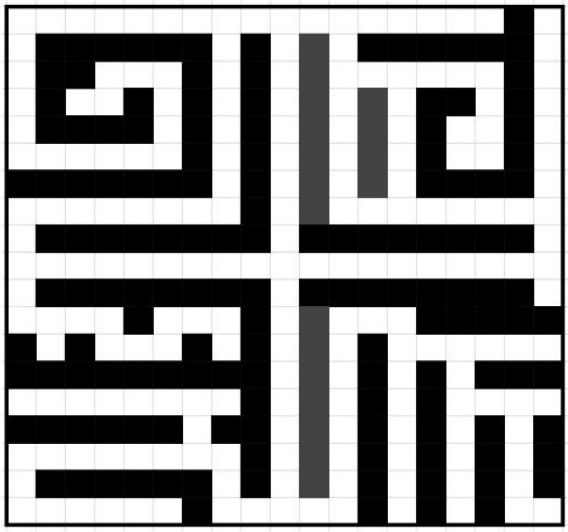
Final Exam Q1: Finding Your Way

This question asks to accurately determine the position of a drone in the nuclear reactor of a power station.
We are unaware where the drone is initially. We are only allowed to instruct the drone to move in the directions right, left, up and down. If the drone encounters a wall while moving, it chooses to stay there. We are supposed to find the optimal sequence of steps to accurately determine where the drone is.
Image of the Reactor given in the question -

Figure 1: TH23-SA74-VERW Reactor Schematic



Question 1:
Before you do anything, what is the probability that the drone is in the top left corner? Why?
As mentioned in the question, the drone can only occupy the cells which are marked in white.
The drone cannot pass through the walls.
Before any move is made, the belief we have for the exact position needs to be divided equally amongst all the cells which are not blocked by walls.
For the diagram attached, the probability of the top left cell having the drone is –
1 / Number of white cells
=1/199
=0.005025125628140704

Question 2:
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.
The various ways the probabilities of the cells behave while moving down –

- If the cell has a wall/boundary above it and a free cell below it, the probability of this cell after one move down becomes 0.
- If the cell has no wall/boundary above or below it, the probability of this cell becomes the probability of the cell above it.
- If the cell has a wall/boundary below it and a free cell above it, the probability of this cell is the sum of the probabilities of the cell above it and the cell itself.

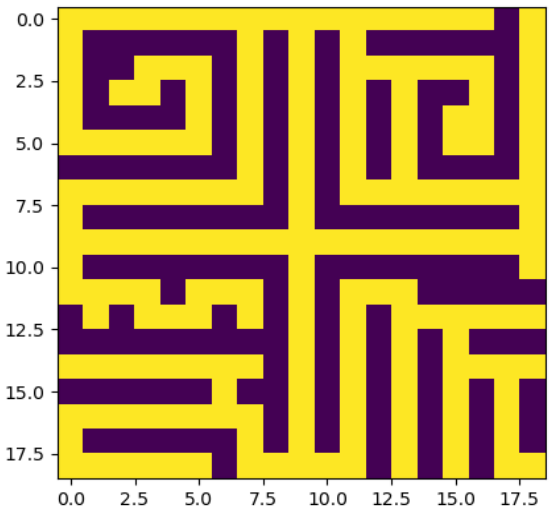
Most Likely cells –

- Cells with a wall below it and no wall above it.

Least Likely cells –

- Cells with a wall above it and no walls below it.

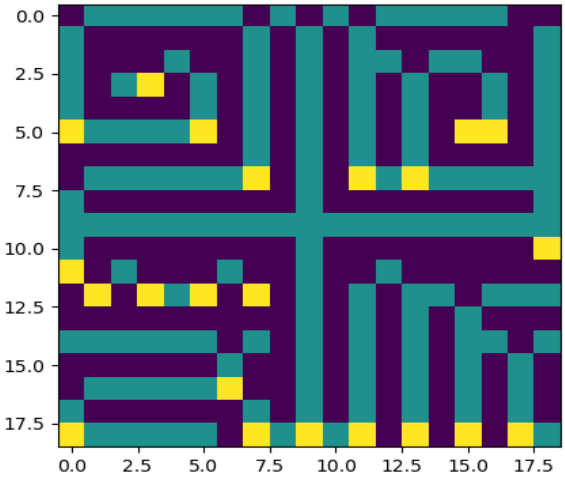
Before Down Movement –



The different colors show the different probability values.

