Artificial Intelligence Project 1 (CS520)
Ghosts in the Maze

Name – Viswajith Menon
Net ID – VM623

The Environment –

The Maze -

The environment consists of mazes of the dimension 51x51. Each cell in the maze is filled with either 1s or 0s. The 1s denote a cell through which the agent could move and the 0s denote cells which are blocked due to walls.

The 1s are populated with a probability of 0.72 and the 0s are populated with a probability of 0.28. Once the maze is populated, a couple of conditions are checked before using it as part of the simulation for the agents.
Condition 1 –
The source and destinations are unblocked cells.
Condition 2 –
There is a viable path between the source and the destination.

Creating the maze with the given probabilities and initializing another matrix called 'visited' with the value 'N' -

```
p1 = 0.28
p2 = 0.72
matrix = np.array(np.random.choice([0, 1], size=(row_size,col_size), p=[p1, p2]))
visited = np.array(np.random.choice(['N', 'N'], size=(row_size, col_size), p=[p1, p2]))
```

The visited matrix is created to make sure we can cross check the nodes that we have visited as we traverse the maze.

Grey Boxed Question
What algorithm is most useful for checking for the existence of these paths? Why?
Finding a path from the source to the destination –
Algorithm used – Breadth First Search (BFS).
The BFS algorithm makes sure that the shortest path between the source and destination is found. The only disadvantage is that all unblocked cells of the 2D matrix is visited before it returns the path. This is a more time-consuming approach as compared to a Depth First Search (DFS), which returns a path as soon as it's found.

The reason I am using a BFS initially to determine if the maze is viable is to make sure I can reuse this same path for the maze for Agent 1 to traverse on. Agent 1 evaluates the shortest path from the source to the destination.

Once the maze is deemed to be usable and the path is found using a BFS, both are stored in two separate dictionaries – Maze_Set and Path_Set.

Implementation of BFS –

A class Matrix is created to keep track of each cell's parent and the distance it has already travelled in a particular path.

```
class Matrix:
  def __init__(self, row, column,distance, parent, parentx, parenty):
    self.row = row
    self.column = column
    self.distance = distance
    self.parent = parent
    self.parentx = parentx
    self.parenty = parenty
    # Used for A-Star algorithm
    self.g = 0
    self.h = 0
    self.f = 0
```

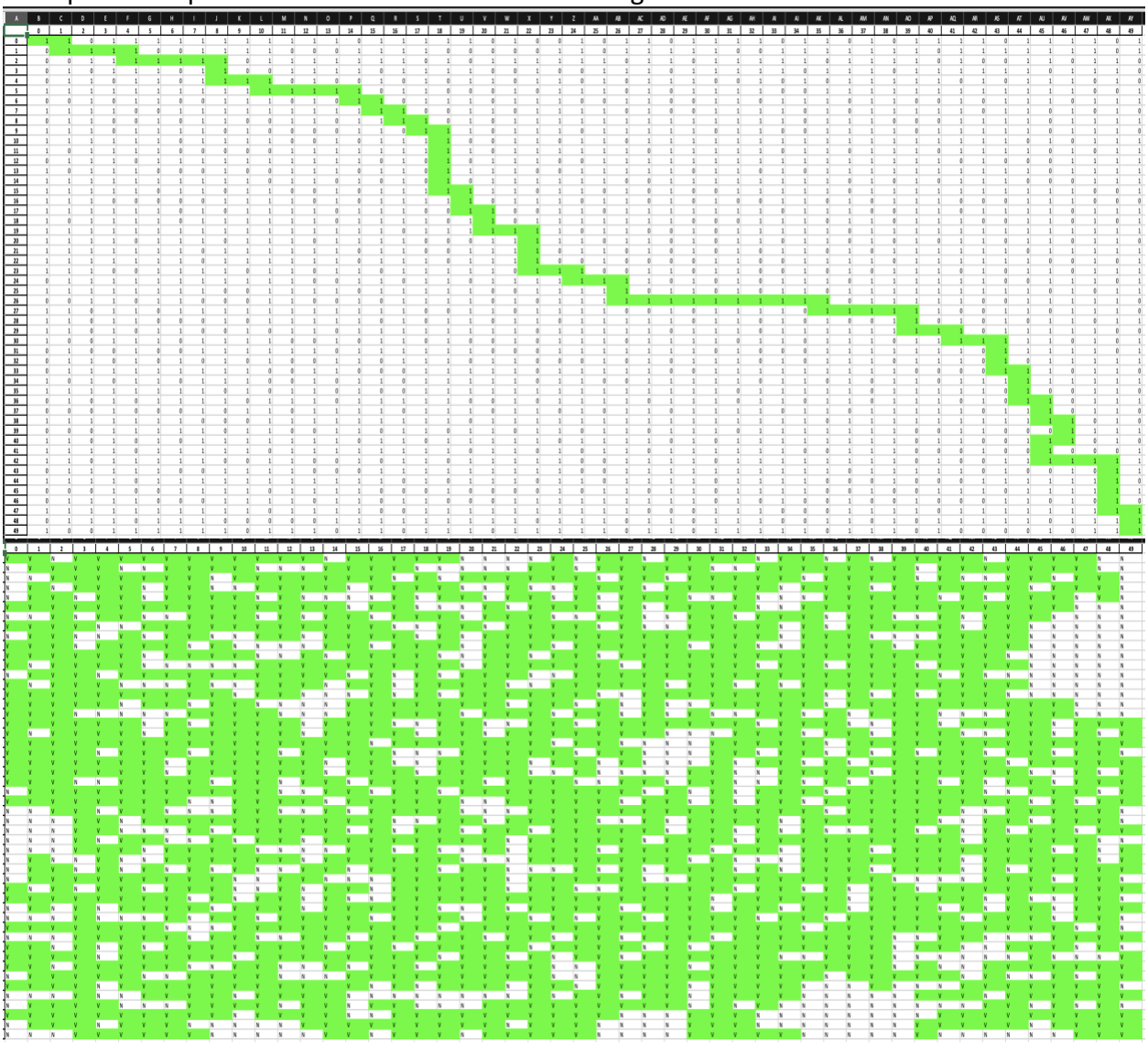
The starting point is defined as an object of the class Matrix.

```
source = Matrix(0,0,0,None,0,0)
```

