CS 520: ASSIGNMENT 3 - PROBABILISTIC SEARCH (AND DESTROY)

Netid: adp178

IMPLEMENTATION:

We implement the map as Numpy matrix with each cell assigned value based on terrain.

```
5]: def map_terrain(size):
        terrain = np.zeros((size,size))
        for i in range(size):
            for  j in range(size):
                x=random.random()
                if x<0.2:
                    terrain[i,j]= 100 #Flat terrain
                elif x<0.5:
                    terrain[i,j]= 101 #hilly terrain
                elif x<0.8:
                    terrain[i,j]= 102 #forested terrain
                else:
                    terrain[i,j]= 103 #maze of caves terrain
        target_loc_row= random.randint(0,size-1)
        target_loc_col=random.randint(0,size-1)

        return terrain,(target_loc_row,target_loc_col)
```

```
size=10
landscape=map_terrain(size)
print(landscape)
```

```
(array([[103., 101., 100., 100., 103., 100., 102., 102., 100., 101.],
       [100., 102., 102., 100., 102., 103., 102., 101., 102., 102.],
       [102., 103., 102., 101., 100., 101., 103., 103., 100., 102.],
       [102., 102., 103., 101., 101., 102., 101., 101., 102., 102.],
       [102., 101., 100., 101., 103., 102., 103., 103., 102., 102.],
       [101., 101., 101., 101., 103., 101., 102., 103., 103., 103.],
       [100., 101., 101., 100., 102., 102., 103., 103., 101., 102.],
       [100., 101., 102., 100., 101., 100., 101., 102., 102., 100.],
       [100., 102., 101., 100., 103., 100., 100., 100., 102., 100.],
       [100., 101., 101., 102., 100., 103., 101., 101., 103., 101.]]), (7, 7))
```

Cell Value	Terrain
100	Flat
101	Hilly
102	Forested
103	Caves

Terrain for each cell is assigned based on probabilities given (flat with probability 0.2, hilly with probability 0.3, forested with probability 0.3, and caves with probability 0.2). The target is assigned by randomly choosing row and column.

```
def map_specific_Target_Terrain(size,target_terrain_loc):
    terrain = np.zeros((size,size))
    for i in range(size):
        for  j in range(size):
            x=random.random()
            if x<0.2:
                terrain[i,j]= 100 #Flat terrain
            elif x<0.5:
                terrain[i,j]= 101 #hilly terrain
            elif x<0.8:
                terrain[i,j]= 102 #forested terrain
            else:
                terrain[i,j]= 103 #maze of caves terrain
    if target_terrain_loc == 100:
        result = list(zip(*np.where(terrain == 100.0)))
        target_loc=random.choice(result)
        target_loc_row=target_loc[0]
        target_loc_col=target_loc[1]
    if target_terrain_loc == 101:
        result = list(zip(*np.where(terrain == 101.0)))
        target_loc=random.choice(result)
        target_loc_row=target_loc[0]
        target_loc_col=target_loc[1]
    if target_terrain_loc == 102:
        result = list(zip(*np.where(terrain == 102.0)))
        target_loc=random.choice(result)
        target_loc_row=target_loc[0]
        target_loc_col=target_loc[1]
    if target_terrain_loc == 103:
        result = list(zip(*np.where(terrain == 103.0)))
        target_loc=random.choice(result)
        target_loc_row=target_loc[0]
        target_loc_col=target_loc[1]


    return terrain,(target_loc_row,target_loc_col)
```

A second map function is used to add target in a specific terrain. This will be used during further analysis.

I used prob_table and prob_table2 to find the probability of each cell containing target. We keep updating the tables after every search. Initially all cells in prob_table and prob_table2 will have probability prob_table [Cell i] $= \frac{1}{\dim * \dim}$.

PART 1: A STATIONARY TARGET

1) **Given observations up to time t (Observations at t), and a failure searching Cell j (Observations at t+1 = Observations at t ∧ Failure in Cell j), how can Bayes' theorem be used to efficiently update the belief state, i.e., compute: P (Target in Cell i |Observations at t ∧ Failure in Cell j)**

First cell we search_cell will be randomly selected as prob_table will have same probability of 1/dim^2. After first search we will update the prob table using P (Target in Cell i |Observations at t ∧ Failure in Cell j).

