FINAL EXAM

ARTIFICIAL INTELLIGENCE - CS520

Zeeshan Ahsan

NET ID – ZA224 | RUTGERS UNIVERSITY

PROBLEM DESCRIPTION

The given an environment in the form a grid which has walls and open spaces.

The problem is to locate a drone in the grid which is present somewhere in the open spaces in the grid.

I am allowed to control the drone and give it commands to control its behavior inside the grid. The commands which are available to me are namely – Up, Down, Left and Right.

The problem is to locate the exact position of the drone after certain number of commands even though I can't see the drone.

PROBLEM FORMULATION

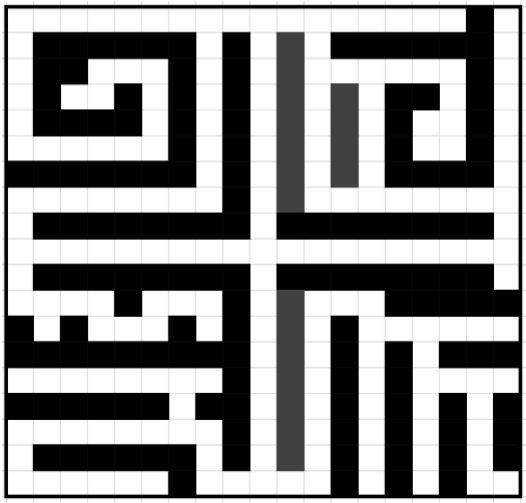
As I do not know about the exact location of the drone, I have used a system and collection of beliefs for the drone being in a certain position. These beliefs always should sum to one.

My general idea is that after a certain number of commands of Up, Down, Left and Right in a feasible order, my beliefs will change regarding the position of the drone in the grid.

At the point that my belief becomes 1 at any one position in the grid, I will know with certainty about the position of the drone.

GIVEN GRID – named as question_grid.txt

Figure 1: TH23-SA74-VERW Reactor Schematic



Question 1 (5 Points): Before you do anything, what is the probability that the drone is in the top left corner? Why?

As I have no prior knowledge about the position of the drone, I equal belief in all the free cells of the grid to contain the drone.

By this logic, for this grid which is given the probability of drone in the top left corner is (1/ total number of free cells) which equals to 0.005025125628140704.

```
Belief of [0][0]:  0.005025125628140704

(base) zeeshanahsan@Zeeshans-MacBook-Pro Code
```

Code used to show this:

```
grid = readFile_and_returnGrid()
beliefMatrix = initialize_beliefMatrix(grid)
print(beliefMatrix)
print(beliefMatrix[0][0])
```

Consider issuing the command 'DOWN'. While you don't know exactly where the drone is, you can say where it *isn't* - it isn't, for instance, in the top left corner anymore.

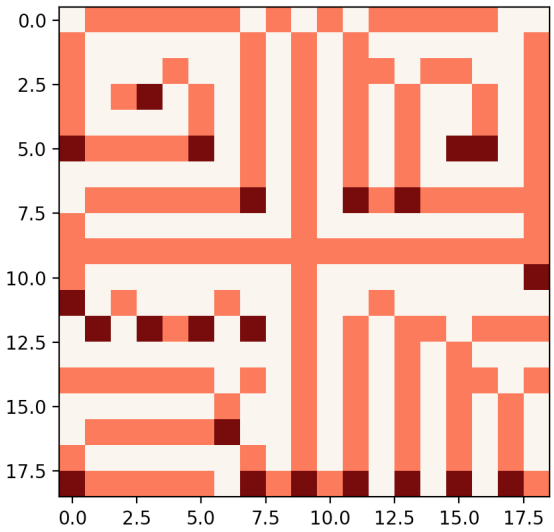
Question 2 (5 Points): What are the locations where the drone is most likely to be? Least likely to be? How likely is it to be in all the other locations? Indicate your results visually.

After issuing the first down command, all the cells on the topmost row will have zero probability of having the drone. I know for sure that these positions won't have the drone.

The cells which are above walls in each column will have the highest probabilities of containing the drone after issuing the first down command.

This is because, P(drone in cell above wall now) = P(drone in cell above wall before) + P(drone in cell above that cell before).

The distribution of probabilities after the first down command is as follows:



Here, we can confirm visually that the probabilities are maximum in cells which are just above walls in each column. We also see that the probabilities are zero in the cells which are immediately below walls in each column.

The code used to show this is:

```
grid = readFile_and_returnGrid()
beliefMatrix = initialize_beliefMatrix(grid)
print(beliefMatrix)
beliefMatrix = moveDown(grid, beliefMatrix)
plt.imshow(beliefMatrix, cmap='Reds')
plt.show()
```

Question 3 (25 Points): Write a program that takes a reactor schematic as a text file (see associated file for this reactor) and finds a sequence of commands that, at the end of which, you know *exactly* what cell the drone is located in. Be clear in your writeup about how you are formulating the problem, and the algorithms you are using to find this sequence.

What is the sequence for this reactor?

CODE FOR READING INPUT

```
def readFile_and_returnGrid():
    #Read text File
    with open("FinalExam520/inputGrid.txt") as file_in:
        lines = []
        for line in file_in:
            lines.append(line)

    no_of_rows = len(lines)
    no_of_cols = len(lines[0]) - 1

    print("No of rows : ", no_of_rows)
    print("No. of Cols :", no_of_cols)

    #Make a grid with 1's as walls and 0's as open cells
    grid = np.zeros((no_of_rows,no_of_cols),dtype=(int))
    #print(grid)
    for i in range(no_of_rows):
        for j in range(no_of_cols):
            #If character is 'X' -> it is a wall -> assign 1 to the matrix
            if(lines[i][j] == 'X'):
                grid[i][j] = 1

    #print(grid)
    return(grid)
```

In the above code, I read the input line by line. After reading the input, I have made the environment for the problem. The environment is a 2-D grid. Here,

the number of rows and columns of the 2-D grid are same number of rows and columns in the input text file.

I have created a 2-D array with all zeros initially. Whenever I encounter an 'X' character, I insert a '1' at the corresponding 2-D grid indices as shown in code above. In the end, the grid comprises of 0's and 1's with 0's as free cells in the grid and 1's as walls in the grid. After this, I return the grid from this function.

INITIALIZATION OF BELIEF / PROBABILITY MATRIX

The initialization of belief is done in initialize_beliefMatrix(grid) function. This takes the grid as an input parameter. Here, the beliefMatrix is again a 2-D matrix of float values. Each value in the matrix corresponds to the same index cell in the grid. For example beliefMatrix[0][0] holds the probability of drone in grid[0][0] position.

During the initialization, the probabilities are divided evenly in all open cells of the grid as we do not know anything about the location of the drone in the beginning. The probabilities corresponding to the location of walls are assigned a value of zero in the beliefMatrix as the drone can't be there. In the end, this method returns the beliefMatrix.