- Purples - 0
- Yellow - 0.005025125628140704. (1/199)

After Down Movement –



- Purples – 0 (Least likely)
- Yellow – 0.01005025 (Most Likely)
- Green – 0.00502513 (Likelihood everywhere else)

Question 3:
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?

Start State – Probability table where the belief is equally split amongst the free cells.
Goal State – Probability table where the belief in one cell is 1 and rest are 0.
Action Set – [Right, Left, Up, Down]
Cost – As we are trying to find the optimal number of steps to reach the goal state, the cost of each action from the action set is 1.

Reading the text file and initialization -
To solve this problem, I have converted the given reactor schematic into a 2D matrix where a '0' represents a path and a '1' represents a wall.
A separate matrix called 'probab_matrix' is maintained to store the probability of each cell containing the drone.

The probab_matrix helps us keep track of the probabilities of each cell as we execute operations.

```
with open('schema.txt') as f:                                  # Reading the Text file
    lines = f.readlines()
rows=len(lines)
cols=len(lines[0])-1
print(rows,cols)
free=0
wall=0
matrix = np.zeros((rows,cols),dtype=int)
prob_matrix = np.zeros((rows,cols),dtype=float)
for i in range(rows):                                           # Converting the text file into a 2d matrix v
    for j in range(cols):
        if(lines[i][j]=='_'):
            matrix[i][j]=0
            free=free+1
        else:
            matrix[i][j]=1
            wall=wall+1

for i in range(rows):
    for j in range(cols):
        if(matrix[i][j]==0):
            prob_matrix[i][j]=(1/free)                         # Defining the initial Probabilities for the gr
```

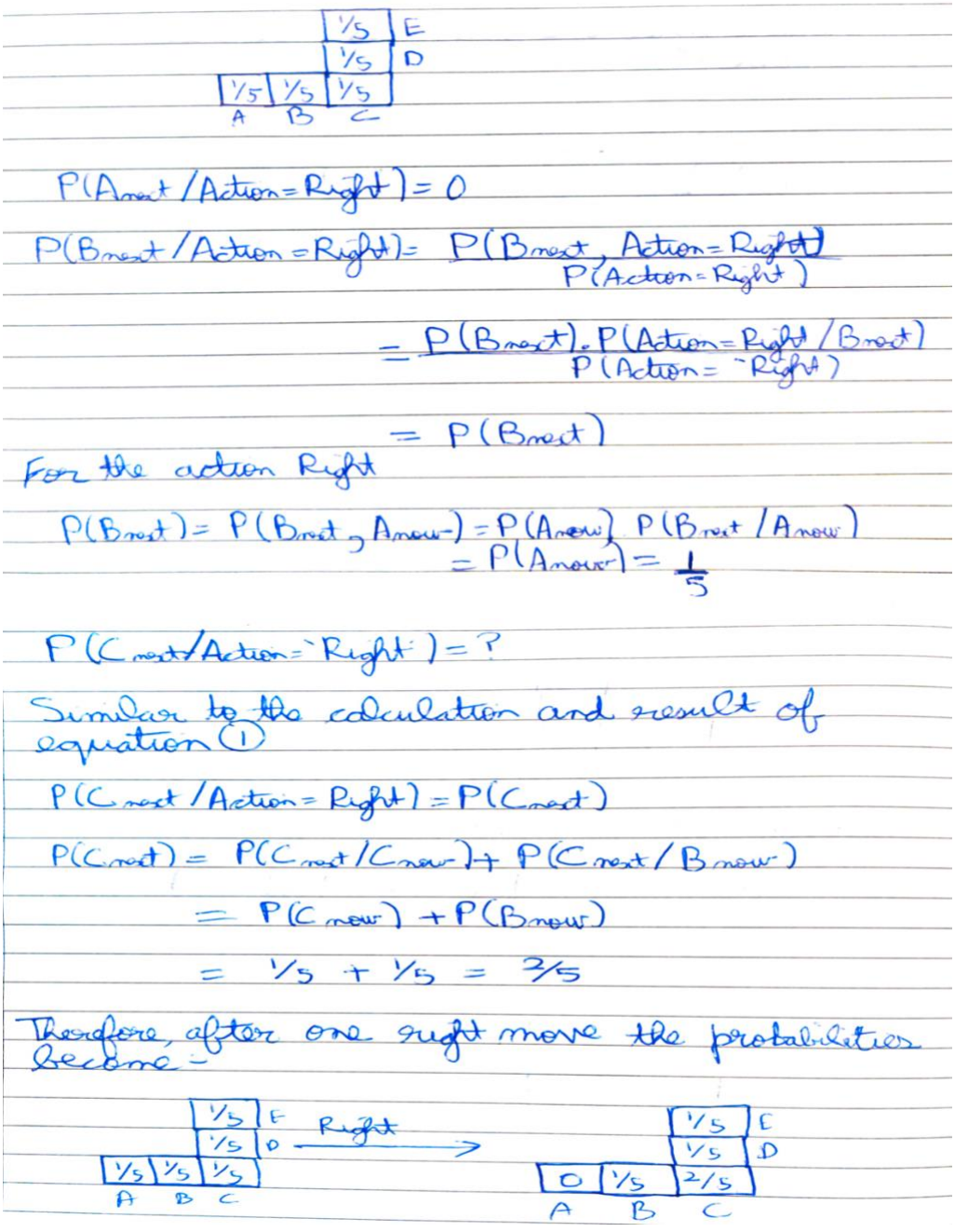
As we are only allowed to run operations which makes the drone move while receiving no information as feedback, we need to implement how the probabilities change in the environment after each move –

