Radboud University

Search, Planning and Machine Learning

Assignment 4

Value Iteration & Q-Learning

Authors:
Johannes-Lucas Löwe s1013635
Alexander Zehetmaier s1013644

January 12, 2019

Contents

1 Introduction **2**

2 Design **2**

2.1 Environment . . . 2

2.1.1 Adapted Environment from Denny Britz + new Front-end . . . 2

2.1.2 openAI Gym Taxi-v2 Enviroment . . . 3

3 Implementation **3**

3.1 Value Iteration . . . 3

3.1.1 Mathematical . . . 3

3.1.2 Pseudo-Code . . . 4

3.1.3 Python . . . 5

3.2 Q-Learning . . . 6

3.2.1 Mathematical . . . 6

3.2.2 Pseudo-Code . . . 6

3.2.3 Python . . . 7

4 Testing **8**

4.1 Value Iteration . . . 8

4.2 Q-Learning . . . 9

4.3 Taxi Environment . . . 10

5 Conclusion **11**

Appendices **12**

A Pictures **12**

B Code **19**

B.1 Class for rendering the environment . . . 19

B.2 Class for the Gridworld Logic (Denny Britz extension) . . . 21

1 Introduction

This report investigates in explaining the value iteration and Q-learning algorithm which are both used to cleverly estimate the optimal policy [1]. In the course of this report, the environments used and the algorithm implemented in python3 will be described. As there was no official framework provided we used openAI gym and an rewritten and extended version of an example grid-world Implementation as well as our own graphic interface.

2 Design

2.1 Environment

In order to focus on the algorithm we used minimal python frameworks to facilitate our development. First gym from openAI was used and a grid-world file, written by Denny Britz [2]. It was adapted with our own front-end library and most of it has been rewritten to make it fit our needs. Secondly the Taxi environment from openAI gym was used. Here a re-implementation of value-iteration and an implementation for Q-learning were made, in order to make a valid comparison between these two.

2.1.1 Adapted Environment from Denny Britz + new Front-end

The environment is mainly based on the grid-world written by Denny Britz [2]. However it only supports output on console line and no changes to the environment parameters. Neither does it support obstacles.The console graphics do not make for clear visuals. In order to overcome these issues, next to many adaptations, a new front-end has been designed. It consists of the following color-coding:

Color	Meaning
Green	Goal
Blue	Agent [not used]
Black	obstacle
violett to red	color coded normal fields, red being the smallest

Table 1: Color Codes of new Front-end

2.1.2 openAI Gym Taxi-v2 Enviroment

The environment provided by openAI gym is similar to the one provided by the java example code. As such the application can access transition probabilities for value iteration and perform actions for Q-learning. the main effort is put in the implementation of the algorithms themselves. Output is available per console, but it is coded nicely. Simply said the task is to handle a taxi on a 5x5 field with walls. There are 4 locations (titled R,G,Y,B). For each scenario a person is at one of the points and need to go to one of the other. Therefore the state-space is made up as follows: There are 25 fields. For each field there are 4 goal locations and (4+1) passenger locations, i.e. the original point or in the Taxi. Therefore it consists of $25 * 4 * 5 = 500$ states.
In each of them there are 6 Actions available. So the shape of the array describing the Q-Table must be $(500, 6)$. A clear picture of that can be seen in Figure 1.

3 Implementation

The implementation was made in python3, algorithms have been implemented in a Jupyter Notebook, the Graphical Library and the Grid-world in a python3 file that can be loaded as module.

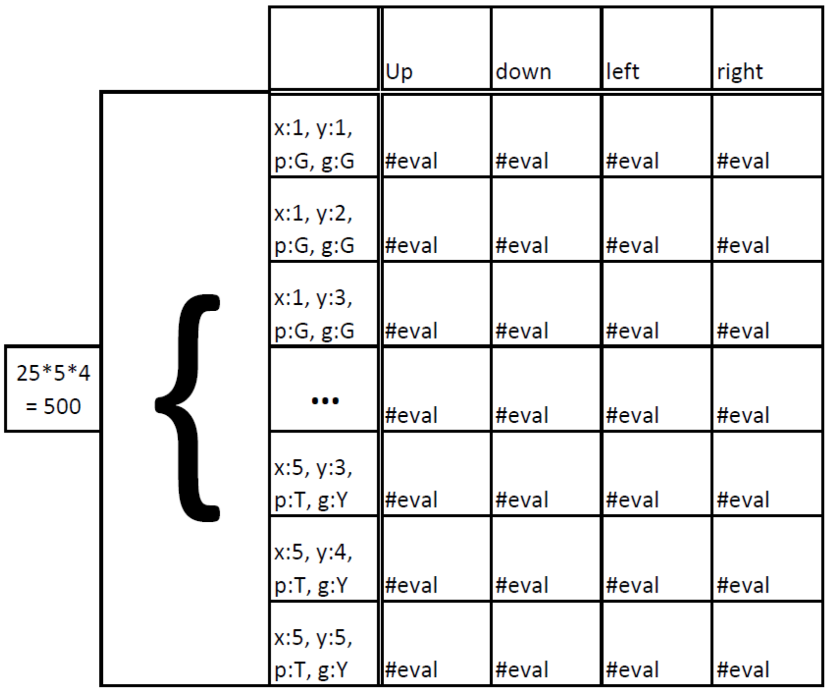
3.1 Value Iteration

In Value Iteration the algorithm can fully observe the environment. Based on that it build a value-table and to determine a policy to get the highest reward.

3.1.1 Mathematical

Value iteration approximates the optimal policy by looking at all state, action combinations and updates the q-value according to the bellman equations as shown below [1].

$$Q(state, action) \leftarrow reward + \gamma \sum_{state'} P(state' \mid state, action) V(state')$$



	Up	down	left	right
x:1, y:1, p:G, g:G	#eval	#eval	#eval	#eval
x:1, y:2, p:G, g:G	#eval	#eval	#eval	#eval
x:1, y:3, p:G, g:G	#eval	#eval	#eval	#eval
...	#eval	#eval	#eval	#eval
x:5, y:3, p:T, g:Y	#eval	#eval	#eval	#eval
x:5, y:4, p:T, g:Y	#eval	#eval	#eval	#eval
x:5, y:5, p:T, g:Y	#eval	#eval	#eval	#eval

Figure 1: State-space Taxi Environment

After each state for all actions, $V(state)$ gets updated with the maximum action from the current q-values.

$$V(state) \leftarrow \max_{a}\big(Q(state, action)\big)$$

It stops repeating this process when the absolute difference between $V(state')$ and $V(state)$ from one update to the other is less than some small given value θ.