CODE FOR BELIEF INITIALIZATION

```
def initialize_beliefMatrix(grid):

    no_of_rows = len(grid)
    no_of_cols = len(grid[0])

    #Belief Matrix to capture belief of containing drone in each cell
    beliefMatrix = np.zeros((no_of_rows,no_of_cols),dtype=(float))

    #Count number of open cells in the grid
    countCells = 0
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if(grid[i][j] == 0):
                countCells += 1

    # Divide Belief equally for all open cells
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if(beliefMatrix[i][j] == 0):
                beliefMatrix[i][j] = 1/countCells

    #Set Belief of walls to be zero
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if(grid[i][j] == 1):
                beliefMatrix[i][j] = 0

    return beliefMatrix
```

PROBABILITY TRANSITIONS AFTER EACH ACTION

1. ***MOVE LEFT*** – This action is implemented in the function moveLeft(grid, beliefMatrix). This function takes in the grid and the beliefMatrix as the inputs.

   Initially, I make a copy of the beliefMatrix into copy_of_beliefs so that I can access the belief values before the move and update the new values in the beliefMatrix.

   Here, I iterate through the beliefMatrix row-wise as the probabilities will shift left row-wise after left command. The following are the ways in which the probability changes are realized in the code –

- ***Case for 1st column-*** In the leftmost column of the row(j=0) which is next to the boundary of the grid, I check if the cell is not a wall and the cell on the right to this cell is not a wall. If these conditions are met then, I update the probability of current cell as the probability from copy_of_beliefs for current cell + probability in copy_of_beliefs in the cell present on the immediate right of this cell after a left move.

- ***General case when there is no left wall or right wall of the current cell –*** In this case I check if we are within the boundaries of the grid, and if the cell left and cell right to the current cell are not walls and the current cell is not in the 1st column. If these conditions are met then, I update the probability of the current cell as the probability of the cell right to the current cell from the copy_of_beliefs matrix after a left move.

- ***When the left cell is a wall and right cell is not a wall of the current cell –*** In this case, I check if we are within the boundaries of the grid, and if the left cell is a wall and right cell is not a wall of the current cell and the current cell is not in 1st column. If these conditions are met then, I update the current probability of this cell as the probability from copy_of_beliefs for this cell + probability in copy_of_beliefs in the cell present on the immediate right of this cell after a left move.

- ***When there is a wall on the right side of the current cell –*** In this case, I check if we are within the boundaries of the grid, and if there is a walled cell on the right of the current cell and the current cell is not a wall and the current cell is not in the first column. If these conditions are met, then the probability of the current cell is zero after a left move.

- ***When the current cell is in the last column –*** In this I check if the current cell is in the last column of the grid, and the cell is not a wall and the cell on the left of the current cell is not a wall. If these conditions are met, then I update the probability of current cell as zero after a left move.

- ***When there are walls on left and right of the current cell –*** In this case, I check if the current cell is not in the 1st column, and the current cell is within the boundaries of the grid, and the current cell has walls on the left and right, and the current cell is not a wall. If these conditions are met, then I update the probability of the current cell as the probability of the same cell from the copy_of_beliefs matrix after a left move.

- ***CODE FOR MOVE LEFT –***

```
def moveLeft(grid,beliefMatrix):
    copy_of_beliefs = np.copy(beliefMatrix)
    rows = len(grid)
    cols = len(grid[0])

    #ITERATING ROW WISE
    for i in range(rows):
        for j in range(cols):
            #belief(col1) = belief(col1) + belief(col2) if col1 is not wall
            if(j == 0 and grid[i][j] != 1 and grid[i][j+1] != 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i][j+1]

            #General Case for no left wall or right wall
            if(j+1 <= cols-1 and j != 0 and grid[i][j] !=1 and grid[i][j+1] != 1 and grid[i][j-1] != 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j+1]

            #When left cell is wall and right cell is no wall
            if(j+1 <= cols-1 and j != 0 and grid[i][j] !=1 and grid[i][j+1] != 1 and grid[i][j-1] == 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i][j+1]

            #When there's a wall on the right side
            if(j+1 <= cols-1 and j != 0 and grid[i][j] !=1 and grid[i][j+1] == 1):
                beliefMatrix[i][j] = 0

            #Set Last Col as Zero if it is in the last column and doesn't have wall on the left
            if(j == (cols-1) and grid[i][j] != 1 and grid[i][j-1] != 1):
                beliefMatrix[i][j] = 0

            #If Left and Right both walls -> No change in belief
            if(j != 0 and j+1 <= cols - 1 and j-1 >=0 and grid[i][j] != 1 and grid[i][j-1] == 1 and grid[i][j+1] == 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j]

    #Return Updated Probability Matrix
    return beliefMatrix
```

2. ***MOVE RIGHT –*** I have updated the probabilities with similar logic after right move as well. The conditions are same and are just modified for a right move.

CODE FOR MOVE RIGHT

```
def moveRight(grid,beliefMatrix):
    copy_of_beliefs = np.copy(beliefMatrix)
    rows = len(grid)
    cols = len(grid[0])

    #Iterating Row Wise
    for i in range(rows):
        for j in range(cols):
            #belief(col'n') = belief(col'n-1') + belief(col'n') and col'n' is not wall
            if(j == cols - 1 and grid[i][j] != 1 and grid[i][j-1] != 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i][j-1]

            #General Case for no left or right wall
            if(j-1 >= 0 and j != cols -1 and grid[i][j] != 1 and grid[i][j+1] != 1 and grid[i][j-1] != 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j-1]

            #When right cell is wall and left cell is no wall
            if(j-1 >= 0 and j != cols -1 and grid[i][j] != 1 and grid[i][j+1] == 1 and grid[i][j-1] != 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i][j-1]

            #When there's a wall on the left side
            if(j-1 >= 0 and j != cols -1 and grid[i][j] != 1 and grid[i][j-1] == 1):
                beliefMatrix[i][j] = 0

            #Set 1st Col as zero and doesn't have wall on the right
            if(j == 0 and grid[i][j] != 1 and grid[i][j+1] != 1):
                beliefMatrix[i][j] = 0

            #If Left and Right both walls -> No change in belief
            if(j != 0 and j+1 <= cols - 1 and j-1 >=0 and grid[i][j] != 1 and grid[i][j-1] == 1 and grid[i][j+1] == 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j]

    #Return Updated Probability Matrix
    return beliefMatrix
```

3. ***MOVE UP*** **–** Here also I have used the same logic to update probabilities but here I iterate column wise as move up/ down will affect probabilities column wise.

CODE FOR MOVE UP –

```
def moveUp(grid,beliefMatrix):
    copy_of_beliefs = np.copy(beliefMatrix)
    rows = len(grid)
    cols = len(grid[0])

    # Iterating Column wise
    for j in range(cols):
        for i in range(rows):
            #belief row1 = belief(row1) + belief(row2) if row is not wall
            if(i==0 and grid[i][j] != 1 and grid[i+1][j] != 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i+1][j]

            #General case for no up wall or down wall
            if(i+1 <= rows - 1 and i != 0 and grid[i][j] != 1 and grid[i+1][j] != 1 and grid[i-1][j] != 1):
                beliefMatrix[i][j] = copy_of_beliefs[i+1][j]

            #When Cell Up is Wall and Cell Down is no wall
            if(i+1 <= rows - 1 and i != 0 and grid[i][j] != 1 and grid[i+1][j] != 1 and grid[i-1][j] == 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i+1][j]

            #When there's a Wall Down
            if(i+1 <= rows-1 and i != 0 and grid[i][j] != 1 and grid[i+1][j] == 1):
                beliefMatrix[i][j] = 0

            #Set Last Row as Zero
            if(i == rows - 1 and grid[i][j] != 1 and grid[i-1][j] != 1):
                beliefMatrix[i][j] = 0

            #If Up and down both walls -> No change in belief
            if(i != 0 and i+1 <= rows - 1 and i-1 >=0 and grid[i][j] != 1 and grid[i+1][j] == 1 and grid[i-1][j] == 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j]

    #Return updated Probability Matrix
    return beliefMatrix
```

4. ***MOVE DOWN –*** Here also I have used the same logic to update probabilities. Here also I iterate column wise.