I have used a deque to implement a queue data structure. Using this allows me to access and remove cells in the order they were visited, which helps me achieve a level order traversal of the matrix.

```
    q = deque()
    q.append(source)
    visited[source.row][source.column] = 'V'
    while (q):
        temp = q.popleft()
        if(temp.row == row_size-1 and temp.column == col_size-1):
            ans = temp.distance
            path = []
            t = temp
            while(t):
                l = []
                l.append(t.parentx)
                l.append(t.parenty)
                path.append(l)
                t = t.parent
        x = temp.row
        y = temp.column
        l = []
        l.append(x)
        l.append(y)
        if (y - 1 >= 0 and matrix[x][y - 1] == 1 and visited[x][y - 1] == 'N'):
            q.append(Matrix(x,y - 1, temp.distance+1,temp,x,y))
            visited[x][y-1] = 'V'
        if (y + 1 <= col_size-1 and matrix[x][y + 1] == 1 and visited[x][y + 1] == 'N'):
            q.append(Matrix(x, y + 1,temp.distance+1,temp,x,y))
            visited[x][y+1] = 'V'
        if (x - 1 >= 0 and matrix[x - 1][y] == 1 and visited[x - 1][y] == 'N'):
            q.append(Matrix(x - 1, y,temp.distance+1,temp,x,y))
            visited[x-1][y] = 'V'
        if (x + 1 <= row_size-1 and matrix[x + 1][y] == 1 and visited[x + 1][y] == 'N'):
            q.append(Matrix(x + 1, y,temp.distance+1,temp,x,y))
            visited[x+1][y] = 'V'

    if (visited[row_size-1][col_size-1] == 'V'):
        Maze_Set[count] = matrix
        Path_Set[count] = path
```

Example of the path found and the cells visited during a BFS –



As shown in the image, all the accessible nodes are visited before returning the path.

Example of how the dictionary stores the maze and path for a 5x5 maze –

Maze_Set= {0: array([[1, 1, 1, 0, 0],
[1, 0, 1, 1, 1],
[0, 1, 0, 1, 1],
[1, 0, 1, 1, 0],
[1, 0, 1, 1, 1]])}

Path_set = {0: [[0, 0], [0, 0], [0, 1], [0, 2], [1, 2], [1, 3], [2, 3], [3, 3], [4, 3],[4,4]]}

In both the dictionaries shown, the key is the ‘maze number’ while the value stores the maze and path itself.

The Ghosts –

Survivability of each agent is evaluated with respect to the number of ghosts.
The ghosts are placed in random points of the maze during the start. (A ghost cannot spawn in the cell [0,0].). These same initial ghost positions are sent to all the Agent functions.

Implementation –
Ghost starting positions –

```
number_of_ghost_list=[25,35,45,55,65]
ag1_survival=[]
ag2_survival=[]
ag3_survival=[]
ag4_survival=[]
for i in number_of_ghost_list:
    print("Iteration ",i)
    count=0
    ghost_position_list=[]
    while count<i:
        ghost_start_row=random.choice(index)
        ghost_start_column=random.choice(index)
        if(ghost_start_row==0 and ghost_start_column==0):
            count=count
        else:
            ghost_position_list.append([ghost_start_row,ghost_start_column])
            count=count+1
```

As shown in the image above, while iterating over the specific number of ghosts, the ghost positions are stored in a list ‘ghost_position_list’.

Ghost movement –
The function for ghost movement takes the ghost’s current position [ghost_start_row, ghost_start_column] and the maze as an input.
It then finds out the possible directions the ghost can travel and stores it in a list called ‘directions’. Random.choice is run over this list to get the direction the ghost has chosen to travel.

```
def ghost(ghost_start_row, ghost_start_column, temp_matrix):
    final_pos = []
    #print(ghost_start_row - 1, ghost_start_row + 1, ghost_start_column - 1, ghost_start_column + 1)
    flag=0
    if((ghost_start_row - 1 < 0 and ghost_start_column - 1 < 0) or (ghost_start_row - 1 < 0 and ghost_start_column + 1 > col_size-1)
        or (ghost_start_row + 1 > row_size-1 and ghost_start_column - 1 <0) or (ghost_start_row + 1 > row_size-1 and ghost_start_column + 1 > col_size-1)):
        if(ghost_start_row - 1 < 0 and ghost_start_column - 1 < 0):
            directions = ["Down","Right"]
            flag=1
        if(ghost_start_row - 1 < 0 and ghost_start_column + 1 > col_size-1):
            directions = ["Down","Left"]
            flag=1
        if(ghost_start_row + 1 > row_size-1 and ghost_start_column - 1 <0):
            directions = ["Up","Right"]
            flag=1
        if(ghost_start_row + 1 > row_size-1 and ghost_start_column + 1 > col_size-1):
            directions = ["Up","Left"]
            flag=1
    else:
        if(ghost_start_row - 1 < 0):
            directions = ["Down","Left","Right"]
            flag=1
        if(ghost_start_row + 1 > row_size-1):
            directions = ["Up","Left","Right"]
            flag=1
        if(ghost_start_column - 1 < 0):
            directions = ["Down","Up","Right"]
            flag=1
        if(ghost_start_column + 1 > col_size-1):
            directions = ["Down","Left","Up"]
            flag=1
    if(flag==0):
        directions = ["Up","Down","Left","Right"]
    direction_choice = random.choice(directions)
    #print(direction_choice)
    blocked = ["Move","Stay"]
    Movement_choice = "Move"
```

