Dynamic Programming

Vansh Gupta
49003814

Written Report 2

For the MAB project, we were supposed to do the following:

In this assignment, you are required to draw a custom mine-grid world individually with

1. Different map size.
2. Determined initial & terminal state position.
3. Mines are mounted.
4. States differ within specific probability if action is chosen

[Rules for designing mine-grid world environment]

1. From **your student ID,** let us define the values of **x** and **y**.
   (e.g. ID: 2021**2324**, then x=23, y=24) **// For me: x=38, y=14**
2. The map of grid world follows the instructions below (% = **modular operation**):
   a. Map width: $w = 8 + x\%4$ **// For me: w=10** (e.g. x=23 → $w = 8 + (23\%4) = 8 + 3 = 11$)
      Map height: $h = 8 + y\%4$**// For me: h=10** (e.g. y=24 → $h = 8 + (24\%4) = 8 + 0 = 8$)
   b. Initial position of agent: $\left((x + y)\%3,\ (x - y)\%3\right)$**// For me: (1, 0)**
      (e.g. x=23, y=24 → $(2,\ 2)$)
      Termination grid position: $(w - 1 - (x - y)\%3,\ h - 1 - (x + y)\%3)$**// For me: (9, 8)**
      (e.g. x=23, y=24 → $(8,\ 5)$)
   c. 3 sinkhole mines will be installed at:
      i. $((x + y)^2\%w, (x - y)^2\%h)$ **// For me: (4, 6)** (e.g. x=23, y=24 → $(9,\ 1)$)
      ii. $(x^4\%w, y^4\%h)$**// For me: (6, 6)** (e.g. x=23, y=24 → $(1,\ 0)$)
      iii. $\left((2022 - y)\%w, (2022 - x)\%h\right)$**// For me: (8,4)** (e.g. x=23, y=24 → $(7,\ 7)$)
      iv. If mines are overlapped with initial or terminal position, those mines would be ignored and not be planted.

Similarly, if mines are overlapped themselves, only single mine would be planted on the spot.

v. If your agent steps on a mine, the mine explodes, and the agent jumps to initial position.
(This teleportation would not be counted as any additional action of your agent.)

vi. The mine still remains on the same spot, even after it explodes.

3. You have the actions as follows:

   a. If the agent moves left or right,

it moves 1 step to that direction with probability of $x/100$. **// For me: 0.38**

Otherwise, it moves 2 steps with probability of $(1 - x/100)$. **// For me: 0.62**

   b. If the agent moves up or down,

it moves 1 step to that direction with probability of $y/100$. **// For me: 0.14**

Otherwise, it moves 2 steps with probability of $(1 - y/100)$. **// For me: 0.86**

   c. [Wall handling instruction] When the agent moves to the outside of the world (i.e. the agent meets the wall), it stays still.
   However, when the agent is moving by **2 steps** and falls into the wall, but when it does not fall if the agent proceed to the direction by **1 step**, the agent moves 1 step regardless of x or y.

   d. [Mine handling instruction] When the agent passes through a mine but does not finish the action on the mine spot, mine is not exploded.

4. You have a reward as follows:

   a. For every action, the agent gets reward of -1 until it terminates.

   b. The mine explosion occurs no additional reward.

[What you need to do]

Complete your code in the "your code here comment" part of the template code.

1. Design your custom grid world following the rules above.

2. Complete the policy evaluation function in your skeleton code.
3. Complete the policy iteration function **using the policy evaluation function**.
4. Complete the value iteration code.
5. Answer **the questions** below, and include your answers in your report.
    a. Between policy iteration and value iteration, which algorithm converges faster in your implementation? why? (4 pts)
    b. What happen if discounting factor is zero on your grid world? Why does value iteration not converge? (3 pts)
    c. Design your own reward that makes value iteration converge to optimal policy even zero discounting factor is given. (8 pts)

I have already implemented (with comments) my code and talked about it in the video. So, I will only be addressing the questions in this report.

But before that, first we look at the rendered grid for our gridworld:

```
o o o o o o o o o o
X o o o o o o o o o
o o o o o o o o o o
o o o o o o o o o o
o o o o o o M o o o
o o o o o o o o o o
o o o o o o M o o o
o o o o o o o o o o
o o o o M o o o o o
o o o o o o o o T o
```

Next, we check out the output of policy iteration:

Reshaped Grid Policy (0=up, 1=right, 2=down, 3=left):

```
[[1 1 1 1 1 1 2 2 2 3]
 [1 1 1 1 2 1 1 2 2 3]
 [1 1 1 1 1 1 1 2 2 3]
 [1 1 1 1 1 1 1 2 2 3]
 [1 1 1 1 0 2 0 2 2 3]
 [1 1 1 1 1 1 1 2 2 3]
 [1 1 2 1 0 2 0 2 2 3]
 [1 1 1 1 1 1 1 2 2 3]
 [2 2 2 2 0 1 1 1 2 3]
 [0 0 0 0 0 0 0 1 0 3]]
```