3.1.2 Pseudo-Code

1. Initialize V(s) and V(s') to different arbitary values. Otherwise the while loop below would not execute
2. Initialize a policy arbitary where the row-size euqals the states & the column-size equals the actions

3. Repeat 4-9 till difference of V(s) and V(s') is smaller than some small specific amount θ

4. V(s) updates to be V(s')

5. Go through all states and in every state look which action are available

6. Make a list where all the q-values for the possible state, action pairs will be put in

7. Put the q-values according to the bellman equations into the list created in step 4

   - $Q(state, action) \leftarrow reward + \gamma \sum_{state'} P(state' \mid state, action) V(state')$

8. Update V(s') to be the maximum value from the q-value list

9. One-hot-encoding the the action we take and reasing solution to policy array

3.1.3 Python

The implementation of the value iteration algorithm below is indicated with the steps of the pseudo-code above.

```
def bellman_equation(prob_nextState_reward_goal, gamma, V_old):
    prob_to_nextState = prob_nextState_reward_goal[0][0]
    nextState = prob_nextState_reward_goal[0][1]
    reward = prob_nextState_reward_goal[0][2]
    return reward + gamma*(prob_to_nextState*V_old[nextState])
```

```
def value_iteration(env, theta=0.0001, discount_factor=1.0):
    # 1. Initialize V(s) and V(s') to different arbitary values
    V_new = np.zeros(env.nS)
    V_old = np.ones(env.nS)
    # 2. Initialize a policy arbitary where the row-size euqals the #states
    # & the column-size equals the #actions
    policy = np.zeros([env.nS, env.nA])

    # 3. Repeat 4-7 till difference of V(s) and V(s') is smaller
    # than some small specific amount theta
    while np.all(np.abs(V_new-V_old) > theta):
        # 4. V(s) updates to be V(s')
        V_old = np.copy(V_new)
        # 5. Go through all states and in every state look
        # which action are available
        for state in range(env.nS):
            # 6. Make a list where all the q-values for
            # the possible actions are put in
            q_s_a = []   # Q(state , action)
            for action in range(env.nA):
                # 7. Put the q-values according to the bellman
                # equations into the list created in step 4
                prob_nextState_reward_goal = env.P[state][action]
                q_s_a.append( bellman_equation(prob_nextState_reward_goal ,
                    ↪ discount_factor , V_old) )
            # 8. Update V(s') to be the maximum value from the q-value list
            V_new[state] = max(q_s_a)
            # 9. One-hot-encoding the the action we take and
            # reasing solution to policy array
            policy[state]=0
            policy[state , np.argmax(q_s_a)] = 1
    return policy , V_new
```

3.2 Q-Learning

In Value Iteration the algorithm can not fully observe the environment. The Agent explores the environment and tries to build a representation of that environment in a Q-table. Based on that it then determines the optimal policy.

3.2.1 Mathematical

The key difference to value iteration is that q-learning does not have the transition probabilities and rewards given [1]. This is why value iteration is called a model-based and q-learning a model-free reinforcement learning algorithm. The formula below updates the q-table on a given state and action.

$$Q(state, action) \leftarrow (1-\alpha)Q(state, action) + \alpha\Big(reward + \gamma \max_a Q(next\ state, all\ actions)\Big)$$

Important to notice is that the reward variable in the formula is gained through the agents interaction with the environment.

3.2.2 Pseudo-Code

1. Initialize Q-Table
    - random
    - all zeros

    - with domain knowledge initialization

2. Explore vs Exploit Action
    - select an action for current state s
3. Execute selected Action → Go to s'
4. Selcet Action in s'
    - action with highest q-value
5. Update Q-Table according to the Bellman Equations
    - $Q(state, action) \leftarrow (1-\alpha)Q(state, action) + \alpha\Big(reward + \gamma \max_a Q(next\ state, all\ actions)\Big)$
6. Update the Current State to be the Next State
7. If Goal-State → New Episode

3.2.3 Python

The implementation of the Q-learning algorithm below is indicated with the steps of the pseudo-code above.

```
# Values we need before we start with the actual Algorithm
state_space = env.observation_space.n
action_space = env.action_space.n

#1. Initialize Q-Table
q_table = np.zeros([state_space, action_space])
```

```
def q_learning(env, q_table, epsilon=0.1, alpha=0.1, gamma=0.6):
    for episode in range(1, 100001):
        state = env.reset()
        done = False
        # 7. If Goal-State -> New Episode
        while not done:
            # 2. Explore vs Exploit Action
            if random.uniform(0, 1) < epsilon:
                action = random.randint(0, 5)
            else:
                action = np.argmax(q_table[state])

            # 3. Execute selected action -> Go to s'
            next_state, reward, done, info = env.step(action)
            # 4. Update Q-Table according to the Bellman Equations
            max_next_action = np.max(q_table[next_state])
            q_table[state][action] = (1-alpha)*q_table[state][action] +
                ↪ alpha*(reward+gamma*max_next_action)

            # Update the Current State to be the Next State
            state = next_state

        if episode%100 == 0:
            print(str(episode) + '_Episode')
```

4 Testing

Testing was executed mainly with respect of the parameters onto the value table. Naturally having a 500 state state-space makes it hard to draw direct conclusions. Therefore most of the testing was done on our own grid-world. Nevertheless the algorithm performed fully functional also in the taxi-environment **??**. This shows it can be generalized.

4.1 Value Iteration

The test cases for value iteration will be as follows:

trial	world	state reward	discount factor	trans. prob.	satisfying test condition	Figure
1	1	-0.01	1	1	i	6
2	1	0	1	1	iii	7
3	1	-0.1	1	1	iii	8
4	1	-1	0.9	1	ii	9
5	1	-1	0.1	1	ii	10
6	2	-1	1	1	i	11
7	3	-1	1	1	i	12
8	1	-1	1	0.5	iv	13
9	1	-1	1	0.25	iv	14

Table 2: Test Conditions for Value Iteration

As the test show, the value table works correctly. It is important to note that all shown numbers are rounded to 3 decimal places. The values get higher as the way to the goal gets longer. As one can see in the color coding, the relative value of states closer to the goals with respect to the rest changes with each trial. In trial 2, where the state penalty is 0, no change happens in the value table. In all the other test cases a valid conclusion to be drawn appears to be: *The lower either the value for the discount factor or the transition probability, the the bigger the difference gets between states closer and further away from the goal.* No effect towards that regard appears to be made by the size of the state reward.