Another list ‘blocked’ is created with the values ‘Move’ and ‘Stay’. If the ghost’s current position is a blocked cell, then a value from ‘blocked’ is chosen and the ghost decides to move or stay according to the generated choice. (50% chance of moving if the ghost is in a blocked cell).

```
    blocked = ["Move","Stay"]
    Movement_choice = "Move"
    if(temp_matrix[ghost_start_row][ghost_start_column]==0):
        Movement_choice = random.choice(blocked)
    if(Movement_choice == "Stay"):
        final_pos.append(ghost_start_row)
        final_pos.append(ghost_start_column)
    else:
        # UP
        if(direction_choice == "Up"):
            #print("Checkesh ",temp_matrix[ghost_start_row - 1][ghost_start_column])
            if(temp_matrix[ghost_start_row - 1][ghost_start_column] == 1):
                final_pos.append(ghost_start_row - 1)
                final_pos.append(ghost_start_column)
            elif(temp_matrix[ghost_start_row - 1][ghost_start_column] == 0):
                blocked_choice = random.choice(blocked)
                if(blocked_choice == "Move"):
                    final_pos.append(ghost_start_row - 1)
                    final_pos.append(ghost_start_column)
                elif(blocked_choice == "Stay"):
                    final_pos.append(ghost_start_row)
                    final_pos.append(ghost_start_column)
            elif(temp_matrix[ghost_start_row - 1][ghost_start_column] == 6):
                final_pos.append(ghost_start_row)
                final_pos.append(ghost_start_column)
```

In the else block the directions down, left and right are executed as well.
The final position of the ghost is stored in a list ‘final_pos’ and returned to the calling function.

Agent 1 –

Agent 1 plans the shortest path through the maze. Agent 1 traverses the maze unbothered by the movement of the ghosts.
As agent 1 uses the shortest path to reach the destination, we can use the path we generated using BFS while checking the maze's usability.
We can access the path the agent 1 will use by using the dictionary Path_Set which has the paths for each maze.

While traversing the specific BFS-generated path for a maze, the ghosts move every time the agent moves to the next step in the path. If the ghost and agent end up in the same cell the agent 1 is considered dead.

Once the agent 1 dies, a variable 'agent_1_death_count' is incremented and the next maze is checked the same way.

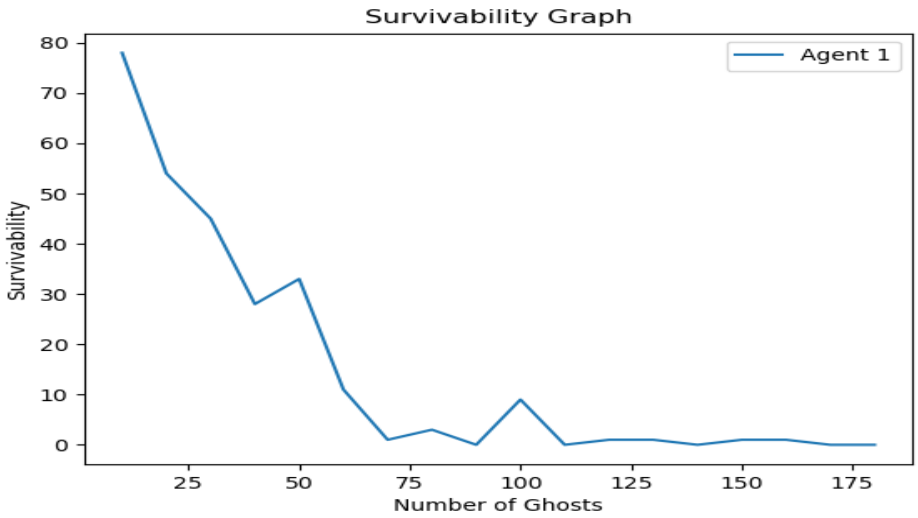
At the end 'agent_1_death_count' is returned and survivability is calculated using the formula –
Agent_1_survivability = (1 - (agent_1_death_count/Number_Of_Mazes)) * 100.
This formula remains the same for all agents.

Implementation of Agent 1 –

As shown in the screenshot below the initial positions of the ghost are passed into the Agent 1 function. I have then iterated over every maze in the dictionary using the path generated during BFS.
After every move the agent makes, the 'ghost' function is called to move the ghosts.

```
def Agent1(ghost_position_list):
    agent_1_death_count=0
    for i in range(0,len(Maze_Set)):
        print("Agent 1 ",i)
        flag=0
        temp_matrix = Maze_Set[i]
        temp_path=Path_Set[i]
        temp_path = temp_path[::-1]
        temp_path.pop(0)
        for i in temp_path:
            if([i[0],i[1]] in ghost_position_list):
                flag=1
                agent_1_death_count=agent_1_death_count+1
                break

            for i in range(len(ghost_position_list)):
                ghost_current=ghost_position_list[i]
                ghost_new_position=ghost(ghost_current[0],ghost_current[1],temp_matrix)
                ghost_position_list[i]=[ghost_new_position[0],ghost_new_position[1]]

    print("agent_1_death_count ",agent_1_death_count)
    return agent_1_death_count
```

Agent 1's performance while the number of ghosts in the maze increase.
Number of mazes used – 100.



Agent 2 –

Agent 2 is aware of the changing environment while travelling and considers the positions of the ghost while planning a path.
Agent 2 analyzes the entire matrix every time it moves and calculates the path every time-step while considering the position of the ghosts.

I have evaluated the path Agent 2 takes using the A-star algorithm as running a BFS from every cell the agent is in, is a highly time-consuming operation.

The A-star algorithm is a smart algorithm which is guided by a certain heuristic. At every cell the agent 2 finds itself, 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 of agent 2.