The probability of any given cell having the drone after any move is made is dependent on the adjacent cells.

Below I have attached a screenshot of how the probabilities change after making one move to the right.
The grid I have taken for this example has only 5 cells (A, B, C, D and E), with no walls in between them.

Initially every cell has a probability of (1/5) of having the drone in them.

After moving right, the probabilities for each cell can be calculated as shown below.



The probabilities behave similarly for every other action in the action set.

The general rules followed while making these actions based on the calculations are given below.

Probability changes for each cell given an action –

Down –

- If the cell has a wall/boundary above it and a free cell below it, the probability of this cell after one move down becomes 0.
- If the cell has no wall/boundary above or below it, the probability of this cell becomes the probability of the cell above it.
- If the cell has a wall/boundary below it and a free cell above it, the probability of this cell is the sum of the probabilities of the cell above it and the cell itself.

Up –

- If the cell has a wall/boundary below it and a free cell above it, the probability of this cell after one move up becomes 0.
- If the cell has no wall/boundary above or below it, the probability of this cell becomes the probability of the cell below it.
- If the cell has a wall/boundary above it and a free cell below it, the probability of this cell is the sum of the probabilities of the cell below it and the cell itself.

Left –

- If the cell has a wall/boundary to the right and a free cell to the left, the probability of this cell after one move left becomes 0.
- If the cell has no wall/boundary to the left or right, the probability of this cell becomes the probability of the cell to the right.
- If the cell has a wall/boundary to the left and a free cell to the right, the probability of this cell is the sum of the probabilities of the cell to the right and the cell itself.

Right –

- If the cell has a wall/boundary to the left and a free cell to the right, the probability of this cell after one move right becomes 0.
- If the cell has no wall/boundary to the left or right, the probability of this cell becomes the probability of the cell to the left.
- If the cell has a wall/boundary to the right and a free cell to the left, the probability of this cell is the sum of the probabilities of the cell to the right and the cell itself.