4.2 Q-Learning

In order to asses if Q learning works, a Cumulative Reward curve for our agent has been drawn.

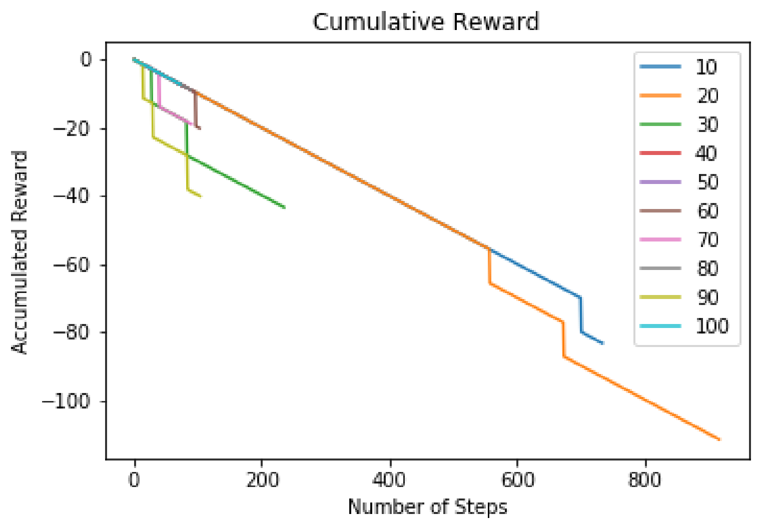
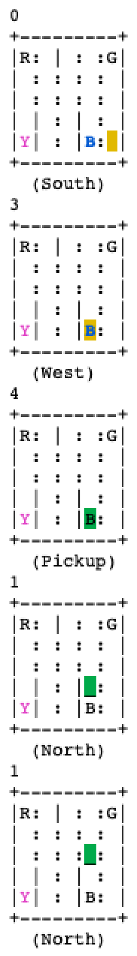


Figure 2: Cumulative Reward for our agent every 10 episodes in a 100 episode training

As the graph shows, the agent keep getting better, i.e. the negativity of the reward to finish decreases, which means a shorter path has been found. There are a few exceptions, e.g. 20. This is most likely because he ran into a wall which is punished quite strongly in our enviroment. Also in the first few episodes the greedy epsilon randomness has a bigger effect, since many states have not been visited before, leading to more random decision. Overall the graph does show that the agent get better until it reaches the optimum of 18 steps.

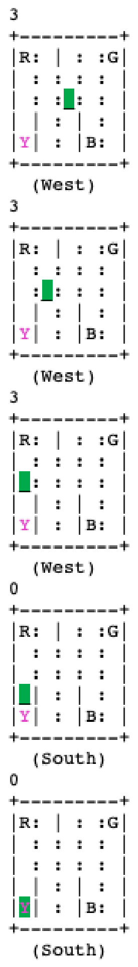
4.3 Taxi Environment

Also in the Taxi Environment a successful plan is found by our Algorithm:



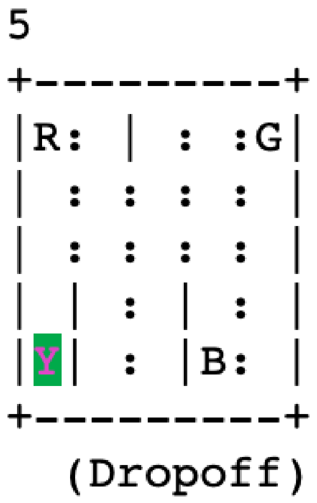
```
0
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
3
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
4
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Pickup)
1
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
1
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
```

Figure 3: Start of the taxi in a new environment



```
3
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
3
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
3
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
0
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
0
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
```

Figure 4: Person picked up by the taxi, on the way to the goal



```
5
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
```

Figure 5: Final State of the Taxi Environment, Taxi dropped off the passenger at the right goal

5 Conclusion

All in all the project has turned out with more features then required. Through the implementation in 2 environments the validity of the Algorithms has been shown. A simple representation for a grid world has also been created and therefore it can run with a minimum of dependencies. Furthermore a real-time simulation of an agent can be seen. Overall the result is an elegant solution that is adaptible to many problems. In the future the graphical implementation could be ported to a faster library and GUI controls could be added to make the visualization more interactive.

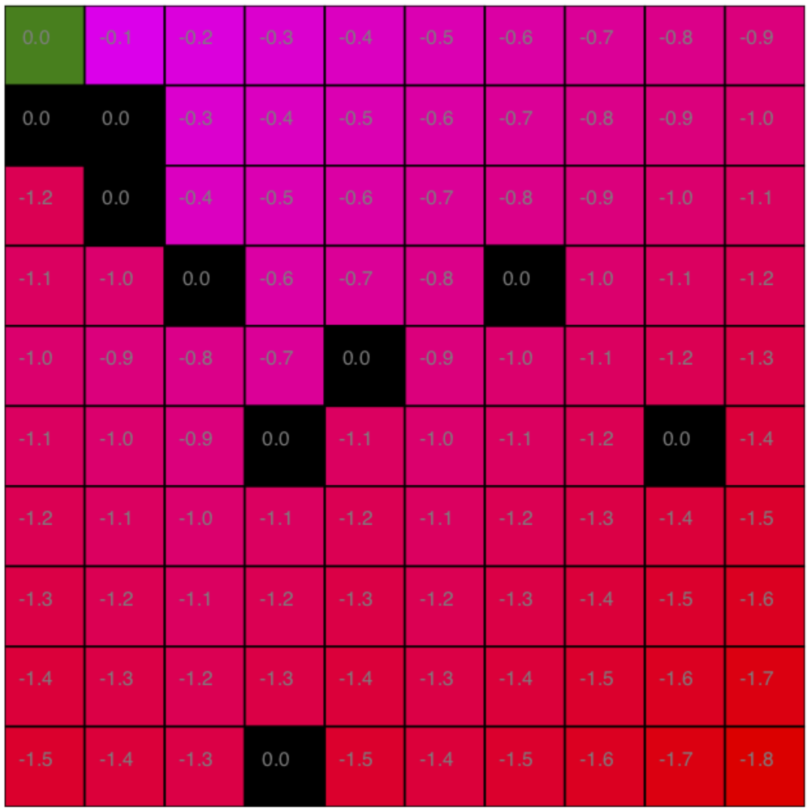
References

[1] Reinforcement Learning: A Survey, Authors: Leslie Pack Kaelbling, Michael L. Littman and Andrew W. Moore