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 the Agent is only allowed to move in the cardinal directions (Up, down, left and right).

```python
#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)
```

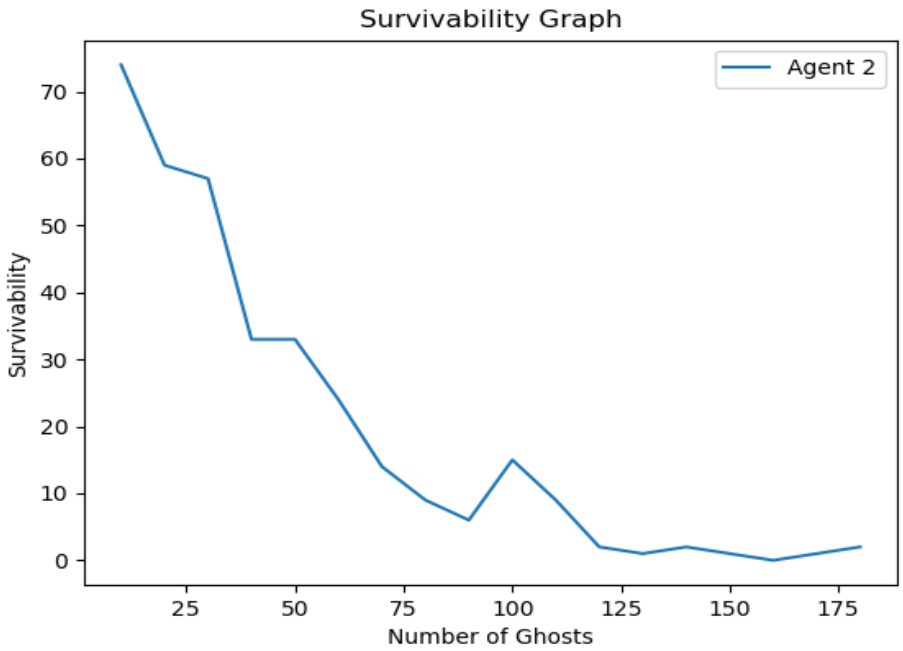
Every time a path is evaluated from a cell, it is stored in a list called 'new_path'. The first time the agent 2 runs from position [0,0], it calls the A-star algorithm to calculate a path from the source to the destination in such a way that it evades both the ghosts and the walls.
The agent then moves to the next step in this path.

Grey Boxed Question
Agent 2 requires multiple searches - you'll want to ensure that your searches are efficient as possible, so they don't take much time. Do you always need to replan? When will the new plan be the same as the old plan, and as such you won't need to recalculate?
The path is recalculated again only if a ghost ends up in an adjacent position to the current position of the agent. This helps us reduce the overall time of the program as it is unnecessary to calculate the path from a particular point if the ghosts are not in any of the adjacent cells.

Agent 2 Implementation –

```
def Agent2(ghost_position_list_2):
    agent_2_death_count=0
    for i in range(0,len(Maze_Set)):                                        # Moving over all the valid mazes
        flag=0
        temp_matrix = Maze_Set[i]
        print("Agent 2 ",i)
        current_row=0
        current_column=0
        new_path=[]
        ghost_position=[]
        for i in ghost_position_list_2:
            ghost_position.append(i)
        while(TRUE):
            if([current_row,current_column] in ghost_position):             # Checking if the agent and ghost are in the same position
                flag=1
                agent_2_death_count=agent_2_death_count+1
                break

            if(flag==1):
                break

            if(current_row==row_size-1 and current_column==col_size-1):     # The Agent has reached the destination cell.
                break

            adjacent_cells=[[current_row,current_column-1],[current_row,current_column+1],[current_row-1,current_column],[current_row+1,current_column],
                            [current_row+1,current_column-1],[current_row+1,current_column+1],[current_row-1,current_column-1],[current_row-1,current_column+1]]


            if(len(new_path)==0 or common_member(adjacent_cells,ghost_position)==True):   # Recalculating the path if the ghosts are present in the adjacent
                new_path=astar_original(temp_matrix,(current_row,current_column),ghost_position)

            if(len(new_path)==0):                                           # As there is no viable path from the current cell to the destinati
            else:…


            for i in range(len(ghost_position)):                                    # GHOST MOVEMENT…

            if(flag==1):
                break

    print("agent_2_death_count ",agent_2_death_count)
    return agent_2_death_count
```

Agent 2's performance while the number of ghosts in the maze increase.
Number of mazes used – 100.



Agent 1 vs Agent 2 –

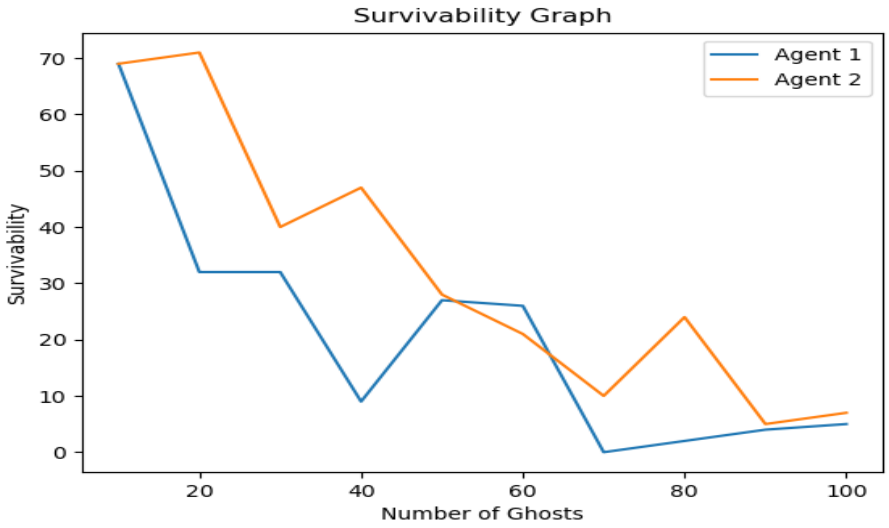
Mazes used – 100

Agent 1

Number of ghosts	Survivability
10	69.0
20	31.999999999999996
30	31.999999999999996
40	8.999999999999996
50	27
60	26
70	0
80	2.0000000000000018
90	4.0000000000000036
100	5.000000000000004

Agent 2

Number of ghosts	Survivability
10	69.0
20	71.0
30	40.0
40	47.0
50	28.000000000000004
60	20.999999999999996
70	9.999999999999998
80	24
90	5.000000000000004
100	6.999999999999995



Agent 3 –

Like agent 2, agent 3 is aware of the changing environment while travelling and considers the positions of the ghost while planning a path.
But unlike agent 2, agent 3 does not recalculate the path at every time-step. Agent 3 simulates what agent 2 would do from its current position and all the adjacent positions and then makes an informed choice.