Right Implementation -

```
def Move_Right(matrix,temp_prob_matrix):
    copy_of_beliefs = np.copy(temp_prob_matrix)
    rows = len(matrix)
    cols = len(matrix[0])

    for i in range(rows):
        for j in range(cols):
            if(j==cols-1):                                                  # Edge Case
                if(matrix[i][j]!=1 and matrix[i][j-1]!=1):                  # Probability calculation for
                    temp_prob_matrix[i][j]=copy_of_beliefs[i][j]+copy_of_beliefs[i][j-1]

            if(j-1>=0 and j!=cols-1):                                       # Edge Case
                if(matrix[i][j]!=1 and matrix[i][j+1]!=1 and matrix[i][j-1]!=1):      # Probability calculation for c
                    temp_prob_matrix[i][j]=copy_of_beliefs[i][j-1]

            if(j-1>=0 and j!=cols-1):                                       # Edge Case
                if(matrix[i][j]!=1 and matrix[i][j+1]==1 and matrix[i][j-1]!=1):      # Wall to the right
                    temp_prob_matrix[i][j]=copy_of_beliefs[i][j]+copy_of_beliefs[i][j-1]

            if(j-1>=0 and j!=cols-1):                                       # Edge Case
                if(matrix[i][j]!=1 and matrix[i][j-1]==1):                  # Wall to the left
                    temp_prob_matrix[i][j]=0

            if(j==0):                                                       # First column
                if(matrix[i][j]!=1 and matrix[i][j+1]!=1):
                    temp_prob_matrix[i][j]=0

            if(j!=0 and j+1<=cols-1 and j-1>=0):                            # Edge Case
                if(matrix[i][j]!=1 and matrix[i][j-1]==1 and matrix[i][j+1]==1):      # Wall on the right and left
                    temp_prob_matrix[i][j]=copy_of_beliefs[i][j]
    return temp_prob_matrix
```

Left Implementation -

```
def Move_Left(matrix,temp_prob_matrix):
    copy_of_beliefs = np.copy(temp_prob_matrix)
    rows = len(matrix)
    cols = len(matrix[0])

    for i in range(rows):
        for j in range(cols):
            if(j == 0):                                                     # Probability calculation for le
                if(matrix[i][j] != 1 and matrix[i][j+1] != 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i][j+1]

            if(j+1 <= cols-1 and j != 0):                                   # Probability calculation for ce
                if(matrix[i][j] !=1 and matrix[i][j+1] != 1 and matrix[i][j-1] != 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i][j+1]

            if(j+1 <= cols-1 and j != 0):                                   # Wall to the left and no wall t
                if(matrix[i][j] !=1 and matrix[i][j+1] != 1 and matrix[i][j-1] == 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i][j+1]

            if(j+1 <= cols-1 and j != 0):                                   # Wall to the right and no wall
                if(matrix[i][j] !=1 and matrix[i][j+1] == 1):
                    temp_prob_matrix[i][j] = 0

            if(j == (cols-1)):
                if(matrix[i][j] != 1 and matrix[i][j-1] != 1):
                    temp_prob_matrix[i][j] = 0

            if(j != 0 and j+1 <= cols - 1 and j-1 >=0):                     # Wall to the left and right
                if(matrix[i][j] != 1 and matrix[i][j-1] == 1 and matrix[i][j+1] == 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i][j]

    return temp_prob_matrix
```

Up implementation –

```
def Move_Up(matrix,temp_prob_matrix):
    copy_of_beliefs = np.copy(temp_prob_matrix)
    rows = len(matrix)
    cols = len(matrix[0])

    for j in range(cols):
        for i in range(rows):
            if(i==0):                                                                   # Probability of the top row
                if(matrix[i][j] != 1 and matrix[i+1][j] != 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i+1][j]

            if(i+1 <= rows - 1 and i != 0):                                             # Probability calculation for cell
                if(matrix[i][j] != 1 and matrix[i+1][j] != 1 and matrix[i-1][j] != 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i+1][j]

            if(i+1 <= rows - 1 and i != 0):                                             # Wall above and no wall below
                if(matrix[i][j] != 1 and matrix[i+1][j] != 1 and matrix[i-1][j] == 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i+1][j]

            if(i+1 <= rows-1 and i != 0):                                               # Wall below
                if(matrix[i][j] != 1 and matrix[i+1][j] == 1):
                    temp_prob_matrix[i][j] = 0

            if(i == rows - 1):
                if(matrix[i][j] != 1 and matrix[i-1][j] != 1):
                    temp_prob_matrix[i][j] = 0

            if(i != 0 and i+1 <= rows - 1 and i-1 >=0):                                  # No walls above or below
                if(matrix[i][j] != 1 and matrix[i+1][j] == 1 and matrix[i-1][j] == 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i][j]
    return temp_prob_matrix
```