[2] https://github.com/dennybritz/reinforcement-learning/blob/master/DP/Policy

Appendices

A Pictures



0.0	-0.1	-0.2	-0.3	-0.4	-0.5	-0.6	-0.7	-0.8	-0.9
0.0	0.0	-0.3	-0.4	-0.5	-0.6	-0.7	-0.8	-0.9	-1.0
-1.2	0.0	-0.4	-0.5	-0.6	-0.7	-0.8	-0.9	-1.0	-1.1
-1.1	-1.0	0.0	-0.6	-0.7	-0.8	0.0	-1.0	-1.1	-1.2
-1.0	-0.9	-0.8	-0.7	0.0	-0.9	-1.0	-1.1	-1.2	-1.3
-1.1	-1.0	-0.9	0.0	-1.1	-1.0	-1.1	-1.2	0.0	-1.4
-1.2	-1.1	-1.0	-1.1	-1.2	-1.1	-1.2	-1.3	-1.4	-1.5
-1.3	-1.2	-1.1	-1.2	-1.3	-1.2	-1.3	-1.4	-1.5	-1.6
-1.4	-1.3	-1.2	-1.3	-1.4	-1.3	-1.4	-1.5	-1.6	-1.7
-1.5	-1.4	-1.3	0.0	-1.5	-1.4	-1.5	-1.6	-1.7	-1.8

Figure 6: Test 1

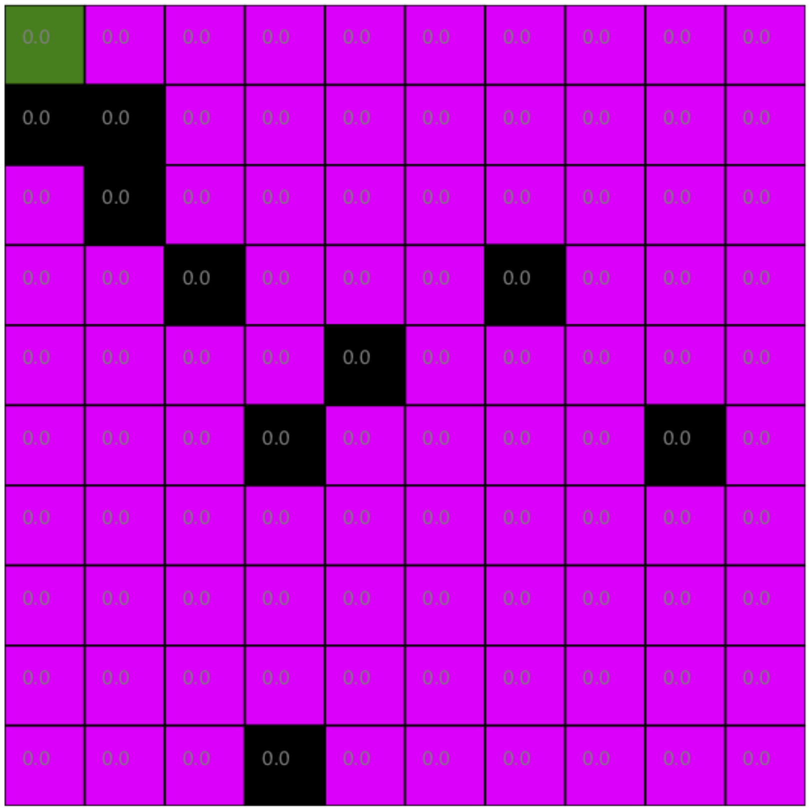
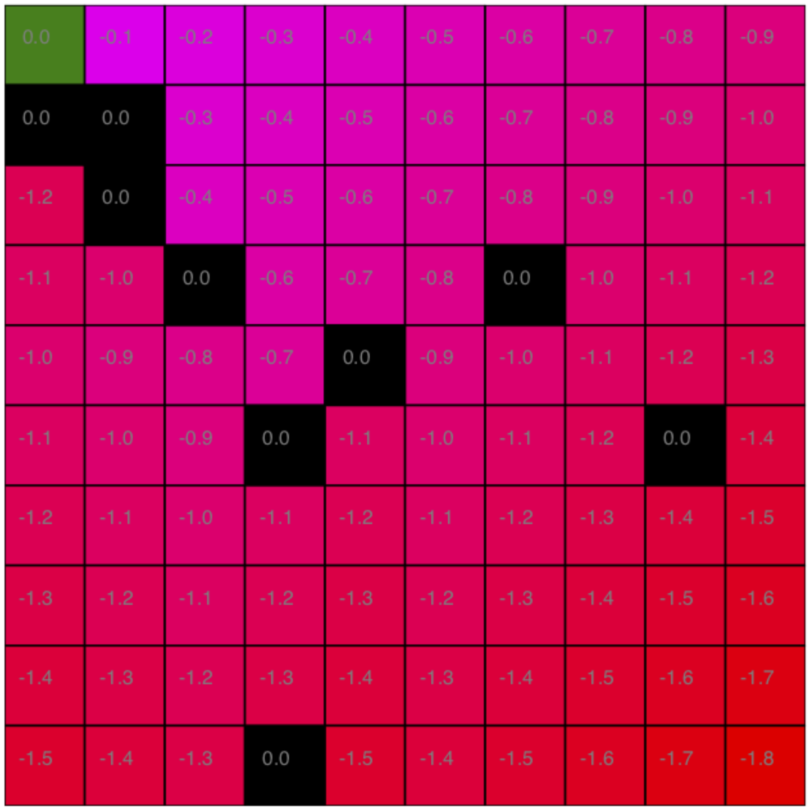
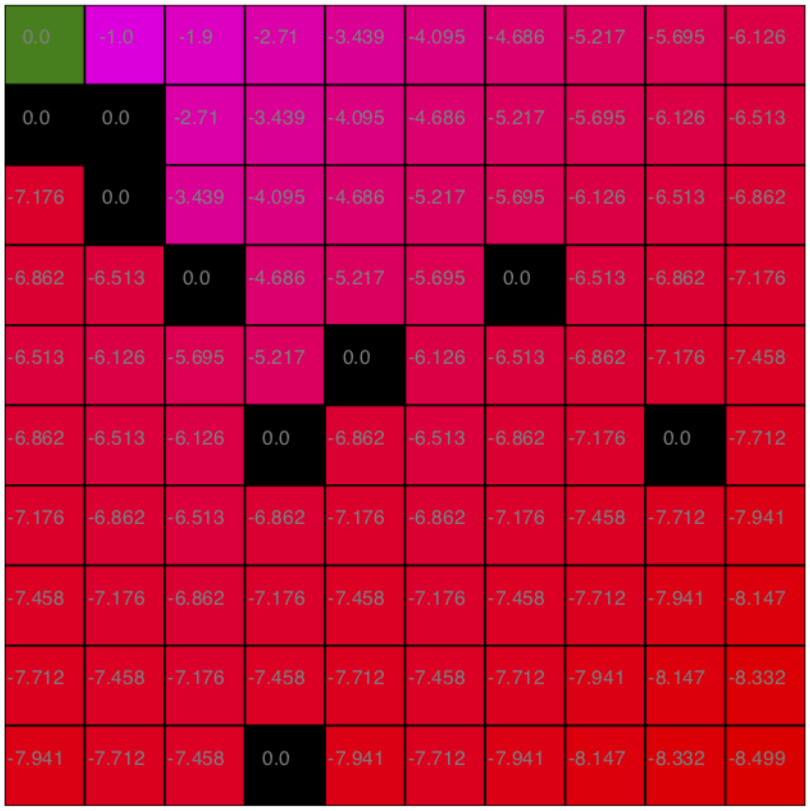