I have implemented this by creating a function called 'Agent_2_implementation'. This function is used by the agent 3 to run an agent 2 simulation from any point in the maze.
This function takes the maze, current position of the agent and the ghost positions as an input. It then simulates the agent's movement until it either reaches the destination or dies.
If the destination is reached the function returns 1, else it returns 0.

```
def Agent2_implementation(temp_matrix,s,temp_ghost_position):
    agent_2_death_count=0
    flag=0
    current_row=s[0]
    current_column=s[1]
    new_path=[]
    ghost_position=[]
    for i in temp_ghost_position:
        ghost_position.append(i)
    while(TRUE):
        if([current_row,current_column] in ghost_position):
            flag=1
            agent_2_death_count=agent_2_death_count+1
            break

        if(flag==1):
            break

        if(current_row==row_size-1 and current_column==col_size-1):
                break

        adjacent_cells=[[current_row,current_column-1],[current_row,current_column+1],[current_row-1,current_column],[current_row+1,current_column]] #13th oct

        #if(len(new_path)==0 or common_member(new_path,ghost_position)==True): 13th oct
        if(len(new_path)==0 or common_member(adjacent_cells,ghost_position)==True):        #13th Oct
            new_path=astar_original(temp_matrix,(current_row,current_column),ghost_position)

        if(len(new_path)==0):                                                    # Checking nearest ghost and moving away
        else:

        for i in range(len(ghost_position)):                                     # Ghost Movement

        if(flag==1):
            break

    if(flag==0):
        return 1
    else:
        return []
```

The agent 3 function calls the function 'Agent_2_implementation' 5 times from each adjacent cell including its current position.
Agent 3 then plans its step based on whichever move has the highest survivability. If all moves result in the death of the agent, the agent simply moves away from the nearest ghost.

If 2 different moves show the same survivability, the agent chooses to move in the direction closest to the destination node.

The issue with Agent 3 –

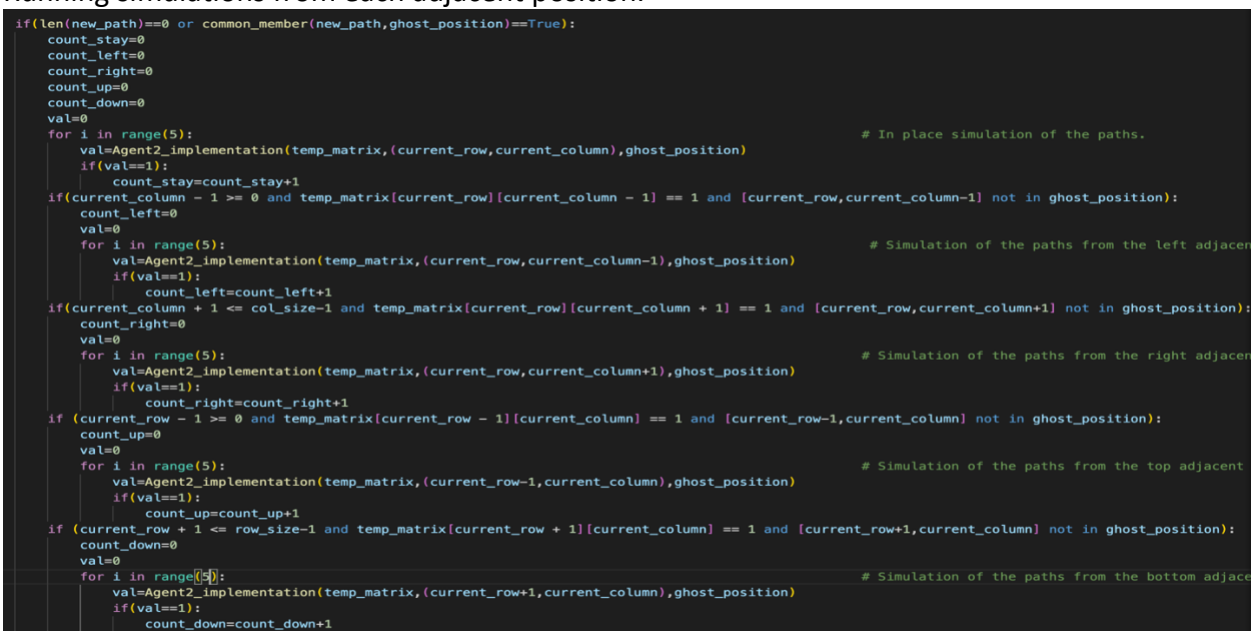
Agent 3 forecasting the path from the adjacent cells at every time-step leads to the agent being stuck oscillating within a couple of cells.

This happens because the result of the simulation of the previous cells might show a higher survivability compared to any new position the ghost tries to move to. This leads to the agent continuously swapping positions between 2 or 3 cells.

To avoid this issue, I simulate the paths from the adjacent cells only if a ghost moves into the path the agent is currently moving on.

For example, initially the agent is at the source [0,0]. Simulations are then run for the unblocked adjacent cells. The block with the highest survivability is then chosen and a path from this point to the destination is evaluated and stored. In the next iteration, if the ghosts have not moved into a position in this path, the agent moves along. If the ghosts do move into the already calculated path, the agent runs the simulations from the new adjacent cells.

