

DOCUMENTATION

PRELIMINARIES:

POLICY:

We define a policy π in terms of the Markov Decision Process to which it refers. A Markov Decision Process is a tuple of the form (S, A, P, R) , structured as follows.

- The first element is a set S containing the internal states of the agent.
- The second element is a set A containing the actions of the agent.
- An action can also lead to a modification of the state of the agent. This is represented by the matrix P containing the probability of transition from one state to another.
- The fourth element $R(s)$ comprises the reward function for the agent.

Policy can be of 2 types:

1. Deterministic Policy: Here the policy function maps each and every state to a particular action, which means that at each state the action is determined.
2. Stochastic Policy: Here the policy function maps each and every state to a probability distribution over all possible actions. This is a more general case.

VALUE FUNCTION:

The Value Function represents the value for the agent to be in a certain state. More specifically, the state value function describes the expected return G_t from a given state, where G_t is the future reward when we follow a particular policy. This is why the value of a certain state also depends on the policy.

The expression of value at a state 's' is

$$v_{\pi}(s) = \mathbb{E}_{\pi}[G_t | S_t = s]$$

From the value function we can get an idea that how good a state is.

BELLMAN EQUATION:

A fundamental property of the value functions is that they satisfy some recursive relations known as Bellman equation. According to it, long-term- reward in a given action is equal to the reward from the current action combined with the expected reward from the future actions taken at the following time.

The mathematical form of bellman equation is

$$V(s) = \max_a (R(s,a) + \gamma V(s'))$$

Where, V represents the value function, s represents the current state, s' represents the next state reached by taking action a , R is the reward function and γ is the discount factor.

VALUE ITERATION ALGORITHM:

To find the optimal policy, a widely used algorithm is value-iteration algorithm. First we assign a zero value to the goal and then by using the BELLMAN EQUATION we assign values to every other state. Also we create a policy table where we store the action corresponding to which our maximum value is occurring. After completion, the value table gives us the maximum reward that we can get from that state to goal and the policy table gives us the optimal policy. The benefit of this algorithm is that once the table is created we can find optimal policy for any initial state.

We will use this algorithm to solve our task with slight modification in the BELLMAN EQUATION.

In BELLMAN EQUATION when an action from state s leads to a particular state s_1 then the value due to that action is (let $\gamma=1$)

$$R(s,a)+V(s_1)$$

But when a particular action a can lead to multiple states, say s_1 and s_2 with probability p_1 and $(1-p_1)$ then the value due to that action will be

$$R(s,a)+p_1*V(s_1)+p_2*V(s_2)$$

EXPLANATION OF MY CODE:

1. Basic Formats:

First we will import 'gridworld' module and take the 'GridWorld' class of that module as a new 'world' object. So now inside this world object we have all the information about the environment (example: height, width, position of obstacles, wind regions and wind power, position of goal and initial position). This is done because if we completely change the gridworld.py file (to describe some different environment then also my code will give correct result).

Then a 2D array named 'value_tables' is created which will finally store the value of each state according to BELLMAN EQUATION. Initially all the elements of this table are initialized with some minus infinity(which is nothing but a large negative number) for the simplicity in calculation, as any valid value(a value that is relevant) will be larger than the minus infinity and replace that. Only the goal position of the value_tables is made zero, so that the modification in the value table can start from the neighbouring states of goal (back tracking method).

The table 'optimal_action' will store the best action corresponding to every states.

```
import gridworld
import numpy as np

world=gridworld.GridWorld()
minus_inf=-99
value_tables=np.full((world.WORLD_HEIGHT,world.WORLD_WIDTH),minus_inf,dtype=float)
optimal_action=np.full((world.WORLD_HEIGHT,world.WORLD_WIDTH),-1)
value_tables[world.GOAL[0]][world.GOAL[1]]=0
```

✓ 0.8s

2. The '*assign_value(current_state)*' function:

This function will take a state 's' as argument and will return 2 parameters:

- The value of the state according to the equation
$$V(s)=\max_a (R(s,a)+ \sum (p_{s'})*V(s'))$$
- Where $p_{s'}$ is the probability of reaching the state s' from s when action 'a' is taken.