Figure 7: Test 2



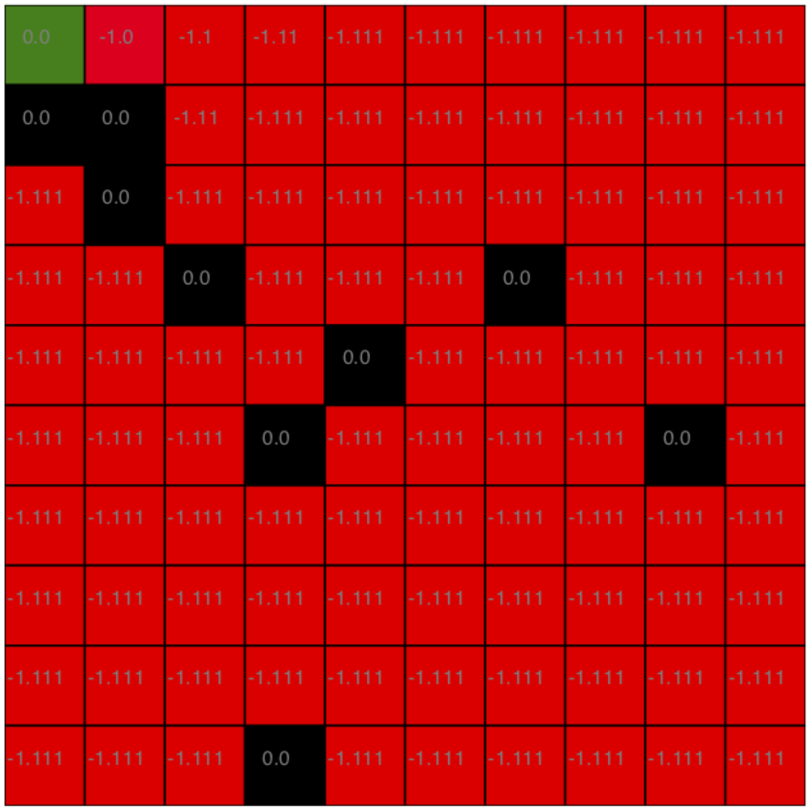
0.0	-0.1	-0.2	-0.3	-0.4	-0.5	-0.6	-0.7	-0.8	-0.9
0.0	0.0	-0.3	-0.4	-0.5	-0.6	-0.7	-0.8	-0.9	-1.0
-1.2	0.0	-0.4	-0.5	-0.6	-0.7	-0.8	-0.9	-1.0	-1.1
-1.1	-1.0	0.0	-0.6	-0.7	-0.8	0.0	-1.0	-1.1	-1.2
-1.0	-0.9	-0.8	-0.7	0.0	-0.9	-1.0	-1.1	-1.2	-1.3
-1.1	-1.0	-0.9	0.0	-1.1	-1.0	-1.1	-1.2	0.0	-1.4
-1.2	-1.1	-1.0	-1.1	-1.2	-1.1	-1.2	-1.3	-1.4	-1.5
-1.3	-1.2	-1.1	-1.2	-1.3	-1.2	-1.3	-1.4	-1.5	-1.6
-1.4	-1.3	-1.2	-1.3	-1.4	-1.3	-1.4	-1.5	-1.6	-1.7
-1.5	-1.4	-1.3	0.0	-1.5	-1.4	-1.5	-1.6	-1.7	-1.8