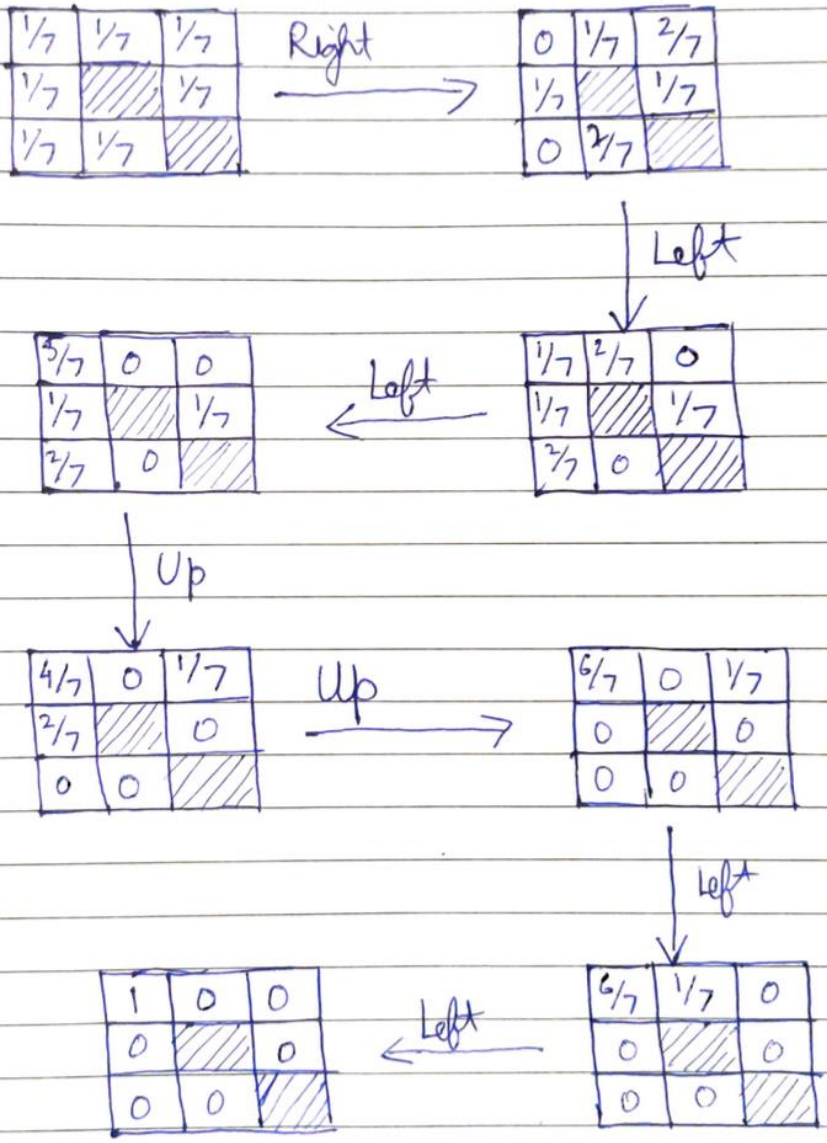
Down implementation –

```
def Move_Down(matrix,temp_prob_matrix):
    copy_of_beliefs = np.copy(temp_prob_matrix)
    rows = len(matrix)
    cols = len(matrix[0])

    for j in range(cols):
        for i in range(rows):
            if(i== rows -1):                                                            # Probability of the bottom
                if(matrix[i][j] != 1 and matrix[i-1][j] != 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i-1][j]

            if(i+1 <= rows - 1 and i != 0):                                             # Probability calculation fo
                if(matrix[i][j] != 1 and matrix[i+1][j] != 1 and matrix[i-1][j] != 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i-1][j]

            if(i+1 <= rows - 1 and i != 0):                                             # Wall below and no wall abo
                if(matrix[i][j] != 1 and matrix[i+1][j] == 1 and matrix[i-1][j] != 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i][j] + copy_of_beliefs[i-1][j]

            if(i-1 >= 0 and i != rows - 1):                                             # Wall above
                if(matrix[i][j] != 1 and matrix[i-1][j] == 1):
                    temp_prob_matrix[i][j] = 0

            if(i == 0):
                if(matrix[i][j] != 1 and matrix[i+1][j] != 1):
                    temp_prob_matrix[i][j] = 0

            if(i != 0 and i+1 <= rows - 1 and i-1 >=0):                                  # Wall above and below
                if(matrix[i][j] != 1 and matrix[i+1][j] == 1 and matrix[i-1][j] == 1):
                    temp_prob_matrix[i][j] = copy_of_beliefs[i][j]
    return temp_prob_matrix
```