The action 'a' corresponding to which the above max value is occurring.

```
def assign_value(current_state):
    action_values=np.array([0,0,0,0])
    for action in world.ACTIONS:
        new_state,reward=world.step(current_state,action)
        if(world.WIND[new_state[0]]==0):
            action_values[action]=reward+value_tables[new_state[0]][new_state[1]]
        else:
            if(action==0 or action==1):
                if(new_state[1]==current_state[1]):
                    state2=value_tables[new_state[0]][new_state[1]]
                    state8=value_tables[new_state[0]][max(0,new_state[1]-world.WIND[new_state[0]])]
                else:
                    state8=value_tables[new_state[0]][new_state[1]]
                    state2=value_tables[new_state[0]][current_state[1]]
            elif(action==3):
                if(current_state[1]==world.WORLD_WIDTH-1):
                    state2=value_tables[current_state[0]][current_state[1]]
                    state8=value_tables[current_state[0]][max(0,current_state[1]-world.WIND[current_state[0]])]
                else:
                    state2=value_tables[current_state[0]][current_state[1]+1]
                    state8=value_tables[current_state[0]][max(0,1+current_state[1]-world.WIND[current_state[0]])]
            elif(action==2):
                if(current_state[1]==0):
                    state2=value_tables[current_state[0]][current_state[1]]
                    state8=value_tables[current_state[0]][max(0,current_state[1]-world.WIND[current_state[0]])]
                else:
                    state2=value_tables[current_state[0]][current_state[1]-1]
                    state8=value_tables[current_state[0]][max(0,-1+current_state[1]-world.WIND[current_state[0]])]
            action_values[action]=reward+(1-world.WIND_PROB)*state2+(world.WIND_PROB)*state8
    max_val, max_action=action_values[0],0
    for i in range(4):
        if(action_values[i]>max_val): max_val,max_action=action_values[i],i
    return max_val,max_action
```

To calculate the value of

$$V(s) = \max_a (R(s,a) + \sum (p_{s'}) * V(s'))$$

we will have to calculate the expression

$$(R(s,a) + \sum (p_{s'}) * V(s'))$$

for all possible actions that is achieved in the program by running a loop on 'world.ACTIONS'.

Now inside the loop (when we are dealing with a particular action) it is also a challenging task to calculate $\sum (p_{s'}) * V(s')$ as multiple final states will be possible in the region of wind. That problem is solved by taking two variables 'state2' and 'state8' where state2 is the value of the final state that has 20% probability to be reached and state8 is the value of the final state that has 80% probability to be reached.

Therefore,

$$\sum (p_{s'}) * V(s') = 0.2 * \text{state2} + 0.8 * \text{state8}.$$

In this way the value corresponding to all the actions have been found and stored in an array named 'action_values'. From that array the maximum value and the action corresponding to that maximum value have been found and returned.

3. Modification of the value table:

We will run a nested loop to take all possible states except the goal where the agent can be present. As the agent can not be present in the position of a obstacle so we will also have to take care of that.

Then we will call the **'assign_value(current_state)'** function with the state as argument. Then we will compare the max_value returned by the function with the present state value. If the max_value is greater then we will substitute that value in the 'value_table' and the corresponding max_action in the 'optimal_action' table.

Next we will have to repeat this step as many number of times possible to get more accurate results. The more we iterate, the more the values will converge and a time will come when they will become constant (none of the values of the table will change with further iteration). At that point our value table will be accurate. Here I have used 10000 iterations which appeared to be sufficient.

```
for k in range(10000):
    for i in range(world.WORLD_HEIGHT):
        for j in range(world.WORLD_WIDTH):
            if [i,j] not in (world.obstacles+[world.GOAL]):
                if (value_tables[i][j]<(assign_value([i,j])[0])):
                    value_tables[i][j],optimal_action[i][j]=assign_value([i,j])[0],assign_value([i,j])[1]
```

✓ 26.2s

4. Printing the value table:

Now, after the 10000 iterations our value table is completely modified. So we will observe it by printing.

```
print(value_tables)
```

[44] ✓ 0.6s

```
... [[-20. -19. -18. -17. -16. -15. -14. -13. -12. -11. -10. -9. -8. -7.
      -7.]
      [-19. -18. -17. -16. -15. -14. -13. -12. -11. -10. -9. -8. -7. -6.
      -6.]
      [-20. -19. -18. -99. -14. -13. -12. -11. -10. -9. -8. -7. -6. -5.
      -5.]
      [-21. -20. -19. -99. -13. -12. -11. -10. -9. -8. -7. -6. -5. -4.
      -4.]
      [-22. -21. -20. -19. -14. -13. -12. -11. -10. -9. -8. -7. -3. -3.
      -3.]
      [-23. -22. -21. -20. -19. -14. -13. -12. -11. -8. -3. -2. -2. -3.
      -3.]
      [-24. -23. -23. -22. -21. -19. -18. -13. -8. -3. -2. -1. -2. -2.
      -3.]
      [-25. -24. -24. -99. -21. -20. -19. -18. -99. -2. -1. 0. -1. -2.
      -3.]
      [-26. -25. -25. -99. -22. -21. -20. -19. -99. -1. 0. 0. 0. -1.
      -2.]
      [-27. -26. -26. -99. -23. -22. -21. -20. -99. -2. -1. 0. -1. -2.
      -3.]]
```

This table gives us an idea that what is the maximum reward possible to reach the goal from any state. The states that have -99 value represent the obstacles.