Figure 8: Test 3



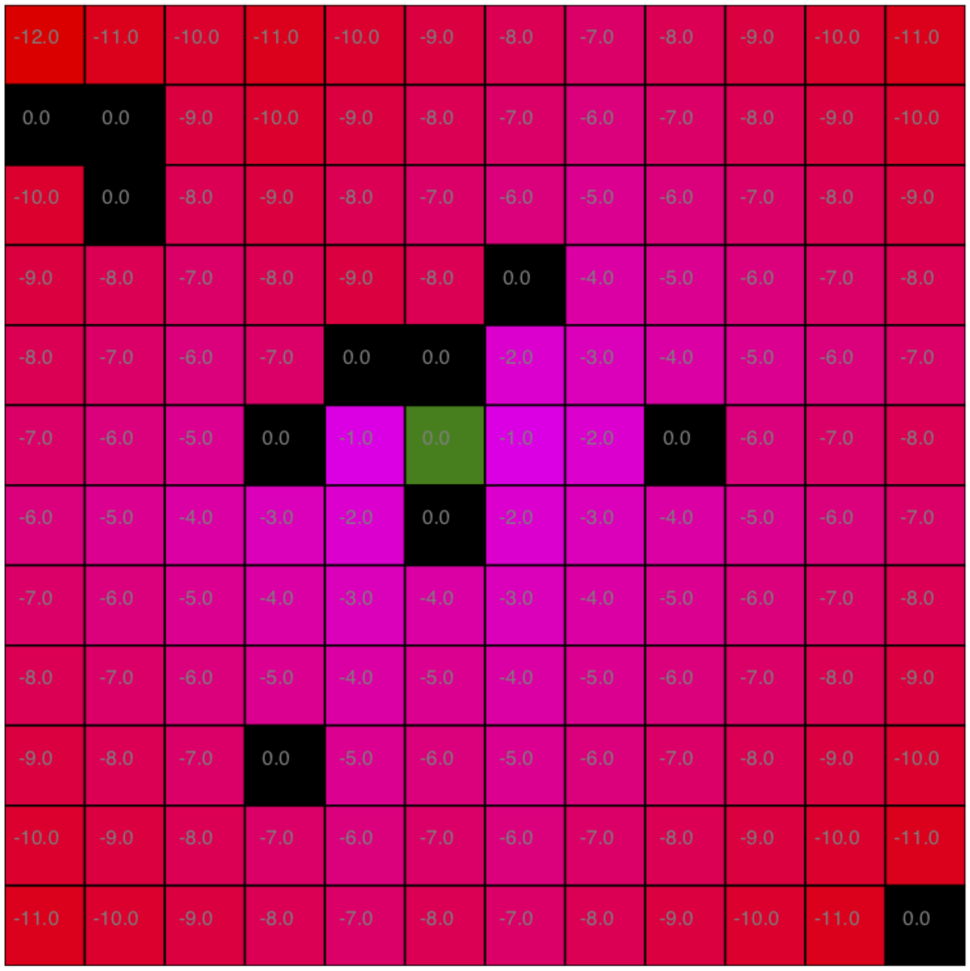
0.0	-1.0	-1.9	-2.71	-3.439	-4.095	-4.686	-5.217	-5.695	-6.126
0.0	0.0	-2.71	-3.439	-4.095	-4.686	-5.217	-5.695	-6.126	-6.513
-7.176	0.0	-3.439	-4.095	-4.686	-5.217	-5.695	-6.126	-6.513	-6.862
-6.862	-6.513	0.0	-4.686	-5.217	-5.695	0.0	-6.513	-6.862	-7.176
-6.513	-6.126	-5.695	-5.217	0.0	-6.126	-6.513	-6.862	-7.176	-7.458
-6.862	-6.513	-6.126	0.0	-6.862	-6.513	-6.862	-7.176	0.0	-7.712
-7.176	-6.862	-6.513	-6.862	-7.176	-6.862	-7.176	-7.458	-7.712	-7.941
-7.458	-7.176	-6.862	-7.176	-7.458	-7.176	-7.458	-7.712	-7.941	-8.147
-7.712	-7.458	-7.176	-7.458	-7.712	-7.458	-7.712	-7.941	-8.147	-8.332
-7.941	-7.712	-7.458	0.0	-7.941	-7.712	-7.941	-8.147	-8.332	-8.499

Figure 9: Test 4



0.0	-1.0	-1.1	-1.11	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111
0.0	0.0	-1.11	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111
-1.111	0.0	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111
-1.111	-1.111	0.0	-1.111	-1.111	-1.111	0.0	-1.111	-1.111	-1.111
-1.111	-1.111	-1.111	-1.111	0.0	-1.111	-1.111	-1.111	-1.111	-1.111
-1.111	-1.111	-1.111	0.0	-1.111	-1.111	-1.111	-1.111	0.0	-1.111
-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111
-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111
-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111
-1.111	-1.111	-1.111	0.0	-1.111	-1.111	-1.111	-1.111	-1.111	-1.111

Figure 10: Test 5



-12.0	-11.0	-10.0	-11.0	-10.0	-9.0	-8.0	-7.0	-8.0	-9.0	-10.0	-11.0
0.0	0.0	-9.0	-10.0	-9.0	-8.0	-7.0	-6.0	-7.0	-8.0	-9.0	-10.0
-10.0	0.0	-8.0	-9.0	-8.0	-7.0	-6.0	-5.0	-6.0	-7.0	-8.0	-9.0
-9.0	-8.0	-7.0	-8.0	-9.0	-8.0	0.0	-4.0	-5.0	-6.0	-7.0	-8.0
-8.0	-7.0	-6.0	-7.0	0.0	0.0	-2.0	-3.0	-4.0	-5.0	-6.0	-7.0
-7.0	-6.0	-5.0	0.0	-1.0	0.0	-1.0	-2.0	0.0	-6.0	-7.0	-8.0
-6.0	-5.0	-4.0	-3.0	-2.0	0.0	-2.0	-3.0	-4.0	-5.0	-6.0	-7.0
-7.0	-6.0	-5.0	-4.0	-3.0	-4.0	-3.0	-4.0	-5.0	-6.0	-7.0	-8.0
-8.0	-7.0	-6.0	-5.0	-4.0	-5.0	-4.0	-5.0	-6.0	-7.0	-8.0	-9.0
-9.0	-8.0	-7.0	0.0	-5.0	-6.0	-5.0	-6.0	-7.0	-8.0	-9.0	-10.0
-10.0	-9.0	-8.0	-7.0	-6.0	-7.0	-6.0	-7.0	-8.0	-9.0	-10.0	-11.0
-11.0	-10.0	-9.0	-8.0	-7.0	-8.0	-7.0	-8.0	-9.0	-10.0	-11.0	0.0