Running simulations from each adjacent position.



```
if(len(new_path)==0 or common_member(new_path,ghost_position)==True):
    count_stay=0
    count_left=0
    count_right=0
    count_up=0
    count_down=0
    val=0
    for i in range(5):                                                          # In place simulation of the paths.
        val=Agent2_implementation(temp_matrix,(current_row,current_column),ghost_position)
        if(val==1):
            count_stay=count_stay+1
    if(current_column - 1 >= 0 and temp_matrix[current_row][current_column - 1] == 1 and [current_row,current_column-1] not in ghost_position):
        count_left=0
        val=0
        for i in range(5):                                                       # Simulation of the paths from the left adjacen
            val=Agent2_implementation(temp_matrix,(current_row,current_column-1),ghost_position)
            if(val==1):
                count_left=count_left+1
    if(current_column + 1 <= col_size-1 and temp_matrix[current_row][current_column + 1] == 1 and [current_row,current_column+1] not in ghost_position):
        count_right=0
        val=0
        for i in range(5):                                                      # Simulation of the paths from the right adjacen
            val=Agent2_implementation(temp_matrix,(current_row,current_column+1),ghost_position)
            if(val==1):
                count_right=count_right+1
    if (current_row - 1 >= 0 and temp_matrix[current_row - 1][current_column] == 1 and [current_row-1,current_column] not in ghost_position):
        count_up=0
        val=0
        for i in range(5):                                                      # Simulation of the paths from the top adjacent
            val=Agent2_implementation(temp_matrix,(current_row-1,current_column),ghost_position)
            if(val==1):
                count_up=count_up+1
    if (current_row + 1 <= row_size-1 and temp_matrix[current_row + 1][current_column] == 1 and [current_row+1,current_column] not in ghost_position):
        count_down=0
        val=0
        for i in range(5):                                                      # Simulation of the paths from the bottom adjace
            val=Agent2_implementation(temp_matrix,(current_row+1,current_column),ghost_position)
            if(val==1):
                count_down=count_down+1
```

Moving to the cell with the highest survivability –

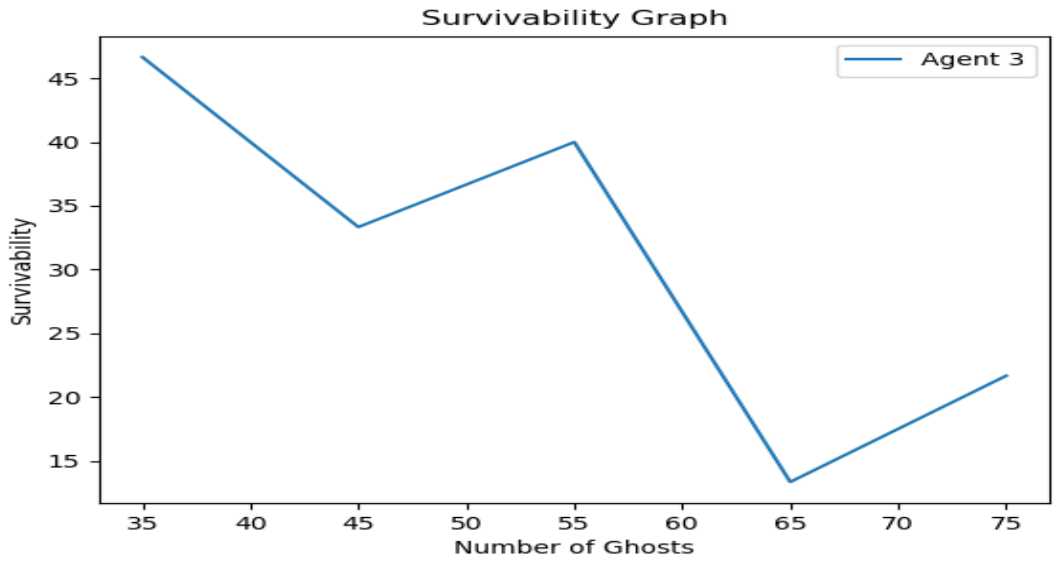
```
l=[count_stay,count_left,count_right,count_up,count_down]
max_val = sorted(l)[-1]
if(count_down == max_val):                                    # Moving to the cell with the highest rate of survivability.
    current_row=current_row+1
    current_column=current_column
elif(count_right == max_val):
    current_row=current_row
    current_column=current_column+1
elif(count_stay == max_val):
    current_row=current_row
    current_column=current_column
elif(count_up == max_val):
    current_row=current_row-1
    current_column=current_column
elif(count_left == max_val):
    current_row=current_row
```

Calculating the path from the cell that the agent chooses to move –

```
if(len(new_path)==0 or common_member(new_path,ghost_position)==True):
    new_path=astar_original(temp_matrix, (current_row,current_column) , ghost_position)          # Calculating the new path
else:
    new_path.pop(0)
    current_row=new_path[0][0]                          # Moving to the next step in the path if a ghost has not moved into any cell in the path
    current_column=new_path[0][1]
```

Agent 3's performance while the number of ghosts in the maze increase.
Number of mazes used – 50.

Number of Ghosts	Survivability
35	46.666666666666664
45	33.333333333333336
55	40.0
65	13.33333333333333
75	21.666666666666664