P (Target in Cell i |Observations at t ∧ Failure in Cell j) $=$

$$\frac{\text{prior } prob\ table\ value\ for\ search\ cell\ *(False\ negative\ rate\ of\ search\ cell(\ if\ (i,j) of\ prob\ table=(i,j) of\ searchc\ cell)}{\text{prior } prob\ table\ value\ for\ search\ cell * False\ negative\ rate\ of\ search\ cell\ +(1-\text{prior } prob\ table\ value\ for\ search\ cell)}$$

```
[16]: def compute_prob(terrain,prob_table,f_n_r,search_cell,size):

          terrian_of_selected_cell= int(terrain[0][search_cell]%100)
          #print(terrian_of_selected_cell)#to get false negative rate of terrain, if the value of cell is 102,so
          f_n_r_of_search_cell= f_n_r[terrian_of_selected_cell]  #102%100 = 2, then we will query f_n_r[2] to get its false negative r
                                                                 #we have arranged the false negative rate accordingly in the list
          cdr= (1-prob_table[search_cell])+(prob_table[search_cell]*f_n_r_of_search_cell) #common divison factor
          for i in range(size):
              for j in range(size):
                  if (i,j) != search_cell:
                      prob_table[i][j]=prob_table[i][j]/cdr
                  else:
                      prob_table[i][j]= prob_table[i][j]*f_n_r_of_search_cell/cdr
          return
```

We will be updating prob table every time we search a cell.

2) **Given the observations up to time t, the belief state captures the current probability the target is in a given cell. What is the probability that the target will be found in Cell i if it is searched: P (Target found in Cell i |Observations at t)?**

The belief state which captures the probability of the target found in cell i given the observations till time t, will be stored in prob_table 2. Therefore,

$$\text{Prob\_table2[i][}j\text{]} = \text{prob\_table[i][j]} \ * (1 - \text{false negative rate of (i, j)})$$

Prob_table will be computes as shown above, prob_table2 will also be updated every time we search cell.

```
17]: def compute_prob_rule2(terrain,prob_table,prob_table2,f_n_r,search_cell,size):

         for i in range(size):
             for j in range(size):
                 x=np.int(terrain[0][i][j]%100)
                 prob_table2[i][j]=prob_table[i][j]*(1-f_n_r[x])
         return
```

3) **comparing the following two decision rules:**

```
# selecting max value from prob table, giving next coordinates for searching cell
def decision_rule1(prob_table,size):
    result = list(zip(*np.where(prob_table == np.amax(prob_table))))
    new_search_loc=random.choice(result) # chossing a random location if multiple cells have max probabiltiy value
    return new_search_loc
```

```
# selecting max value from prob_table2, giving next coordinates for searching cell
def decision_rule2(prob_table2,size):
    result = list(zip(*np.where(prob_table2 == np.amax(prob_table2))))
    new_search_loc=random.choice(result)# chossing a random location if multiple cells have max probabiltiy value
    return new_search_loc
```

– Rule 1: At any time, search the cell with the highest probability of containing the target. – Rule 2: At any time, search the cell with the highest probability of finding the target.

In Rule 1, the location to search cell will be based on highest probability of containing target after updating prob_table after every iteration. However, in Rule 2, the location of which cell to search next is based on cell with highest probability of finding target (max value in prob_table2). It is important to note that prob_table2 is updating only after updating prob_table.

Below are the results for dimension 50x50 averaged over 10 iterations for the same map:

Below are the results for dimension 50x50 averaged over 10 iterations for the multiple maps:

In general Rule 2 has a smaller number of searches compared to Rule 1. This is also understandable because Rule 2 has more information than rule 1, Rule 2 also accounts for false negative rates as well as the belief updation from prob_table. But I also observed that Rule 2 takes as much time as Rule 1(or more) when we assign target in cave, this is also because caves have high false negative rates associated with it (0.9). But it is also important to note that Rule 2 is more computationally expensive than Rule 1. But in patterns for Rule 1 and Rule 2 hold across multiple maps as well as with increasing dimensions.

Running Rule 2(Agent 2) with dimension 50x50 over 10 iterations and setting target always in cave:

```
[14111, 5681, 10043, 7302, 5264, 19302, 29927, 17093, 27779, 6715] 14321.7
```

→ These are the number of searches for rule 2 when target is in the cave. The average for 10 searches is 14321 searches.

```
[16706, 7486, 6270, 24527, 16221, 356, 8670, 5041, 17, 5698] 9099.2
```

→ These are the number of searches for rule 1 when target is in the cave. The average for 10 searches is 9099 searches. (It got really lucky with 6th and 9th iteration).

But as we can see that rule2 doesn't have edge when the target is always in the cave.

4) **A. How can you use the belief state and your current location to determine whether to search or move (and where to move), and minimize the total number of actions required?**

All the agent actions are governed by its current search cell location and its belief state. The agent searches the cell based on its current belief state and if the target is not found it updates its current belief and moves on to next search location. That is-

At t=0, each cell has equal probability of containing target. We open a random cell and if the target is not found we update the current belief and in this updation we also reduce the probability of finding target in current search cell by some factor(which is dependent on false negative rate of search cell). Furthermore, we compute the Manhattan distance from current search cell to all the cells. Then we take ratio of the Manhattan distance from current search cell location to probability of finding the target in the cell. The cell with minimum score is then set as next search location.