Figure 11: Test 6

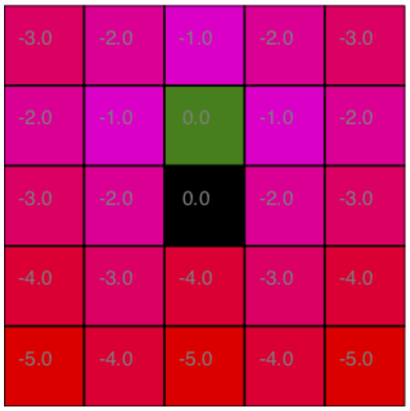


Figure 12: Test 7

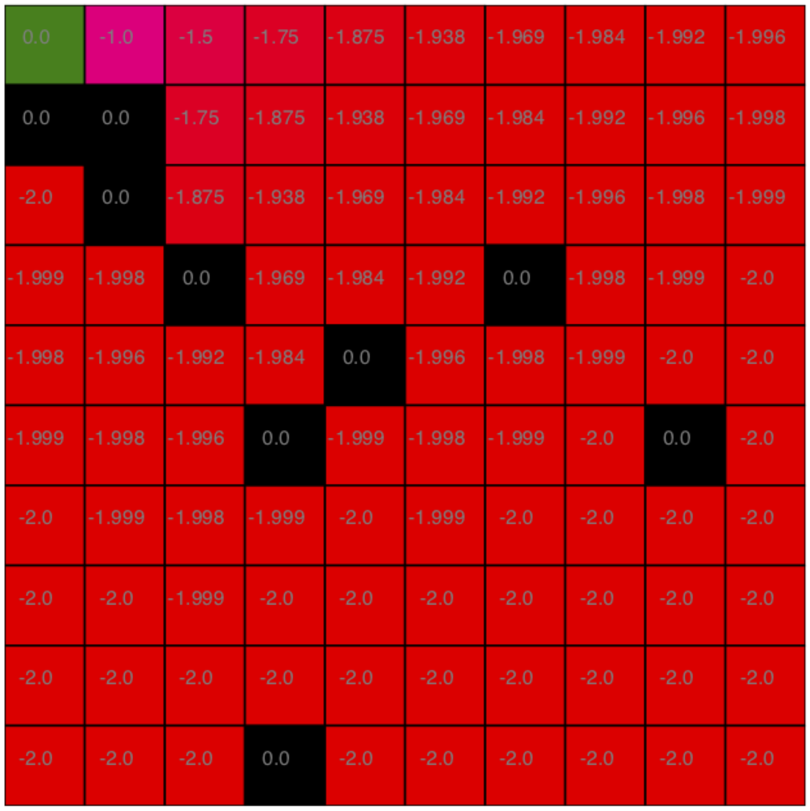


Figure 13: Test 8

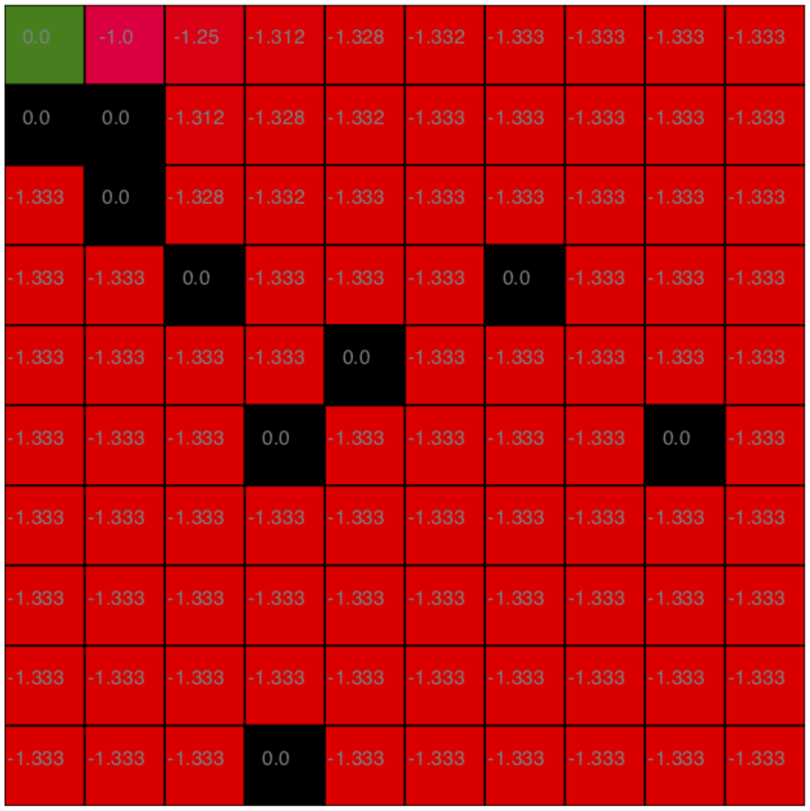


Figure 14: Test 9

B Code

B.1 Class for rendering the environment

```
from graphics import *
import time
from scipy.interpolate import interp1d
from PIL import Image as NewImage

class GridDisp:

    def __init__(self, shape, title = "std"):
        self.rects = {}
        self.texts = {}
        self.rs = 40
        self.win = GraphWin(title, shape[0]*self.rs,shape[1]*self.rs)
        self.buildGrid(shape)
        self.title = title
        print("Display Setup")

    def buildGrid(self, shape):
        for x in range(shape[0]):
            for y in range(shape[1]):
                self.rects[(x,y)] = Rectangle(Point(x*self.rs, y*self.rs),
                    ↪ Point(x*self.rs + self.rs, y*self.rs + self.rs))
                R = self.rects[(x,y)]
                R.draw(self.win)
                self.texts[(x,y)] = Text(Point(x*self.rs + 0.4*self.rs, y*
                    ↪ self.rs + 0.4 * self.rs), 'NA')
                self.texts[(x,y)].draw(self.win)
                self.texts[(x,y)].setSize(int(self.rs/4))
                self.texts[(x,y)].setTextColor('grey')

    def updateGrid(self, shape, obstacles=[], goals=[], agent=(-1,-1),
        ↪ values=[]):
        Vmin = float('inf')
        Vmax = float('-inf')
        for v in values:
            val = values[v]
            if val<Vmin:
                Vmin = val
            if val>Vmax:
                Vmax = val
        m = interp1d([Vmin,Vmax],[0,254])
        for x in range(shape[0]):
            for y in range(shape[1]):
                R = self.rects[(x,y)]
                if (x,y) in values:
                    R.setFill(color_rgb(255, 0, int(m(values[(x,y)]))))
                else:
                    R.setFill("white")
                for o in obstacles:
                    if x == o[0] and y == o[1]:
                        R.setFill("black")
                for g in goals:
                    if x == g[0] and y == g[1]:
                        R.setFill("green")
                if x == agent[0] and y == agent[1]:
                        R.setFill("blue")
                if values != []:
                    self.texts[(x,y)].setText(str(round(values[x,y],3)))
    def destroy(self):
        self.win.postscript(file=self.title + ".eps", colormode='color')
        # Convert from eps format to gif format using PIL
        #img = NewImage.open("image.eps")
        #img.save(self.title + ".gif", "gif")
        self.win.close()
if __name__ == "__main__":
    G = GridDisp([10,10], [(2,2),(2,3),(3,3)], [(0,0),(9,9)], (5,5), [])
    time.sleep(4)
    G.updateGrid([10,10], [(2,2),(2,3),(3,3)], [(0,0),(9,9)], (5,6), [])
    time.sleep(10)
    G.destroy()
```