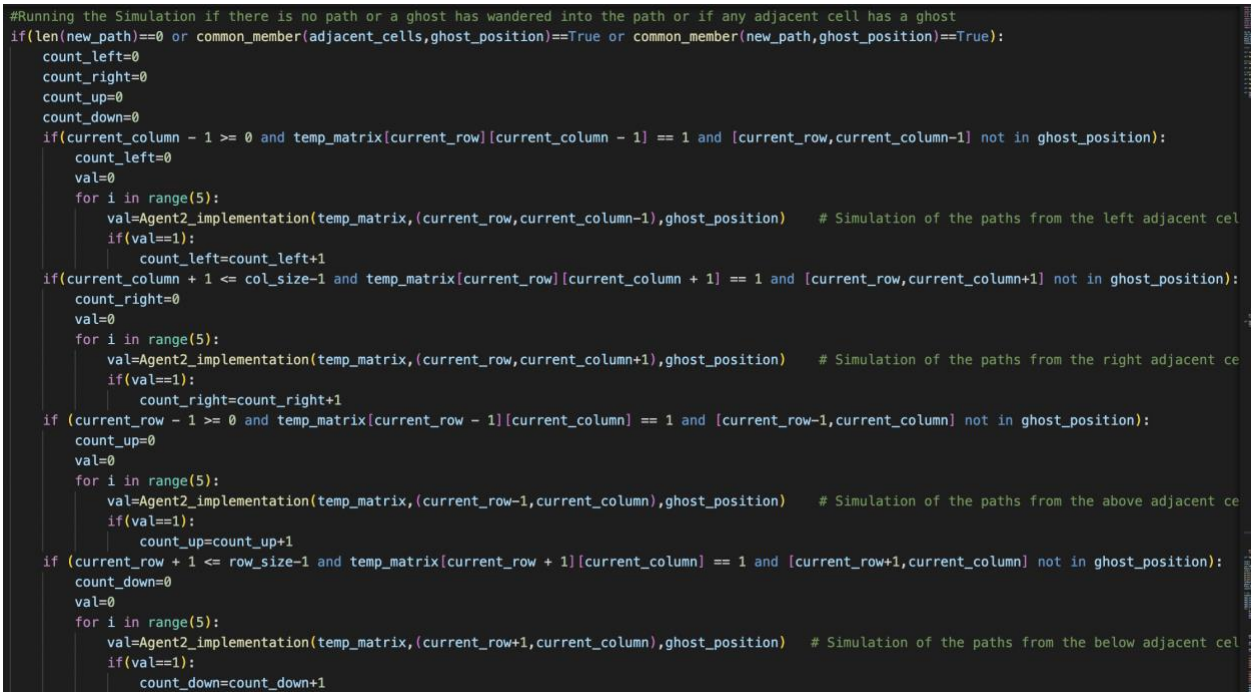
Grey Boxed Question

Agent 3 requires multiple searches - you'll want to ensure that your searches are efficient as possible so they don't take much time. Additionally, if Agent 3 decides there is no successful path in its projected future, what should it do with that information? Does it guarantee that success is impossible?

If the simulations of the adjacent node result in no successful path it does not indicate that success is impossible. It's possible that a ghost has occupied the destination cell. In this case it's better to move away from the closest ghost and then retry the simulations after the ghosts move again.

Agent 4 –

I have implemented Agent 4 by slightly modifying what agent 3 does.
Agent 3 runs the simulations from the adjacent cells only if a ghost moves into the path agent 3 is taking.
Agent 4 does the same while additionally also checking the adjacent cells.
If any adjacent cell to the agent contains a ghost, the agent runs a simulation from the unblocked adjacent cells to evaluate the highest survivability and then makes a move in that direction.



```
#Running the Simulation if there is no path or a ghost has wandered into the path or if any adjacent cell has a ghost
if(len(new_path)==0 or common_member(adjacent_cells,ghost_position)==True or common_member(new_path,ghost_position)==True):
    count_left=0
    count_right=0
    count_up=0
    count_down=0
    if(current_column - 1 >= 0 and temp_matrix[current_row][current_column - 1] == 1 and [current_row,current_column-1] not in ghost_position):
        count_left=0
        val=0
        for i in range(5):
            val=Agent2_implementation(temp_matrix,(current_row,current_column-1),ghost_position)    # Simulation of the paths from the left adjacent cel
            if(val==1):
                count_left=count_left+1
    if(current_column + 1 <= col_size-1 and temp_matrix[current_row][current_column + 1] == 1 and [current_row,current_column+1] not in ghost_position):
        count_right=0
        val=0
        for i in range(5):
            val=Agent2_implementation(temp_matrix,(current_row,current_column+1),ghost_position)    # Simulation of the paths from the right adjacent ce
            if(val==1):
                count_right=count_right+1
    if (current_row - 1 >= 0 and temp_matrix[current_row - 1][current_column] == 1 and [current_row-1,current_column] not in ghost_position):
        count_up=0
        val=0
        for i in range(5):
            val=Agent2_implementation(temp_matrix,(current_row-1,current_column),ghost_position)    # Simulation of the paths from the above adjacent ce
            if(val==1):
                count_up=count_up+1
    if (current_row + 1 <= row_size-1 and temp_matrix[current_row + 1][current_column] == 1 and [current_row+1,current_column] not in ghost_position):
        count_down=0
        val=0
        for i in range(5):
            val=Agent2_implementation(temp_matrix,(current_row+1,current_column),ghost_position)    # Simulation of the paths from the below adjacent cel
            if(val==1):
                count_down=count_down+1
```

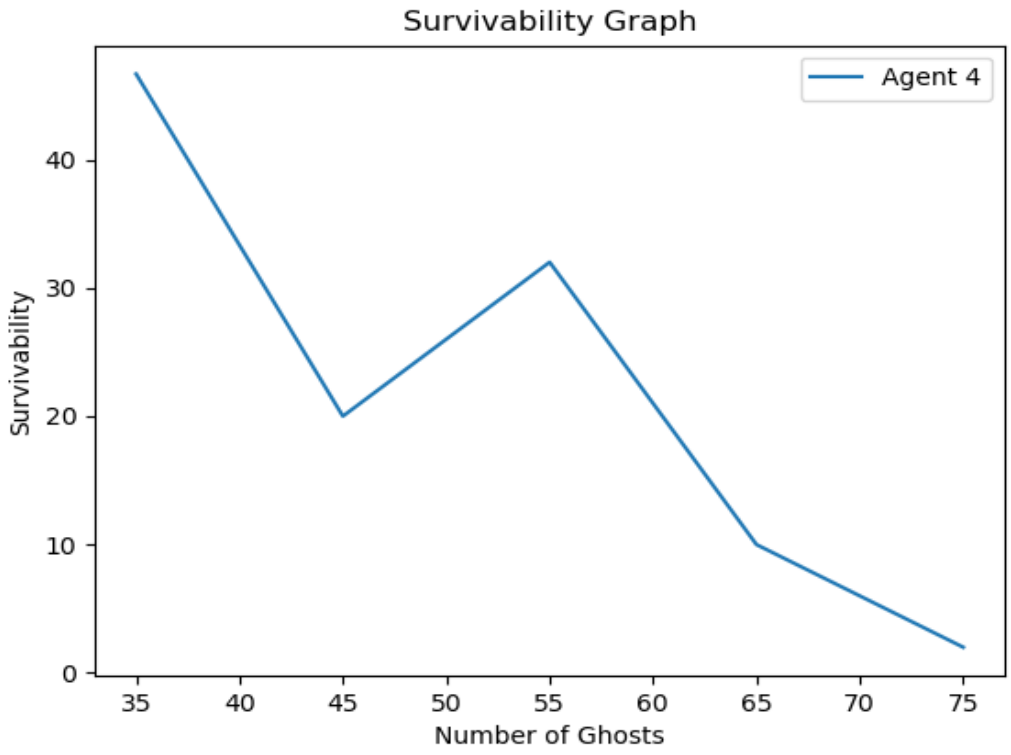
The movement to the cell with the highest survivability is the same as that of agent 3 except for the fact that agent 4 does not stay in the same place. Survivability is only checked from the adjacent cells and not the current cell.