5. Printing the action table:

But always we are most interested about the optimal policy, that will tell us that what should be the best action at any state that will help us to reach the goal with maximum reward. We have stored that data in the 'optimal_action' table. Now we can print that to observe it.

```
print(optimal_action)
```

✓ 0.4s

[1	1	1	1	1	1	1	1	1	1	1	1	1	1]
[3	3	3	3	1	1	1	1	1	1	1	1	1	1]
[0	0	0	-1	1	1	1	1	1	1	1	1	1	1]
[0	0	0	-1	3	3	3	3	3	3	3	3	1	1]
[0	0	0	3	0	0	0	0	0	0	0	0	1	1]
[0	0	0	0	0	0	0	0	0	1	1	1	1	1]
[0	0	0	0	0	0	0	3	3	1	1	1	1	2]
[0	0	0	-1	3	3	3	0	-1	1	1	1	2	2]
[0	0	0	-1	0	0	0	0	-1	3	3	-1	2	2]
[0	0	0	-1	0	0	0	0	-1	0	0	0	0	0]

Here the actions are denoted by an integer which is same as the convention used in gridworld.py that is 0 means up, 1 means down, 2 means left, 3 means right. The state of obstacles and the goal are marked -1.

This is the general action table that will lead us to the goal most optimally for any start position.

The goal and start positions given in our task are circled with blue for easier understanding.

6. Printing the optimal path:

Now finally we will use the `action_table` to find the optimal path to reach the goal from the given start position.

```
current_state=world.START
total_reward=0
cnt=1
print("Initial Co-ordinate : ",current_state)
while(current_state!=world.GOAL):
    new_state,reward=world.step(current_state,optimal_action[current_state[0]][current_state[1]])
    print("Co-ordinate at step ",cnt," : ",new_state)
    total_reward+=reward
    cnt+=1
    current_state=new_state
print("GOAL REACHED in most optimal way.")
print("Maximum possible reward : ",total_reward)
```

✓ 0.4s Python

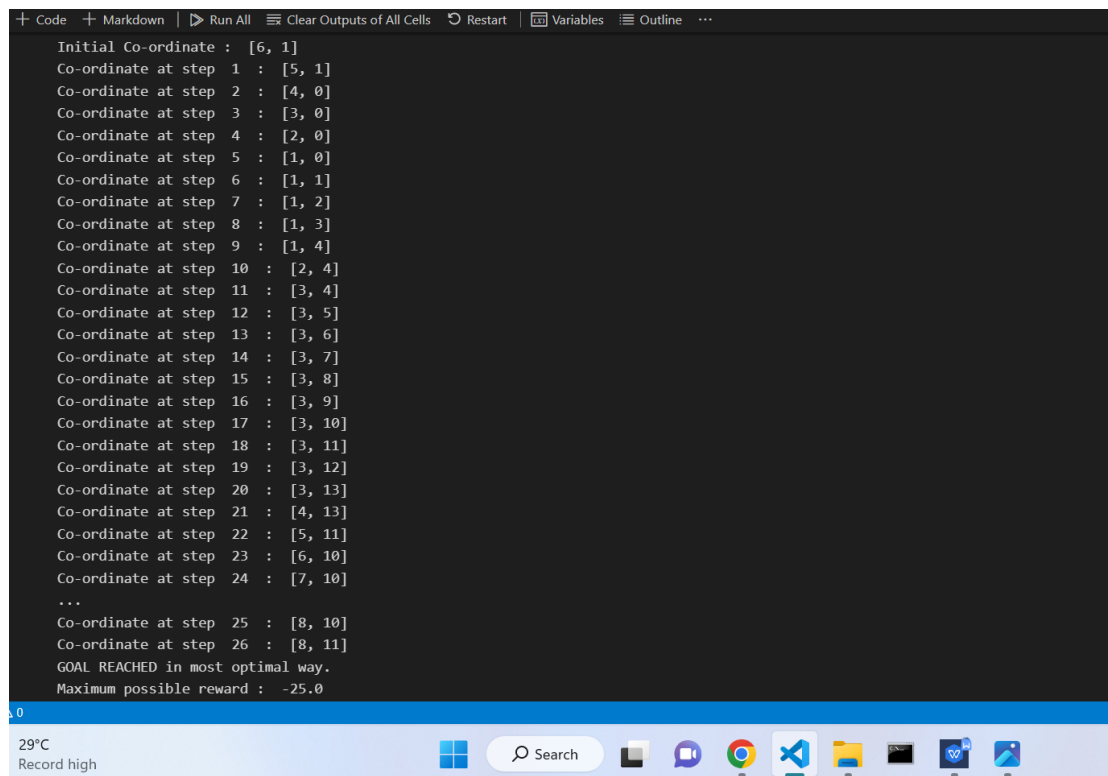
Initial Co-ordinate : [6, 1]
Co-ordinate at step 1 : [5, 0]
Co-ordinate at step 2 : [4, 0]
Co-ordinate at step 3 : [3, 0]
Co-ordinate at step 4 : [2, 0]

```
+ Code + Markdown | ▶ Run All | ✕ Clear Outputs of All Cells | ⌂ Restart | 📄 Variables | 📖 Outline | ⋮
[54] | ✓ 0.4s

... Initial Co-ordinate : [6, 1]
Co-ordinate at step 1 : [5, 0]
Co-ordinate at step 2 : [4, 0]
Co-ordinate at step 3 : [3, 0]
Co-ordinate at step 4 : [2, 0]
Co-ordinate at step 5 : [1, 0]
Co-ordinate at step 6 : [1, 1]
Co-ordinate at step 7 : [1, 2]
Co-ordinate at step 8 : [1, 3]
Co-ordinate at step 9 : [1, 4]
Co-ordinate at step 10 : [2, 4]
Co-ordinate at step 11 : [3, 4]
Co-ordinate at step 12 : [3, 5]
Co-ordinate at step 13 : [3, 6]
Co-ordinate at step 14 : [3, 7]
Co-ordinate at step 15 : [3, 8]
Co-ordinate at step 16 : [3, 9]
Co-ordinate at step 17 : [3, 10]
Co-ordinate at step 18 : [3, 11]
Co-ordinate at step 19 : [3, 12]
Co-ordinate at step 20 : [3, 13]
Co-ordinate at step 21 : [4, 12]
Co-ordinate at step 22 : [5, 12]
Co-ordinate at step 23 : [6, 11]
Co-ordinate at step 24 : [7, 11]
Co-ordinate at step 25 : [8, 11]
GOAL REACHED in most optimal way.
Maximum possible reward : -24.0
```