Below I have attached a screenshot of a dry run in a 3x3 grid with 2 walls.

Grid Dimension = 3×3
Wall positions = (2,2) (3,3).

Initial Probability of each cell having the drone = (1/7)



Final Drone position = (1, 1)
Command Sequence = [Right, Left, Left, Up, Up, Left, Left]

Finding the optimal sequence of moves which helps in finding the drone–

1. Initially I have iterated over the action space and implemented each action until all probabilities accumulate next to a wall or a boundary. For example, if the chosen action is 'Right', I execute it until all the probabilities are in a cell adjacent to a wall or boundary. This is then repeated for every other action in the action set.

   'Right' and 'Left' actions from the action set are implemented either until all cells with a non-zero probability are adjacent to a boundary or wall to its right or left respectively.

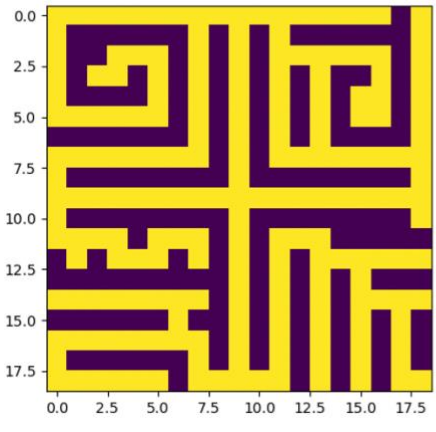
   'Up' and 'Down' actions from the action set are implemented either until all cells with a non-zero probability are adjacent to a boundary or wall to its top or left respectively.

```
for action in Action_Set:                                    # Running each section until most
    print('Action = ',action)
    if(action=='Right'):                                     # Right
        for i in range(cols):
            wall_check=0
            prob_matrix=Move_Right(matrix,prob_matrix)
            for r in range(rows):
                for c in range(cols):
                    if(prob_matrix[r][c]!=0):
                        if(c+1<cols and matrix[r][c+1]==0):
                            wall_check=1
            list_of_actions.append('Right')
            # plt.imshow(prob_matrix)
            # plt.show()
            number_of_moves=number_of_moves+1
            if(wall_check==0):                               # Breaking if all probabilities a
                break
    if(action=='Left'):                                      # Left
        for i in range(cols):
            wall_check=0
            prob_matrix=Move_Left(matrix,prob_matrix)
            for r in range(rows):
                for c in range(cols):
                    if(prob_matrix[r][c]!=0):
                        if(c-1>=0 and matrix[r][c-1]==0):
                            wall_check=1
            list_of_actions.append('Left')
            # plt.imshow(prob_matrix)
            # plt.show()
            number_of_moves=number_of_moves+1
            if(wall_check==0):
                break
```