CODE FOR MOVE DOWN –

```
def moveDown(grid,beliefMatrix):
    copy_of_beliefs = np.copy(beliefMatrix)
    rows = len(grid)
    cols = len(grid[0])

    #Iterating column wise
    for j in range(cols):
        for i in range(rows):
            #belief row'n' = belief(row'n') + belief(row'n-1') if row is not wall
            if(i== rows -1 and grid[i][j] != 1 and grid[i-1][j] != 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i-1][j]

            #General case for no up wall or down wall
            if(i+1 <= rows - 1 and i != 0 and grid[i][j] != 1 and grid[i+1][j] != 1 and grid[i-1][j] != 1):
                beliefMatrix[i][j] = copy_of_beliefs[i-1][j]

            #When Cell Down is Wall and Cell Up is no wall
            if(i+1 <= rows - 1 and i != 0 and grid[i][j] != 1 and grid[i+1][j] == 1 and grid[i-1][j] != 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i-1][j]

            #When there's a Wall Up
            if(i-1 >= 0 and i != rows - 1 and grid[i][j] != 1 and grid[i-1][j] == 1):
                beliefMatrix[i][j] = 0

            #Set First Row as Zero
            if(i == 0 and grid[i][j] != 1 and grid[i+1][j] != 1):
                beliefMatrix[i][j] = 0

            #If Up and down both walls -> No change in belief
            if(i != 0 and i+1 <= rows - 1 and i-1 >=0 and grid[i][j] != 1 and grid[i+1][j] == 1 and grid[i-1][j] == 1):
                beliefMatrix[i][j] = copy_of_beliefs[i][j]

    #Return Updated Belief Matrix
    return beliefMatrix
```

Sample Dry Probability Outputs for Small Grid:

Here I have used a small grid of with 3 rows and 11 columns to show the probability changes.

Grid Used –

XXX_____XXX
_____X_____
XXX_____XXX

Commands Used –

```
grid = readFile_and_returnGrid()
beliefMatrix = initialize_beliefMatrix(grid)
print("Initial Belief Distribution")
print(beliefMatrix)
print()
print("After Move Left")
beliefMatrix = moveLeft(grid, beliefMatrix)
print(beliefMatrix)
print()
print("After Move Up")
beliefMatrix = moveUp(grid, beliefMatrix)
print(beliefMatrix)
print()
print("After Move Right")
beliefMatrix = moveRight(grid, beliefMatrix)
print(beliefMatrix)
print()
print("After Move Down")
beliefMatrix = moveDown(grid, beliefMatrix)
print(beliefMatrix)
print()
```

Stepwise Probabilities After the above Operations –

```
Initial Belief Distribution
[[0.   0.   0.   0.05 0.05 0.05 0.05 0.05 0.   0.   0.  ]
 [0.05 0.05 0.05 0.05 0.05 0.   0.05 0.05 0.05 0.05 0.05]
 [0.   0.   0.   0.05 0.05 0.05 0.05 0.05 0.   0.   0.  ]]

After Move Left
[[0.   0.   0.   0.1  0.05 0.05 0.05 0.   0.   0.   0.  ]
 [0.1  0.05 0.05 0.05 0.   0.   0.1  0.05 0.05 0.05 0.  ]
 [0.   0.   0.   0.1  0.05 0.05 0.05 0.   0.   0.   0.  ]]

After Move Up
[[0.   0.   0.   0.15 0.05 0.05 0.15 0.05 0.   0.   0.  ]
 [0.1  0.05 0.05 0.1  0.05 0.   0.05 0.   0.05 0.05 0.  ]
 [0.   0.   0.   0.   0.   0.05 0.   0.   0.   0.   0.  ]]

After Move Right
[[0.   0.   0.   0.   0.15 0.05 0.05 0.2  0.   0.   0.  ]
 [0.   0.1  0.05 0.05 0.15 0.   0.   0.05 0.   0.05 0.05]
 [0.   0.   0.   0.   0.   0.   0.05 0.   0.   0.   0.  ]]

After Move Down
[[0.   0.   0.   0.   0.   0.05 0.   0.   0.   0.   0.  ]
 [0.   0.1  0.05 0.   0.15 0.   0.05 0.2  0.   0.05 0.05]
 [0.   0.   0.   0.05 0.15 0.   0.05 0.05 0.   0.   0.  ]]
```

IDEA FOR SOLVING THE GRID AND LOCATING THE DRONE

The main idea to locate the drone efficiently is to make my belief matrix zeros in all places and 1 in a cell. This denotes that I have 100% certainty that the drone is in a particular location where the belief matrix has a value of 1.

Essentially, I have to try to make everything 0 by making Left, Right, Up, Down moves.

I started by a brute-forcing solution in which I move L->U->D->R sequentially till I get a 1 in beliefMatrix. This approach was highly infeasible.

I modified my approach of moving right, equal to the number of columns and then moving down equal to the number of rows, then moving up equal to the number of rows and then moving right equal to the number of columns. Using this strategy, I was trying to essentially 'collect' the probabilities next to the walls of the grid and make other cells probabilities zero. I tried to do these till I have one probability of 1 left in the beliefMatrix.

After these attempts I realized that the most effective way to make probabilities 'accumulate' together as fast as possible is to merge two nearest non-zero probabilities. If I can merge the probabilities in shortest possible moves, I will have the shortest sequence of steps to get the exact location of the drone. This can be realized in the most effective way by the method discussed below.

For this, I start by finding out two cells with non-zero probabilities row wise. After I have the location of these probabilities from the grid, I use the Dijkstra's algorithm to find the shortest path between these two cells. I save this shortest path and execute the commands to reach from one cell to the other. For example, the cells are (0,0) and (1,5) and the path is (0,0) -> (0,1) -> (0,2) -> (0,3) -> (0,4) -> (0,5) -> (1,5). So, I execute the command right, right, right, right, right, down. I again find the two cells with non-zero probabilities and execute the step as explained above. This is done till I have only 1 probability left in the beliefMatrix. I do this because it gives me the best possible move each time in every iteration to merge 2 non-zero probabilities. And hence, I am confident that this will generate the shortest possible steps to exactly spot the drone.

Start State – When there are probabilities equally distributed in the beliefMatrix.

Goal State – When there is only 1 probability left in the beliefMatrix with a value of 1.

Actions – Left, Right, Up, Down

Cost – Since we are doing 1 move in every step and we are trying to minimize the number of steps, the cost of each move is +1.