So, an inverse relationship exists there: As the Manhattan distance increases our belief will decrease (though this also depends upon the probability of finding the target in the cell). Here we will not just consider the searches but the also the motion from moving from current search cell to the next search cell. The next search location will be cell with highest value. It is important to note that the next search cell with highest value is likely to be closest to the current search cell as well as likely to have the target. This will help us reduce total number of actions required to find the target.

B. compare the following agents:

- Basic Agent 1: Simply travel to the nearest cell chosen by Rule 1, and search it.
- Basic Agent 2: Simply travel to the nearest cell chosen by Rule 2, and search it.
- Basic Agent 3: At every time, score each cell with (Manhattan distance from current location)/(probability of finding target in that cell); identify the cell with minimal score, travel to it, and search it.
- Improved Agent: Your own strategy, that tries to beat the above three agents.

For modified agent:

We will be implementing modified agent by improvising existing basic agent 3. We select minimum score which represents high likelihood for target being in the cell as well as being closer to current search cell. When we search the next cell, there is a high the chance of target being missed for terrain cave and forested as they have high false negative rates. So, instead of moving to next search location, we will be searching the cell with minimum score based on the terrain type of the cell. For example, the minimum score cell found has a terrain type of cave, so number of times it will search that cell before moving on will be equal to:

Terrain Type	Number of searches before moving on
Flat	1 (due to low false negative rates)
Hilly	1 (due to low false negative rates)
Forested	False negative rate*10 (0.5*10)
Caves	False negative rate*10 (0.9*10)

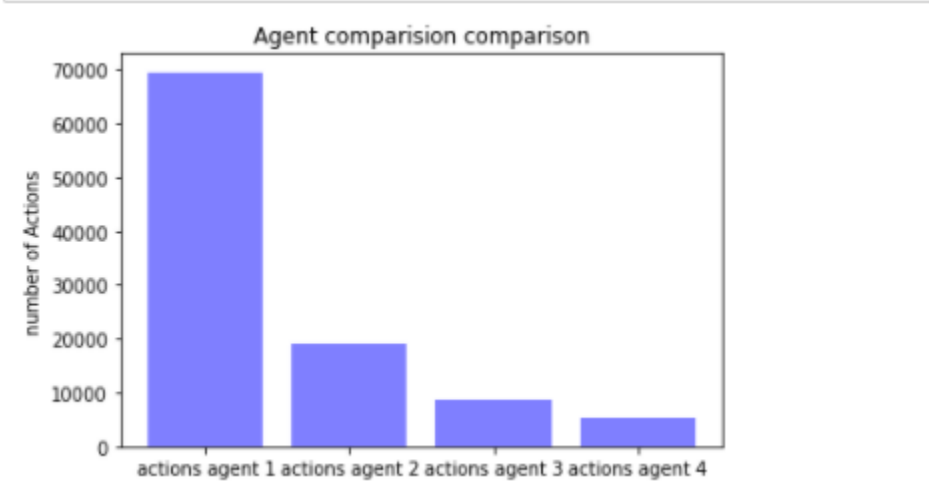
I got the factor 10 right now by manually running trials for different constant values. There is further scope for improving this based on optimizing geometric sum of probabilities for number of times cell is searched with respect to false negative rates.

This saves us many future actions which may have happened in the scenario where we moved to next search location even when the target was in the cell due to false negative rate.

The comparison for agents is given below:

Dimension of landscape: 30x30

Number of iterations: 20, map and target were changed after every iteration.



Number of actions presented is average of 10 iterations. As we can see minimum number of actions required for searching the target is less for agent 3 and 4. The number of actions is sum of searches and Manhattan units between each search location and next search location. The maximum contribution is of the steps it needs to travel from one search location to the next. The results showed by agent 3 and agent 4 is representative of this.

Terrain specific target Analysis for each Agent

Dimension: 20x20

Number of iterations: 10, map is generated for each iteration.

1. For Agent 1

We can see that as target moves from flat to the number of actions increase exponential.

2. For Agent 2

As we can number of actions are nearly 20,000 less compared to agent 1 when the target is always in forested and cave as well.

3. For Agent 3

The performance of agent 3 is drastically better compared to 2 or 1 irrespective of whether the target was always in flat or always in cave.

4. For agent 4

The 4th agent beat agent 3 when target is always in the cave or forest but had an upper hand when target was in the hills. But That may be due to random allocation of target and different map for each iteration.

As always this was a very fun project and learnt a lot of new things. Thank you!