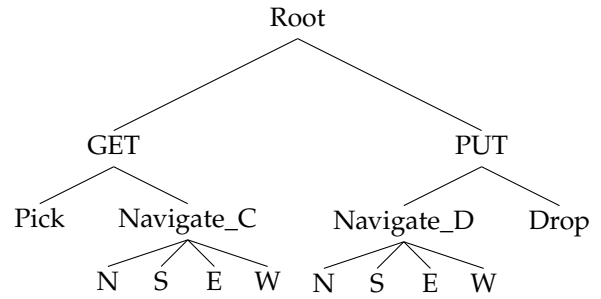
One the other hand, the algorithm's performance depends on how the structure of hierarchy is defined. MAXQ0 provides no way of learning that structure.

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

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

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

where α is the learning rate, γ 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 .

Policy

Implementation

sefsef
sefsef
sefes
sfseef sfesef
sefsef
sefes
sfseef sfesef
sefsef
sefes
sfseef sfesef
sefsef
sefes
sfseef sfesef

sefsef
sefes
sfseef sfesef
sefes
sfseef sfesef
sefsef
sefes
sfseef sfesef
sefsef
sefes
sfseef sfesef
sefsef
sefes
sfseef sfesef

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.