ALGORITHM –

1. Repeat till we have only 1 probability in the beliefMatrix.

2. Find 2 cells which have non-zero probabilities in row – wise manner. (This *is done in find_2_cells (grid, beliefMatrix) function.)*

3. If there is only 1 cell returned from this search -> end (As *only 1 non-zero probability is left in beliefMatrix).*

4. Calculate the shortest distance between these cells and formulate a path with an action list. (Example*: if shortest path from (0,0) to (1,3) is (0,0) -> (0,1) -> (0,2) -> (1,2) -> (1,3), then the action list would be right, right, down, right.)*

5. Execute the commands from the list and add these commands in the movesList.

6. When we have only 1 probability left in the matrix, output the movesList which will contain the moves required to exactly spot the drone location and calculate the number of steps required by finding the length of this list. This is also the cost of the entire execution of the algorithm.

A Brief about the description of the methods used in this algorithm:

1. **find_2_cells(grid, beliefMatrix) –** This method finds the location of the two non-zero locations in the beliefMatrix and returns these locations in the form of a list.

CODE –

```
def find_2_Cells(grid,beliefMatrix):
    rows = len(grid)
    cols = len(grid[0])
    cells_list = []
    count = 0
    #iterate through the belief matrix row-wise
    for i in range(rows):
        for j in range(cols):
            # If there are 2 cells in the cells_list break and return
            if(count == 2):
                break
            #Find the non-zero entries in the matrix and append their locations to the cells_list
            if(beliefMatrix[i][j] != 0):
                cells_list.append((i,j))
                count +=1

    #Return the cells_list with locations(x,y) of the 2 non-zero values in the beliefMatrix
    return cells_list
```

Example –

2 non-zero belief indices calculated using this function for the below beliefMatrix.

```
 Belief Matrix

 [[0.   0.   0.   0.   0.   0.05 0.   0.   0.   0.   0.  ]
  [0.05 0.05 0.05 0.05 0.05 0.   0.05 0.05 0.05 0.05 0.05]
  [0.   0.   0.   0.1  0.1  0.05 0.1  0.1  0.   0.   0.  ]]

 2 Non - Zero Cells indices :  [(0, 5), (1, 0)]
```

Code used for example run –

```
grid = readFile_and_returnGrid()
beliefMatrix = initialize_beliefMatrix(grid)

beliefMatrix = moveDown(grid, beliefMatrix)
print()
print("Belief Matrix")
print()
print(beliefMatrix)
print()
CellList = find_2_Cells(grid, beliefMatrix)
print("2 Non - Zero Cells indices : ", CellList)
print()
```

2. **shortestDistance_ActionsList(grid, beliefMatrix, start, end) –** This method formulates a list of actions to reach from ‘start cell’ to the ‘end cell’ using the shortest path calculated using dijkstra’s algorithm. This returns an array of moves required to reach from start cell to end cell. Example – [up, down, left, left, right]

CODE –

```
def shortestDistance_ActionsList(grid,beliefMatrix,start,end):
    # Reverse Path List of the path of cells using dijkstra algorithm from start to end
    pathList = djikstra(grid, start, end)
    # Appending the start cell to the list
    pathList.append(start)
    # Reversing the reverse Path to get a forward path from start to end
    pathList.reverse()
    # Build the executable actions list
    actionList = []
    for i in range(len(pathList) - 1):
        #Extract the current cell from path list
        curCell = pathList[i]
        #Extract the x & y co-ordinates of the cell
        x_curCell,y_curCell = curCell[0],curCell[1]
        #Extract the next cell in the path list
        nextCell = pathList[i+1]
        #Extract the x & y co-ordinates of the next cell in the path list
        x_nextCell,y_nextCell = nextCell[0],nextCell[1]

        # Formulate moves required to go from one cell to the next cell

        #Same Row, Diff Col -> Left or Right
        if(x_curCell == x_nextCell):
            #Move Right
            if(y_curCell + 1 == y_nextCell):
                actionList.append("right")
            #Move Left
            elif(y_curCell - 1 == y_nextCell):
                actionList.append("left")

        #Same Col, Diff Row -> Up or Down
        elif(y_curCell == y_nextCell):
            #Move Down
            if(x_curCell + 1== x_nextCell):
                actionList.append("down")
            #Move Up
            elif(x_curCell - 1 == x_nextCell):
                actionList.append("up")
    #Return the list of actions required to move from start to end
    return actionList
```

Example – Here the ActionList contains the list of actions required to move from (0,5) to (1,0) on shortest path.

```
Belief Matrix

[[0.   0.   0.   0.   0.   0.05 0.   0.   0.   0.   0.  ]
 [0.05 0.05 0.05 0.05 0.05 0.   0.05 0.05 0.05 0.05 0.05]
 [0.   0.   0.   0.1  0.1  0.05 0.1  0.1  0.   0.   0.  ]]

2 Non - Zero Cells indices :  [(0, 5), (1, 0)]

Shortest Action List to reach from start to end in CellList

ActionList :  ['left', 'left', 'down', 'left', 'left', 'left']
```

Code used for example run –

```
grid = readFile_and_returnGrid()
beliefMatrix = initialize_beliefMatrix(grid)

beliefMatrix = moveDown(grid, beliefMatrix)
print()
print("Belief Matrix")
print()
print(beliefMatrix)
print()
CellList = find_2_Cells(grid, beliefMatrix)
print("2 Non - Zero Cells indices : ", CellList)
print()
print("Shortest Action List to reach from start to end in CellList")
print()
actionList = shortestDistance_ActionsList(grid, beliefMatrix, CellList[0], CellList[1])
print("ActionList : ", actionList)
print()
```

3. **djikstra(grid, start, end) –** In this method I find the shortest path from the start cell to the end cell in the grid and return the path from start cell to the end cell.

CODE –

```
def djikstra(grid,start,end):

    n_rows = len(grid)
    n_cols = len(grid[0])
    unvisited = {}
    curCell = start
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            unvisited[(i,j)] = float('inf') #Instantiate the unvisted dictionary for all keys to infinity cost

    unvisited[start] = 0    # Set the start cell as zero cost
    visited = {}    # Make an empty visited dictionary
    revPath = {}    # Make an empty reverse path Dictionary

    curCell = min(unvisited, key=unvisited.get) # Assign the current cell as the lowest cost cell frem the dictionary of unvisited cells
    #print(curCell)

    while unvisited:    # Traverse till unvisited dictionary gets empty
        curCell = min(unvisited, key=unvisited.get) # Assign the current cell as the lowest cost cell frem the dictionary of unvisited cells
        x,y = curCell[0],curCell[1] # Take the x and y cio-ordinates of the current cell
        visited[curCell] = unvisited[curCell] # Add the Value of Unvisited dictionary for current cell to the corresponding key in visited d
        if(curCell == end): # If we reach goal cell
            if(end in revPath.keys()):  # If goal is in keys of the reverse path dictionary
                path = buildPath(revPath,start,end) # Then build a path
                pathList = []   # Make an empty path
                for key,val in path.items():
                    pathList.append(val)    # Make an array of cells which only contain reverse path
                return (pathList)
            else:
                return(-2)
```

```
    #Check Neighbors of the current Cell
    neighborList = []
    if(y+1 <= n_cols - 1 and grid[x][y+1] !=1 ): #Rneighbor
        neighbor = (x,y+1)
        neighborList.append(neighbor)
        #print("curCell :" ,neighbor)
    if(y-1 >= 0 and grid[x][y-1] !=1): #Lneigh
        neighbor = (x,y-1)
        neighborList.append(neighbor)
        #print("curCell :" ,neighbor)
    if(x+1 <= n_rows - 1 and grid[x+1][y] !=1): #DNeigh
        neighbor = (x+1,y)
        neighborList.append(neighbor)
        #print("curCell :" ,neighbor)
    if(x-1 >= 0 and grid[x-1][y] != 1): #UNeigh
        neighbor = (x-1,y)
        neighborList.append(neighbor)
        #print("curCell :" ,neighbor)


    #Check if Neighbor is already present in visited dict
    for nbr in neighborList:
        if(nbr in visited): # For each valid neighbor if it ios already visited then do nothing and continue to the next neighbor
            continue
        tempDist = unvisited[curCell] + 1
        if(tempDist < unvisited[nbr]):
            unvisited[nbr] = tempDist   # Update the distance of neighbor cell if the current distance is less than previous distance in
            revPath[nbr] = curCell  # Set the reverse path key of neighbor cell to current cell value

    unvisited.pop(curCell)  # Remove the current cell from visited cell dictionary
```