Reshaped Grid Value Function:

```
[[-10.63982689 -10.0546381   -9.38558777  -8.85179798  -8.09984602
   -7.69976854  -6.73215158  -6.67992088  -5.15126072  -7.09903002]
 [-10.25862117  -9.69062482  -8.99384474  -8.50478035  -7.68069066
   -7.39696436  -6.24168421  -6.49213606  -4.47527824  -6.72573009]
 [ -9.53909294  -8.95110538  -8.28656919  -7.74549853  -7.00528992
   -6.58627155  -5.64920441  -5.5476998   -4.09851368  -5.99700907]
 [ -9.21296298  -8.64822101  -7.94619193  -7.46559368  -6.62784893
   -6.36614691  -5.17534371  -5.48309081  -3.3738213   -5.68156839]
 [-10.01614588  -9.21518507  -8.89415412  -7.79904276  -7.95244818
   -6.09211686 -11.25862117  -4.39542685  -3.05369616  -4.88556919]
 [ -8.17773016  -7.61738838  -6.90826222  -6.43911088  -5.58290336
   -5.35098     -4.11214672  -4.49736168  -2.263144    -4.64835896]
 [ -9.91558653  -8.96797263  -8.88347894  -7.40685586  -8.62071191
   -5.04997635 -11.25862117  -3.21604211  -2.0196      -3.76139411]
 [ -7.1565933   -6.60220095  -5.88347894  -5.42980476  -4.54863408
   -4.35697389  -3.0532      -3.54315789  -1.14        -3.62995789]
 [ -9.48217469  -8.92778234  -8.20906033  -7.75538616 -11.25862117
   -4.          -3.          -2.          -1.          -2.62      ]
 [ -8.4821747   -7.92778234  -7.20906033  -6.75538616  -6.48803227
   -5.30699755  -4.045752    -2.63157895   0.          -2.63157895]]
```

And here are the results for value iteration:

Reshaped Grid Policy (0=up, 1=right, 2=down, 3=left):

```
[[1 1 1 1 2 1 1 2 2 3]
 [1 1 1 1 1 1 1 2 2 3]
 [1 1 1 1 1 1 1 2 2 3]
 [1 1 1 1 1 1 1 2 2 3]
 [1 1 1 1 0 2 0 2 2 3]
 [1 1 1 1 1 1 1 2 2 3]
 [1 1 2 1 0 2 0 2 2 3]
 [1 1 1 1 1 1 1 2 2 3]
 [2 2 2 2 0 1 1 1 2 3]
 [0 0 0 0 0 0 0 1 0 3]]
```

Reshaped Grid Value Function:

```
[[-10.63970021 -10.05454229  -9.38551981  -8.85174383  -8.09981126
   -7.6997357   -6.73213694  -6.679897    -5.15126072  -7.09901521]
 [-10.25846381  -9.69050581  -8.99376033  -8.50471308  -7.68064748
   -7.39692356  -6.24166603  -6.4921064   -4.47527824  -6.7257117 ]
 [ -9.53902723  -8.95105568  -8.28653394  -7.74547044  -7.00527189
   -6.58625451  -5.64919681  -5.54768741  -4.09851368  -5.9970014 ]
 [ -9.21286023  -8.6481433   -7.94613681  -7.46554975  -6.62782073
   -6.36612027  -5.17533184  -5.48307144  -3.3738213   -5.68155639]
 [-10.01609018  -9.21514581  -8.89412248  -7.79902289  -7.95242872
   -6.09210891 -11.25846381  -4.39542107  -3.05369616  -4.8855656 ]
 [ -8.17766108  -7.61733614  -6.90822516  -6.43908134  -5.5828844
   -5.35096208  -4.11213874  -4.49734866  -2.263144    -4.64835089]
 [ -9.91548142  -8.96791511  -8.88340907  -7.40684116  -8.62069252
   -5.04997353 -11.25846381  -3.21604006  -2.0196      -3.76139284]
 [ -7.15654526  -6.60216462  -5.88345317  -5.42978423  -4.5486209
   -4.35696144  -3.05319445  -3.54314884  -1.14        -3.62995228]
 [ -9.48208863  -8.92771725  -8.20901416  -7.75534936 -11.25846381
   -4.          -3.          -2.          -1.          -2.62      ]
 [ -8.48212133  -7.92774199  -7.20903171  -6.75536335  -6.4879989
   -5.30698684  -4.04574723  -2.63157242   0.          -2.6315749 ]]
```

Now, we move on to answering the questions

Ans 1. The CPU time taken in executing policy iteration (i.e., for it to converge) is 0.4375 seconds while that for value iteration is 0.0625 seconds. That is, there is a speed up of 7 times when value iteration is used. To me this is quite surprising as policy iteration generally converges in fewer iterations than value iteration. However, if I were to guess, I would say that as we all know, Bellman equation is used to solve the value function of a given policy. Policy iteration solves the value function for current policy which would waste a lot of time in comparison with Value iteration which simply solves the value function for the optimal policy.

Ans 2. I noticed that for a discount factor of 0, neither of them converge (nor take any time). By limiting the penalty (anticipated discounted cost) that an agent can experience in either state, discounting leads to well-defined solutions. The discount factor chosen, however, does have an impact on the policy that emerges. If it is close to 0, the cost incurred in the future has no bearing on the current value calculation as when it is raised to power n, it’s almost 0 (or exactly 0 in our case). Hence, the agent in one of the non terminal but non rewarding state can choose to stay there because the discounted short term reward of staying surpasses the discounted long term cost of repeatedly failing and leave it in a certain stage.

Ans 3. I feel a simple way to do this is to not just set the terminal state with reward and rest as -1. Rather, we should use Manhattan distance between the current state and terminal state (easy since there is only 1 terminal state in our case and thus, we do not have to take the minimum), and use that as the reward function. However, we want to keep this value low or possibly negative, otherwise the agent will keep running in some sort of loop to have infinitely large value.

This very smoothly worked for me and showed good results (Much better compared to earlier where it was not learning anything). I implemented this by making 2 helper functions. One converted state value to x and y coordinates, while the other computed Manhattan distance from the terminal state. The distance was normalized by dividing by (width+height) of grid (max possible distance) and its negative was used as the reward