B.2 Class for the Gridworld Logic (Denny Britz extension)

```
import numpy as np
import sys
from gym.envs.toy_text import discrete
from graphicsLib import GridDisp
UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3

class GridworldEnv(discrete.DiscreteEnv):
    """
    Grid World environment from Sutton's Reinforcement Learning book chapter
        ↪ 4.
    You are an agent on an MxN grid and your goal is to reach the terminal
    state at the top left or the bottom right corner.
    For example, a 4x4 grid looks as follows:
    T  o  o  o
    o  x  o  o
    o  o  o  o
    o  o  o  T
    x is your position and T are the two terminal states.
    You can take actions in each direction (UP=0, RIGHT=1, DOWN=2, LEFT=3).
    Actions going off the edge leave you in your current state.
    You receive a reward of -1 at each step until you reach a terminal state
        ↪ .
    """

    metadata = {'render.modes': ['human', 'ansi']}

    def __init__(self, shape=[4,4], obstacles = [], goals=[(0,0)], statePen
        ↪ =-0.1, transProb=1,  title = "std"):
        if not isinstance(shape, (list, tuple)) or not len(shape) == 2:
            raise ValueError('shape argument must be a list/tuple of length
                ↪ 2')

        self.shape = shape
        self.obstacles = obstacles
        self.goals = goals

        nS = np.prod(shape)
        nA = 4

        MAX_Y = shape[0]
        MAX_X = shape[1]
        P = {}
        grid = np.arange(nS).reshape(shape)
        it = np.nditer(grid, flags=['multi_index'])
        self.states = []
        while not it.finished:
            s = it.iterindex
            y, x = it.multi_index

            P[s] = {a : [] for a in range(nA)}
            is_done = lambda t: (t%self.shape[0], t//self.shape[0]) in self.
                ↪ goals
            reward = 0.0 if is_done(s) else statePen

            # We're stuck in a terminal state
            if is_done(s):
                P[s][UP] = [(transProb, s, reward, True)]
                P[s][RIGHT] = [(transProb, s, reward, True)]
                P[s][DOWN] = [(transProb, s, reward, True)]
                P[s][LEFT] = [(transProb, s, reward, True)]
            # Not a terminal state
            else:
                ns_up = s if (y == 0 or (x,y-1) in obstacles) else s - MAX_X
                ns_right = s if (x == (MAX_X - 1) or (x+1,y) in obstacles)
                    ↪ else s + 1
                ns_down = s if (y == (MAX_Y - 1) or (x,y+1) in obstacles)
                    ↪ else s + MAX_X
                ns_left = s if (x == 0 or (x-1,y) in obstacles) else s - 1
                P[s][UP] = [(transProb, ns_up, reward, is_done(ns_up))]
                P[s][RIGHT] = [(transProb, ns_right, reward, is_done(
                    ↪ ns_right))]
                P[s][DOWN] = [(transProb, ns_down, reward, is_done(ns_down))
                    ↪ ]
                P[s][LEFT] = [(transProb, ns_left, reward, is_done(ns_left))
                    ↪ ]
            if (x,y) not in obstacles:
                self.states.append(s)
            it.iternext()

        # Initial state distribution is uniform
        isd = np.zeros(nS)
        isd[nS-1]=1

        # We expose the model of the environment for educational purposes
        # This should not be used in any model-free learning algorithm
        self.P = P
        self.GD = GridDisp(shape,title)
        super(GridworldEnv, self).__init__(nS, nA, P, isd)

    def _render(self, mode='human', close=False, values = [], valIt = False)
        ↪ :
        if close:
            return

        outfile = StringIO() if mode == 'ansi' else sys.stdout

        grid = np.arange(self.nS).reshape(self.shape)
        it = np.nditer(grid, flags=['multi_index'])
        vals = {}
        while not it.finished:
            s = it.iterindex
            y, x = it.multi_index
            if self.s == s:
                output = "␣x␣"
            elif s == 0 or s == self.nS - 1:
                output = "␣T␣"
            else:
                output = " o "

            if x == 0:
                output = output.lstrip()
            if x == self.shape[1] - 1:
                output = output.rstrip()

            #outfile.write(output)

            # if x == self.shape[1] - 1:
            #     outfile.write("\n")
            #print(values)
            if values!=[]:
                vals[(x,y)] = values[y,x]
            it.iternext()
        if(valIt):
            self.GD.updateGrid(self.shape, self.obstacles, self.goals,
                ↪ (-1,-1), vals)
        else:
            self.GD.updateGrid(self.shape, self.obstacles, self.goals, (self
                ↪ .s%self.shape[0], self.s//self.shape[0]), vals)

    def stepQ(self, action):
        a = action
        #print(a)
        #You can take actions in each direction (UP=0, RIGHT=1, DOWN=2, LEFT
            ↪ =3).
        hindered = False
        obstacles = self.obstacles
        x,y = self.s%self.shape[0], self.s//self.shape[0]
        if (a==0 and (x,y-1) in obstacles):
            hindered = True
        if (a==1 and (x+1,y) in obstacles):
            hindered = True
        if (a==2 and  (x,y+1) in obstacles):
            hindered = True
        if (a==3 and (x-1,y) in obstacles):
            hindered = True
        if not hindered:
           return self.step(action)
        else:
            return (self.s, -10.0, False, "bumdep")

    def close(self):
        self.GD.destroy()
```