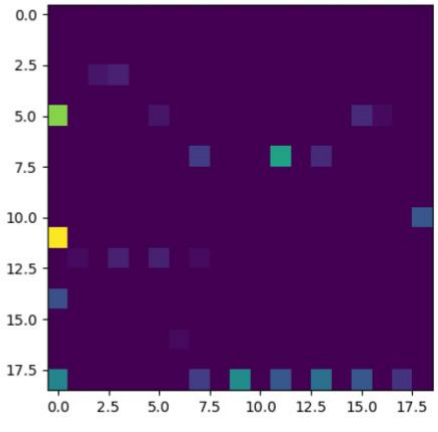
```
if(action=='Up'):                                                    # Up
    for i in range(rows):
        wall_check=0
        prob_matrix=Move_Up(matrix,prob_matrix)
        for r in range(rows):
            for c in range(cols):
                if(prob_matrix[r][c]!=0):
                    if(r-1>=0 and matrix[r-1][c]==0):
                        wall_check=1
        list_of_actions.append('Up')
        # plt.imshow(prob_matrix)
        # plt.show()
        number_of_moves=number_of_moves+1
        if(wall_check==0):
            break
if(action=='Down'):                                                  # Down
    for i in range(rows):
        wall_check=0
        prob_matrix=Move_Down(matrix,prob_matrix)
        for r in range(rows):
            for c in range(cols):
                if(prob_matrix[r][c]!=0):
                    if(r+1<rows and matrix[r+1][c]==0):
                        wall_check=1
        list_of_actions.append('Down')
        # plt.imshow(prob_matrix)
        # plt.show()
        number_of_moves=number_of_moves+1
        if(wall_check==0):
```

The given schematic before and after implementing this logic -

Before



After



Before -
Purple = 0
Yellow = (1/199)

After –
Dark Purple = 0
Every other color represents a value of probability
As shown in the screenshots attached above, after running through the action set, each non-zero probability is adjacent either to a wall or a boundary while all other positions now have a probability of 0.

2. Once all the probabilities are adjacent to a wall/boundary, our aim needs to be to move the probabilities towards each other to have a cell with probability 1 and the rest 0.

Algorithm used –

Repeat until probability matrix has a cell with value 1:

- Iterate over all values of the probability matrix and store the indexes where the value is non-zero in a list ‘vals’.

```
vals=[]
for r in range(rows):
    for c in range(cols):
        if(prob_matrix[r][c]!=0):
            vals.append((r,c))
if(len(vals)==1):
    break
```

- Calculate the distances between all points in ‘vals’ and store it in the dictionary ‘distances’ where the key is the source/destination index and the value is the shortest distance between them.

- Sort the dictionary ‘distances’ in descending order of its values. This helps us know the 2 points in the grid which are closest to each other while having a non-zero probability of having the drone.

```
distances={}
for s in range(len(vals)):
    source=vals[s]
    for d in range(len(vals)):
        destination=vals[d]
        if(source==destination):
            continue
        path=astar_original(matrix,source,destination)
        distances[source,destination]=len(path)
sorted_values = sorted(distances.values())
sorted_dist={}
for i in sorted_values:
    for k in distances.keys():
        if distances[k] == i:
            sorted_dist[k] = distances[k]
```

- The distances and the path between 2 points are calculated using the A* algorithm.

A* algorithm – The A-star algorithm is a smart algorithm which is guided by a certain heuristic. At every cell, the A-star algorithm evaluates a value 'f(n)'. This value is then used as a weight and compared to the other cell's values to govern the movement.

For any cell – f(n) = g(n) + h(n)
Where,
g(n) is the cost of the path from the start node to the node 'n'. h(n) is the cost of the cheapest path from the node 'n' to the goal.

To calculate 'h(n)' I have written a function called heuristic_evaluation which takes the cell's row and column index as the input. It then returns the Manhattan distance between the cell that was passed and the destination cell.
I have chosen to use the Manhattan distance over the Euclidean distance as movement is only allowed to move in the cardinal directions (Up, down, left and right).

```
#A-Star heuristic
def heuristic_evaluation(cell1,cell2):
    x1,y1=cell1
    x2,y2=cell2
    return abs(x1-x2) + abs(y1-y2)
```