FINAL METHOD TO SOLVE THE PROBLEM:

The solve_shortest(grid, beliefMatrix) method solves the problem of finding the drone's exact position in the shortest possible sequence of moves using the Algorithm explained above.

CODE –

```
def solve_shortest(grid, beliefMatrix):
    rows = len(grid)
    cols = len(grid[0])
    #List containing the final moves
    movesList = []
    flag = 0
    while(True):
        #Find 2 nearest cells from top with non-zero probability
        cells_list = find_2_Cells(grid, beliefMatrix)
        #If only 1 non-zero cell left in beliefMatrix -> Drone found
        if(len(cells_list) == 1):
            flag = 1
        # Print Results
        if(flag == 1):
            print()
            print("Moves : ")
            print(movesList)
            print()
            print("BELIEF MATRIX : ")
            print()
            print(beliefMatrix)
            print("Drone Position with 100% Certainity : ", cells_list[0])
            print("Number of Moves Taken : ", len(movesList))
            plt.imshow(beliefMatrix, cmap = 'Blues')
            plt.show()
            break
        cell1 = cells_list[0]
        cell2 = cells_list[1]
        print("Cell 1: ", cell1)
        print("Cell 2: ", cell2)
        #Build Action List
        actionList = shortestDistance_ActionsList(grid, beliefMatrix, cell1, cell2)
        #Execute the actions built
        for i in range(len(actionList)):
            #If action is down -> Execute down move
            if(actionList[i] == "down"):
                beliefMatrix = moveDown(grid, beliefMatrix)
                movesList.append("Down")
                print("Down")
            #If action is up -> Execute up move
            elif(actionList[i] == "up"):
                beliefMatrix = moveUp(grid, beliefMatrix)
                movesList.append("Up")
                print("Up")
            #If action is left -> Execute Left move
            elif(actionList[i] == "left"):
                beliefMatrix = moveLeft(grid, beliefMatrix)
                movesList.append("Left")
                print("Left")
            #If action right down -> Execute right move
            elif(actionList[i] == "right"):
                beliefMatrix = moveRight(grid, beliefMatrix)
                movesList.append("Right")
                print("Right")
```

RESULTS:

Code executed for results:

```
grid = readFile_and_returnGrid()
beliefMatrix = initialize_beliefMatrix(grid)
solve_shortest(grid, beliefMatrix)
```

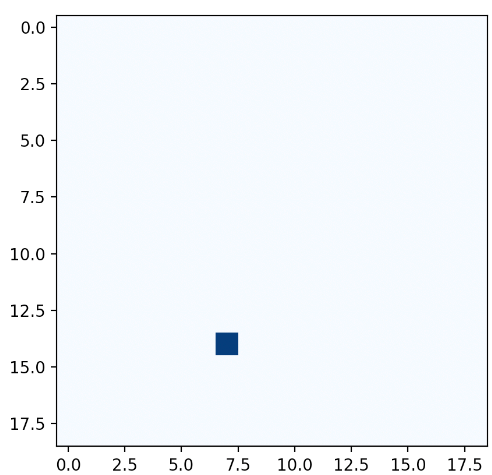
Total Number of moves required to solve the given grid: *332*
Sequence of Steps to Solve the given grid:
['Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Left', 'Left', 'Left', 'Left', 'Left', 'Down', 'Down', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Down', 'Right', 'Right', 'Right', 'Right', 'Right', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Down', 'Right', 'Right', 'Right', 'Right', 'Right', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Left', 'Left', 'Left', 'Left', 'Left', 'Up', 'Up', 'Up', 'Up', 'Up', 'Left', 'Left', 'Up', 'Up', 'Left', 'Left', 'Down', 'Up', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Down', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Up', 'Up', 'Up', 'Up', 'Up', 'Right', 'Right', 'Down', 'Down', 'Right', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Left', 'Down', 'Up', 'Right', 'Up', 'Up', 'Left', 'Left', 'Left', 'Left', 'Left', 'Up', 'Up', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Left', 'Left', 'Left', 'Right', 'Right', 'Right', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Left', 'Left', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Down', 'Up', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Down', 'Down', 'Right', 'Down', 'Up', 'Right', 'Right', 'Down', 'Right', 'Right', 'Up', 'Right', 'Right', 'Down', 'Up', 'Left', 'Left', 'Down', 'Left', 'Left', 'Up', 'Left', 'Left',

'Left', 'Up', 'Up', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Left', 'Left', 'Up', 'Up', 'Left', 'Up', 'Up', 'Right', 'Right', 'Right']

beliefMatrix after the drone's position is found:

```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

Visualization of Probability distribution after drone is found:



Location of Drone: (14, 7)

Terminal Output:

```
Moves :
['Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right
', 'Right', 'Left', 'Left', 'Left', 'Left', 'Left', 'Down', 'Down', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Down', 'Right'
, 'Right', 'Right', 'Right', 'Right', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', '
Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Down', 'Right', 'Right', 'Right', 'Right', 'Right', 'Up', 'Up', 'Up', 'Up', 'Up', '
Up', 'Up', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Left', 'Left', 'Left', 'Left', 'Left', 'Up', 'Up', 'Up', 'Up', 'U
p', 'Left', 'Left', 'Up', 'Up', 'Left', 'Left', 'Down', 'Up', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right',
 'Right', 'Down', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Up', 'Up', 'Up', '
Up', 'Up', 'Right', 'Right', 'Down', 'Down', 'Right', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Left', 'Down', 'Up', 'Righ
t', 'Up', 'Up', 'Left', 'Left', 'Left', 'Left', 'Left', 'Up', 'Up', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', '
Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Up', 'Up', 'Up', 'Up', 'U
p', 'Up', 'Up', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Left', 'Left', 'Left', '
Left', 'Left', 'Left', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Left', 'Left', 'Left', 'Right', 'Righ
t', 'Right', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Down', 'D
own', 'Down', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down
', 'Down', 'Left', 'Left', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right'
, 'Right', 'Right', 'Right', 'Down', 'Up', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'L
eft', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Down', 'Down', 'Right', 'Down', 'Up', 'Right', 'Right', 'Down', 'Right', 'Righ
t', 'Up', 'Right', 'Right', 'Down', 'Up', 'Left', 'Left', 'Down', 'Left', 'Left', 'Up', 'Left', 'Left', 'Left', 'Up', 'Up', 'Right',
'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down
', 'Down', 'Left', 'Left', 'Up', 'Up', 'Left', 'Up', 'Up', 'Right', 'Right', 'Right']
```