This path may change slightly from trial to trial as this is not a deterministic case due to the flow of wind which affects the position of the particle 80% of times. Our work is just to take actions according to the action table. To

explain this I am running the above block of codes again and showing the path.



```
Initial Co-ordinate : [6, 1]
Co-ordinate at step 1 : [5, 1]
Co-ordinate at step 2 : [4, 0]
Co-ordinate at step 3 : [3, 0]
Co-ordinate at step 4 : [2, 0]
Co-ordinate at step 5 : [1, 0]
Co-ordinate at step 6 : [1, 1]
Co-ordinate at step 7 : [1, 2]
Co-ordinate at step 8 : [1, 3]
Co-ordinate at step 9 : [1, 4]
Co-ordinate at step 10 : [2, 4]
Co-ordinate at step 11 : [3, 4]
Co-ordinate at step 12 : [3, 5]
Co-ordinate at step 13 : [3, 6]
Co-ordinate at step 14 : [3, 7]
Co-ordinate at step 15 : [3, 8]
Co-ordinate at step 16 : [3, 9]
Co-ordinate at step 17 : [3, 10]
Co-ordinate at step 18 : [3, 11]
Co-ordinate at step 19 : [3, 12]
Co-ordinate at step 20 : [3, 13]
Co-ordinate at step 21 : [4, 13]
Co-ordinate at step 22 : [5, 11]
Co-ordinate at step 23 : [6, 10]
Co-ordinate at step 24 : [7, 10]
...
Co-ordinate at step 25 : [8, 10]
Co-ordinate at step 26 : [8, 11]
GOAL REACHED in most optimal way.
Maximum possible reward : -25.0
```

The difference in path is illustrated by a small example:

Say when the particle is in the start position which is [6,1]. At that point the particle will move take action 0 that is 'UP' according to the action table. But it is not deterministic that on taking this action whether it will go to [5,1] or [5,0] because that depends on the wind.

7. OpenCV to visualize the optimal path:

By the help of OpenCV tools, I have written a program that will take the states of the optimal path and mark those states by a red circle. I have read the image of the grid from the given 'grid.png' file and showed the path in the 'path.jpeg' file.

```
import cv2 as cv

img=cv.imread("grid.png")
img=cv.resize(img,(1600,1100))
current_state=world.START
y=(current_state[0]+1)*50+(current_state[0]+2)*50
x=(current_state[1]+1)*50+(current_state[1]+2)*50
cv.circle(img,(x,y),40,(0,0,255),-1)
while(current_state!=world.GOAL):
    new_state,reward=world.step(current_state,optimal_action[current_state[0]][current_state[1]])
    y=(new_state[0]+1)*50+(new_state[0]+2)*50
    x=(new_state[1]+1)*50+(new_state[1]+2)*50
    cv.circle(img,(x,y),40,(0,0,255),-1)
    current_state=new_state
cv.imwrite("path.jpeg",img)
```

[29] ✓ 0.7s

... True