Pseudocode of A-star algorithm –
Pq = Priority_queue
g_score = {0 for the start cell, ‘inf’ -> for all other cells }
f_score = {heuristic_evaluation(start, destination) for the start cell, ‘inf’ ->

for all other cells} pq.put -> (f_score, heuristic_evaluation(start, destination),

start)

```
while(pq!=empty):
        current_cell=pq.get(2)
        for every direction(Up, down, left
                and right): t_g_score =
                g_score(current_cell+1)
                t_f_score=temp_g_score+ heuristic_evaluation(child_cell,
                destination) if t_f_score<f_score(child_cell):
                        g_score(child_cell)=t_
                        g_score
                        f_score(child_cell)=t_f
                        _score
                pq.put(f_score(child_cell),
                heuristic_evaluation(child_cell,destination), child_cell)
```

- Store the dictionary key with the lowest value (Distance between the elements in the key) in a variable ‘point’.
- Run the A-Star algorithm between the cells in ‘point’ to generate a path.
- Instruct the drone to move in the direction of the next point in this generated path. Add this direction to ‘list_of_actions’ and add 1 to ‘number_of_moves’.

```
point=list(sorted_dist.keys())[0]
path = astar_original(matrix,point[0],point[1])
a=point[0]
b=point[1]
next_point=path[1]
if(a[0]==next_point[0] and a[1]+1==next_point[1]):
    print('Goes Right')
    prob_matrix=Move_Right(matrix,prob_matrix)
    list_of_actions.append('Right')
    number_of_moves=number_of_moves+1
elif(a[0]==next_point[0] and a[1]-1==next_point[1]):
    print('Goes Left')
    prob_matrix=Move_Left(matrix,prob_matrix)
    list_of_actions.append('Left')
    number_of_moves=number_of_moves+1
elif(a[0]+1==next_point[0] and a[1]==next_point[1]):
    print('Goes Down')
    prob_matrix=Move_Down(matrix,prob_matrix)
    list_of_actions.append('Down')
    number_of_moves=number_of_moves+1
elif(a[0]-1==next_point[0] and a[1]==next_point[1]):
    print('Goes Up')
    prob_matrix=Move_Up(matrix,prob_matrix)
    list_of_actions.append('Up')
    number_of_moves=number_of_moves+1
```

- Break the loop if the probability matrix has a cell with value 1.

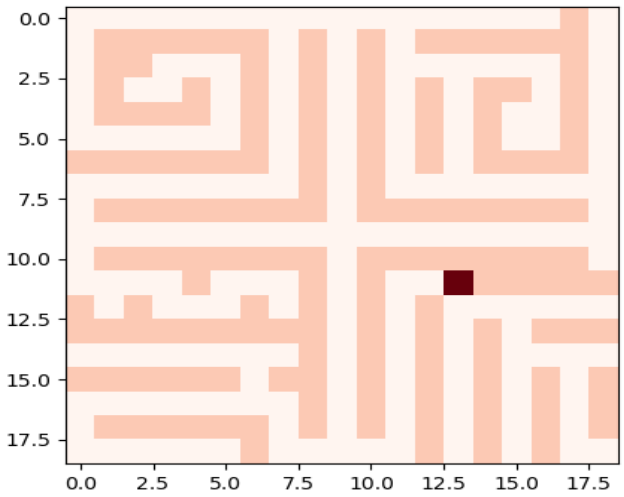
Results –

Number of moves taken to determine where the drone is = 290

Sequence of moves –

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

Final position of the drone – position [11,13]



The algorithm I have implemented does not always return the shortest sequence of moves to locate the drone. Even though I have implemented an A* algorithm to move the probabilities to one location, it is possible that while moving one probability to the destination cell, the destination moves itself.

Question 4:
Write a program that outputs 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.

The brute force way of finding the reactor schema that has the longest sequence of commands needed is unfeasible as the number of grids it would have to try and run the algorithm on would be too high.
The way I planned on proceeding is to place walls on the grid using a utility for each position I planned to put a wall on.
I would first divide the grid into 2.
In one of the grids, I would place only a small number of walls and in the other grid I would place the walls based on a utility function.
The utility of a particular cell would be high if it is able to trap a free cell with only one way in and out. This would mean that the drone can only really reach this cell through one other cell, which would severely limit the drone's options for movement.