```
BELIEF MATRIX :

[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
Drone Position with 100% Certainity :  (14, 7)
Number of Moves Taken :  332
(base) zeeshanahsan@Zeeshans-MacBook-Pro Code %
```

QUESTION 4 –

Question 4 (15 Points): Write a program that outputs a a 19x19 reactor schematic that has the longest possible sequence of commands needed to locate the drone that you can find. Be clear in your writeup about how you are formulating the problem, and the algorithms you are using to find this reactor schematic.

To solve this problem, I thought that we need the most difficult/unique way of propagating probabilities through the grid. If there is a unique way to propagate probabilities, then the agent must make those moves using the drone only then it would be able to essentially 'collect' all the probabilities together and hence say with certainty about the location of the drone.

The design of the grid should be such that every free cell is reachable from every other cell, otherwise the agent can never collect all probabilities in one place and can't ever solve the problem. So, any free cell should not be completely walled off.

In my approach, I make a perfect grid in which every cell is reachable from every other cell and there is only 1 path from the entry to the exit of the grid. In this way the agent will have to make the exact moves to solve the grid and will take the longest sequence of steps to exactly spot the drone.

I have used randomized prim's algorithm to create the perfect grid.

The Algorithm is as follows:

1. Start with a grid which contains all walls.
2. Make all the walls as unvisited/ Add walls in unvisited dictionary.
3. Pick a wall and make the wall as an open cell, mark this as visited. Add the neighboring walls of this cells to a walls list.
4. While the walls list is not empty–
    a. Pick a random item(wall) from this list. If only one of the two cells that the wall divides are visited, then do the following:
        i. Make the wall a free cell and mark the unvisited cell as visited and part of the grid.
        ii. Add the neighbor walls of the current cell to the walls list.
    b. Remove the picked item(wall) from the walls list.

CODE FOR THE GRID GENERATION:

```python
import numpy as np
import random

def createGrid(rows,cols):
    #Initialize with all 1's denoting all walls
    grid = np.ones((rows,cols),dtype=(int))
    visited_dict = {}
    print(grid[0][0])
    unvisited = []
    walls = []

    #Start with a random cell
    x_random = int(random.random()*rows)
    y_random = int(random.random()*cols)

    #Make the values of all cells 3 -> Denoting all cells are unvisited in the beginning
    for i in range(rows):
        for j in range(cols):
            grid[i][j] = 3

    #Make sure we don't start at the edge/boundary of the grid
    if x_random == 0:
        x_random += 1
    if x_random == rows-1:
        x_random -= 1
    if y_random == 0:
        y_random += 1
    if y_random == cols-1:
        y_random -= 1
```

```python
    #Make this (x_random,y_random) as an open path
    grid[x_random][y_random] = 0
    #Add the surrounding cells to the walls list
    walls.append((x_random, y_random - 1))
    walls.append((x_random,y_random + 1))
    walls.append((x_random - 1, y_random))
    walls.append((x_random + 1, y_random))

    grid[x_random][y_random - 1] = 1
    grid[x_random][y_random + 1] = 1
    grid[x_random - 1][y_random] = 1
    grid[x_random + 1][y_random - 1] = 1
```

```python
    while(walls):
        #Pick a randoim wall
        r_wall = walls[int(random.random()*len(walls))-1]

        # Check if it is a left wall
        if (r_wall[1] != 0):
            if (grid[r_wall[0]][r_wall[1]-1] == 3 and grid[r_wall[0]][r_wall[1]+1] == 0):
                # Find number of surrounding cells
                s_cells = surroundingCells(grid,r_wall)

                if (s_cells < 2):
                    # Denote the new path
                    grid[r_wall[0]][r_wall[1]] = 0

                    # Mark the new walls
                    # Upper cell
                    if (r_wall[0] != 0):
                        if (grid[r_wall[0]-1][r_wall[1]] != 0):
                            grid[r_wall[0]-1][r_wall[1]] = 1
                        if ((r_wall[0]-1, r_wall[1]) not in walls):
                            walls.append((r_wall[0]-1, r_wall[1]))
```

```
                # Bottom cell
                if (r_wall[0] != rows-1):
                    if (grid[r_wall[0]+1][r_wall[1]] != 0):
                        grid[r_wall[0]+1][r_wall[1]] = 1
                    if ((r_wall[0]+1, r_wall[1]) not in walls):
                        walls.append((r_wall[0]+1, r_wall[1]))

                # Leftmost cell
                if (r_wall[1] != 0):
                    if (grid[r_wall[0]][r_wall[1]-1] != 0):
                        grid[r_wall[0]][r_wall[1]-1] = 1
                    if ((r_wall[0], r_wall[1]-1) not in walls):
                        walls.append((r_wall[0], r_wall[1]-1))

            # Delete wall
            for wall in walls:
                if (wall[0] == r_wall[0] and wall[1] == r_wall[1]):
                    walls.remove(wall)

            continue
```

```
            # Delete wall
            for wall in walls:
                if (wall[0] == r_wall[0] and wall[1] == r_wall[1]):
                    walls.remove(wall)

            continue
    # Check if it is an upper wall
    if (r_wall[0] != 0):
        if (grid[r_wall[0]-1][r_wall[1]] == 3 and grid[r_wall[0]+1][r_wall[1]] == 0):

            #Find number of surrounding cells
            s_cells = surroundingCells(grid,r_wall)

            if (s_cells < 2):
                # Denote the new path
                grid[r_wall[0]][r_wall[1]] = 0

                #Mark new walls
                # Upper cell
                if (r_wall[0] != 0):
                    if (grid[r_wall[0]-1][r_wall[1]] != 0):
                        grid[r_wall[0]-1][r_wall[1]] = 1
                    if ((r_wall[0]-1, r_wall[1]) not in walls):
                        walls.append((r_wall[0]-1, r_wall[1]))

                # Leftmost cell
                if (r_wall[1] != 0):
                    if (grid[r_wall[0]][r_wall[1]-1] != 0):
                        grid[r_wall[0]][r_wall[1]-1] = 1
                    if ((r_wall[0], r_wall[1]-1) not in walls):
                        walls.append((r_wall[0], r_wall[1]-1))

                # Rightmost cell
                if (r_wall[1] != cols-1):
                    if (grid[r_wall[0]][r_wall[1]+1] != 0):
                        grid[r_wall[0]][r_wall[1]+1] = 1
                    if ((r_wall[0], r_wall[1]+1) not in walls):
                        walls.append((r_wall[0], r_wall[1]+1))
```

```
    # Check the bottom wall
    if (r_wall[0] != rows-1):
        if (grid[r_wall[0]+1][r_wall[1]] == 3 and grid[r_wall[0]-1][r_wall[1]] == 0):

            s_cells = surroundingCells(grid,r_wall)
            if (s_cells < 2):
                # Denote the new path
                grid[r_wall[0]][r_wall[1]] = 0

                # Mark the new walls
                if (r_wall[0] != cols-1):
                    if (grid[r_wall[0]+1][r_wall[1]] != 0):
                        grid[r_wall[0]+1][r_wall[1]] = 1
                    if ((r_wall[0]+1, r_wall[1]) not in walls):
                        walls.append((r_wall[0]+1, r_wall[1]))

                if (r_wall[1] != 0):
                    if (grid[r_wall[0]][r_wall[1]-1] != 0):
                        grid[r_wall[0]][r_wall[1]-1] = 1
                    if ((r_wall[0], r_wall[1]-1) not in walls):
                        walls.append((r_wall[0], r_wall[1]-1))
                    if (r_wall[1] != cols-1):
                        if (grid[r_wall[0]][r_wall[1]+1] != 0):
                            grid[r_wall[0]][r_wall[1]+1] = 1
                        if ((r_wall[0], r_wall[1]+1) not in walls):
                            walls.append((r_wall[0], r_wall[1]+1))

                # Delete wall
                for wall in walls:
                    if (wall[0] == r_wall[0] and wall[1] == r_wall[1]):
                        walls.remove(wall)

                continue
    # Check the right wall
    if (r_wall[1] != cols-1):
        if (grid[r_wall[0]][r_wall[1]+1] == 3 and grid[r_wall[0]][r_wall[1]-1] == 0):

            s_cells = surroundingCells(grid,r_wall)
            if (s_cells < 2):
                # Denote the new path
                grid[r_wall[0]][r_wall[1]] = 0

                # Mark the new walls
                if (r_wall[1] != cols-1):
                    if (grid[r_wall[0]][r_wall[1]+1] != 0):
                        grid[r_wall[0]][r_wall[1]+1] = 1
                    if ((r_wall[0], r_wall[1]+1) not in walls):
                        walls.append((r_wall[0], r_wall[1]+1))

                if (r_wall[0] != rows-1):
                    if (grid[r_wall[0]+1][r_wall[1]] != 0):
                        grid[r_wall[0]+1][r_wall[1]] = 1
                    if ((r_wall[0]+1, r_wall[1]) not in walls):
                        walls.append((r_wall[0]+1, r_wall[1]))

                if (r_wall[0] != 0):
                    if (grid[r_wall[0]-1][r_wall[1]] != 0):
                        grid[r_wall[0]-1][r_wall[1]] = 1
                    if ((r_wall[0]-1, r_wall[1]) not in walls):
                        walls.append((r_wall[0]-1, r_wall[1]))

                # Delete wall
                for wall in walls:
                    if (wall[0] == r_wall[0] and wall[1] == r_wall[1]):
                        walls.remove(wall)

                continue
```

```
        # Delete the wall from the list
        for wall in walls:
            if (wall[0] == r_wall[0] and wall[1] == r_wall[1]):
                walls.remove(wall)

    # Mark the unvisited cells as walls
    for i in range(0, rows):
        for j in range(0, cols):
            if(grid[i][j] == 3):
                grid[i][j] = 1

    # Set start and end
    for i in range(0, cols):
        if (grid[1][i] == 0):
            grid[0][i] = 0
            break

    for i in range(cols - 1, 0, -1):
        if(grid[rows - 2][i] == 0):
            grid[rows-1][i] = 0
            break

    return grid
```

```
#Find the number of surrounding cells
def surroundingCells(grid,r_wall):
    count = 0
    #If Up Cell is free
    if (grid[r_wall[0]-1][r_wall[1]] == 0):
        count += 1
    #If down cell is free
    if(grid[r_wall[0]+1][r_wall[1]] == 0):
        count +=1
    #If left cell is free
    if (grid[r_wall[0]][r_wall[1]-1] == 0):
        count +=1
    #If right cell is free
    if (grid[r_wall[0]][r_wall[1]+1] == 0):
        count += 1

    return count
```

```
def create_file_from_grid(grid):

    with open('/Users/zeeshanahsan/Code/CS520Final_exam/ZeeGrid.txt', 'w') as f:
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                #If wall then write 'X'
                if(grid[i][j] == 1):
                    f.write('X')
                #If free cell then write '_'
                elif(grid[i][j] == 0):
                    f.write('_')
            #Go to new line
            f.write('\n')



#Driver code

grid = createGrid(19,19)
print(grid)
create_file_from_grid(grid)
```

TERMINAL OUTPUT OF THE GRID:

```
[[1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
 [1 0 0 0 0 0 1 0 1 0 0 1 0 1 1 0 0 0 1]
 [1 1 1 0 1 1 1 0 1 1 0 1 0 1 0 0 1 0 1]
 [1 1 0 0 0 1 1 0 0 1 0 0 0 0 0 1 1 1 1]
 [1 0 0 1 0 0 0 0 1 1 0 1 1 1 1 1 0 1 1]
 [1 0 1 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 1]
 [1 0 0 1 1 1 1 0 1 0 1 0 1 1 0 0 0 1 1]
 [1 1 0 0 0 1 1 0 1 0 1 0 1 1 0 1 0 0 1]
 [1 1 1 1 0 0 1 0 1 0 1 0 0 1 0 1 0 1 1]
 [1 0 0 0 0 1 1 0 1 0 1 1 0 1 1 1 0 1 1]
 [1 0 1 0 1 1 1 0 1 0 1 1 0 0 0 1 0 0 1]
 [1 0 1 0 1 1 0 0 1 0 1 1 0 1 1 1 1 1 1]
 [1 0 1 0 1 0 0 1 1 0 1 1 0 0 0 0 0 0 1]
 [1 1 1 1 1 0 1 1 0 0 0 1 1 0 1 0 1 1 1]
 [1 1 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1 1 1]
 [1 0 0 1 1 0 1 1 1 1 1 1 1 0 1 0 0 0 1]
 [1 0 1 1 0 0 0 0 0 0 1 1 0 0 1 0 1 1 1]
 [1 0 0 1 1 0 1 0 1 0 1 1 0 1 1 0 0 0 1]
 [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1]]
```

Here 1's are walls and 0's are free cells.

ACTUAL GRID:

X_XXXXXXXXXXXXXXXX
X_____X_X__X_XX___X
XXX_XXX_XX_X_X__X_X
XX___XX__X_____XXXX
X__X____XX_XXXXX_XX
X_XXX_X________X__X
X__XXXX_X_X_XX___XX
XX___XX_X_X_XX_X__X
XXXX__X_X_X__X_X_XX
X____XX_X_XX_XXX_XX
X_X_XXX_X_XX___X__X
X_X_XX__X_XX_XXXXXX
X_X_X__XX_XX_____X
XXXXX_XX___XX_X_XXX
XX_____XX_XX__X_XXX
X__XX_XXXXXXX_X___X
X_XX_____XX__X_XXX
X__XX_X_X_XX_XX___X
XXXXXXXXXXXXXXXXX_X

STEPS TAKEN TO SOLVE THIS GRID: 354
Drone Position with 100% certainty after 354 moves: (17,12)

MOVES REQUIRED TO FIND THE DRONE:
['Down', 'Right', 'Right', 'Right', 'Right', 'Left', 'Left', 'Down', 'Down', 'Right', 'Down', 'Right', 'Right', 'Right', 'Down', 'Right', 'Right', 'Right', 'Up', 'Up', 'Up', 'Up', 'Down', 'Down', 'Right', 'Right', 'Right',

'Right', 'Up', 'Right', 'Up', 'Right', 'Right', 'Left', 'Left', 'Down', 'Left', 'Down', 'Left', 'Left', 'Left', 'Left', 'Down', 'Down', 'Left', 'Left', 'Left', 'Up', 'Left', 'Left', 'Left', 'Up', 'Left', 'Up', 'Down', 'Left', 'Down', 'Left', 'Down', 'Down', 'Up', 'Up', 'Right', 'Up', 'Right', 'Right', 'Down', 'Right', 'Right', 'Right', 'Down', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Left', 'Up', 'Left', 'Left', 'Up', 'Left', 'Left', 'Left', 'Down', 'Down', 'Down', 'Up', 'Up', 'Up', 'Right', 'Right', 'Right', 'Down', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Left', 'Up', 'Left', 'Left', 'Up', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Up', 'Left', 'Left', 'Left', 'Up', 'Left', 'Left', 'Down', 'Left', 'Down', 'Down', 'Right', 'Down', 'Right', 'Right', 'Down', 'Down', 'Up', 'Up', 'Up', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Up', 'Left', 'Left', 'Left', 'Up', 'Left', 'Left', 'Down', 'Left', 'Down', 'Down', 'Right', 'Down', 'Right', 'Right', 'Down', 'Down', 'Up', 'Up', 'Left', 'Left', 'Up', 'Left', 'Up', 'Up', 'Right', 'Up', 'Right', 'Right', 'Down', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Right', 'Down', 'Down', 'Down', 'Up', 'Up', 'Left', 'Left', 'Up', 'Left', 'Up', 'Up', 'Right', 'Up', 'Right', 'Right', 'Down', 'Right', 'Right', 'Right', 'Down', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Right', 'Down', 'Down', 'Right', 'Right', 'Left', 'Down', 'Left', 'Left', 'Up', 'Left', 'Left', 'Left', 'Down', 'Down', 'Down', 'Right', 'Down', 'Down', 'Right', 'Right', 'Left', 'Left', 'Up', 'Up', 'Left', 'Up', 'Up', 'Up', 'Up', 'Right', 'Right', 'Right', 'Down', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Right', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Right', 'Left', 'Up', 'Up', 'Up', 'Up', 'Left', 'Left', 'Up', 'Left', 'Left', 'Left', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Left', 'Down', 'Left', 'Down', 'Down', 'Left', 'Left', 'Left', 'Down', 'Left', 'Down', 'Down', 'Up', 'Up', 'Right', 'Up', 'Right', 'Right', 'Right', 'Up', 'Up', 'Right', 'Up', 'Right', 'Up', 'Up', 'Up', 'Up', 'Up', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Right', 'Down', 'Down', 'Down', 'Down', 'Right', 'Down', 'Down', 'Down', 'Left', 'Down', 'Left', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Left', 'Down']

TERMINAL OUTPUT:

```
Moves :
['Down', 'Right', 'Right', 'Right', 'Right', 'Left', 'Left', 'Down', 'Down', 'Right', 'Down', 'Right', 'Right', 'Right', 'Down', 'Rig
ht', 'Right', 'Right', 'Up', 'Up', 'Up', 'Up', 'Down', 'Down', 'Right', 'Right', 'Right', 'Right', 'Up', 'Right', 'Up', 'Right', 'Rig
ht', 'Left', 'Left', 'Down', 'Left', 'Down', 'Left', 'Left', 'Left', 'Left', 'Down', 'Down', 'Left', 'Left', 'Left', 'Up', 'Left', 'L
eft', 'Left', 'Up', 'Left', 'Up', 'Down', 'Left', 'Down', 'Left', 'Down', 'Down', 'Up', 'Up', 'Right', 'Up', 'Right', 'Right', 'Down'
, 'Right', 'Right', 'Right', 'Down', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Left', 'Up', 'Left', 'Left', 'Up',
'Left', 'Left', 'Left', 'Down', 'Down', 'Down', 'Up', 'Up', 'Up', 'Right', 'Right', 'Right', 'Down', 'Right', 'Right', 'Down', 'Down'
, 'Down', 'Down', 'Left', 'Up', 'Left', 'Left', 'Up', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Up', 'Left', 'Left', '
Left', 'Up', 'Left', 'Left', 'Down', 'Left', 'Down', 'Down', 'Right', 'Down', 'Right', 'Right', 'Down', 'Down', 'Up', 'Up', 'Up', 'Le
ft', 'Left', 'Left', 'Left', 'Left', 'Left', 'Left', 'Up', 'Left', 'Left', 'Left', 'Up', 'Left', 'Left', 'Down', 'Left', 'Down', 'Dow
n', 'Right', 'Down', 'Right', 'Right', 'Down', 'Down', 'Up', 'Up', 'Left', 'Left', 'Up', 'Left', 'Up', 'Up', 'Right', 'Up', 'Right',
 'Right', 'Down', 'Right', 'Right', 'Right', 'Down', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Right', 'Down', 'Dow
n', 'Down', 'Up', 'Up', 'Left', 'Left', 'Up', 'Left', 'Up', 'Up', 'Right', 'Up', 'Right', 'Right', 'Down', 'Right', 'Right', 'Right',
 'Down', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Right', 'Down', 'Down', 'Right', 'Right', 'Left', 'Down', 'Left
', 'Left', 'Up', 'Left', 'Left', 'Left', 'Down', 'Down', 'Down', 'Right', 'Down', 'Down', 'Right', 'Right', 'Left', 'Left', 'Up', 'Up
', 'Left', 'Up', 'Up', 'Up', 'Right', 'Right', 'Right', 'Down', 'Right', 'Right', 'Down', 'Down', 'Down', 'Down', 'Right', 'Left', 'D
own', 'Down', 'Down', 'Down', 'Down', 'Right', 'Left', 'Up', 'Up', 'Up', 'Up', 'Left', 'Left', 'Up', 'Left', 'Left', 'Left', 'Left',
 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up', 'Up',
 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Down', 'Down', 'Left', 'Down', 'Left', 'Down', 'Down', 'Left', 'Left', 'Left', 'Dow
n', 'Left', 'Down', 'Down', 'Up', 'Up', 'Right', 'Up', 'Right', 'Right', 'Right', 'Up', 'Up', 'Right', 'Up', 'Right', 'Up', 'Up', 'Up
', 'Up', 'Up', 'Right', 'Right', 'Right', 'Right', 'Down', 'Down', 'Down', 'Right', 'Down', 'Down', 'Down', 'Down', 'Right', 'Down',
'Down', 'Down', 'Left', 'Down', 'Left', 'Left', 'Left', 'Down', 'Down', 'Down', 'Down', 'Left', 'Down']
```

BELIEF MATRIX AFTER DRONE FOUND:

```
BELIEF MATRIX :

[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
Drone Position with 100% Certainity :  (17, 12)
Number of Moves Taken :  354
(base) zeeshanahsan@Zeeshans-MacBook-Pro Code %
```

THE OUTPUT OF THIS CODE IS NAMED ZeeGrid_Longest.txt