Calculating the path from the cell that the agent chooses to move –

```
if(len(new_path)==0 or common_member(adjacent_cells,ghost_position)==True or common_member(new_path,ghost_position)==True):
    new_path=astar_original(temp_matrix, (current_row,current_column) , ghost_position)   # Calculating the new path
else:
    new_path.pop(0)
    current_row=new_path[0][0]                    # Moving to the next step in the path if a ghost is not in any cell in the path or any adjacent cell
    current_column=new_path[0][1]
```

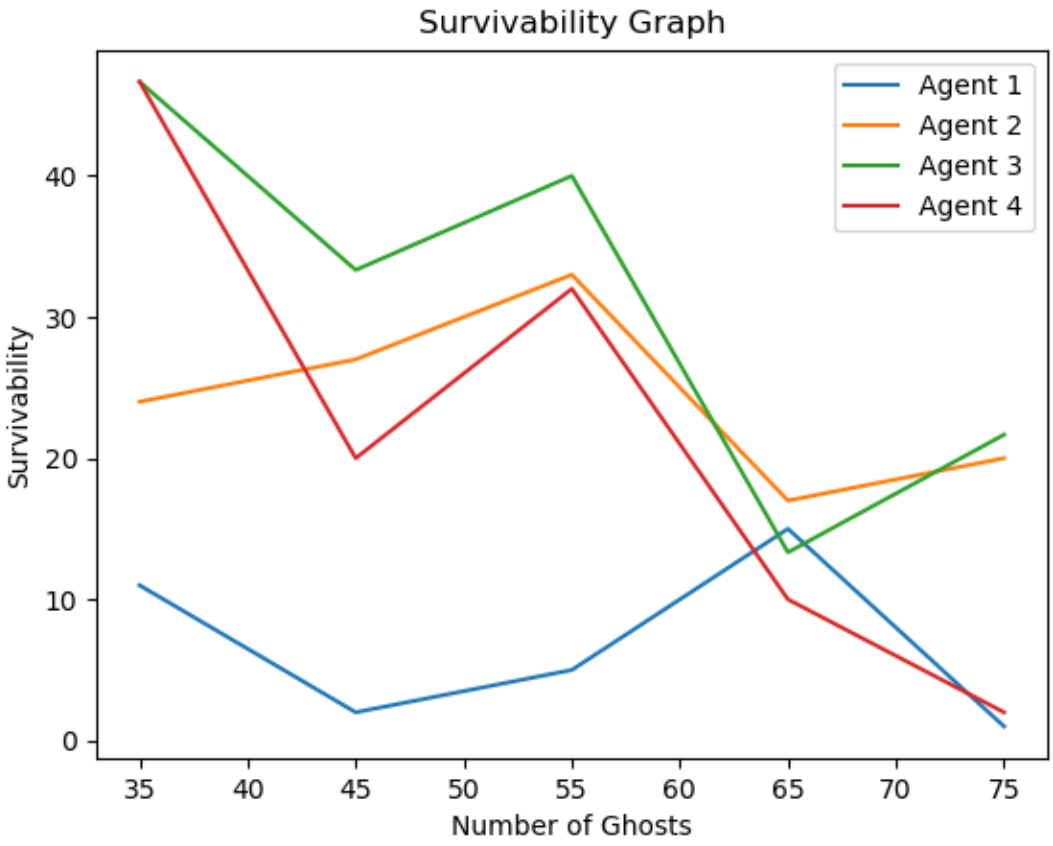
Agent 4's performance while the number of ghosts in the maze increase.
Number of mazes used – 50.

Number of Ghosts	Survivability
35	46.666666666666664
45	19.999999999999996
55	31.999999999999996
65	9.999999999999998
75	2.0000000000000018



Agent 1 vs Agent 2 vs Agent 3 vs Agent 4 –

Number of mazes used 50 –
Number of ghosts – [35,45,55,65,75]



From the graph, it is clear that agent 3 performs better than the rest of the agents.
Agent 4 seems to be running into the same issue agent 3 had initially faced. Like agent 3, agent 4 runs the simulation whenever a ghost wanders into the evaluated path. But additionally running the simulations every time a ghost wanders into an adjacent cell is drastically increasing the number of times agent 4 simulates. This again leads to the agent 4 being stuck oscillating within a couple of cells, which in turn increases the chances of a ghost killing it.

Lower Information Environment

In this environment, the agent does not take the ghosts which are present in blocked cells into consideration when evaluating a path.

I have implemented this by creating a new list called 'ghosts_not_in_walls'. If the position of a particular ghost is in an unblocked cell, it is copied into the new 'ghosts_not_in_walls' list. This list is then passed into the path finding algorithm used by the agent.
This ensures that the agent can only plan a path which evades the blocked cells and the ghosts present in the unblocked cells.

```
ghosts_not_in_walls=[]
for i in ghost_position:
    if(temp_matrix[i[0]][i[1]]==1):
        ghosts_not_in_walls.append([i[0],i[1]])
```

Agent 1 vs Lower Information Environment

Agent 1's performance is unaffected by this change in environment as the agent 1 was initially blind to the ghosts while planning the path.

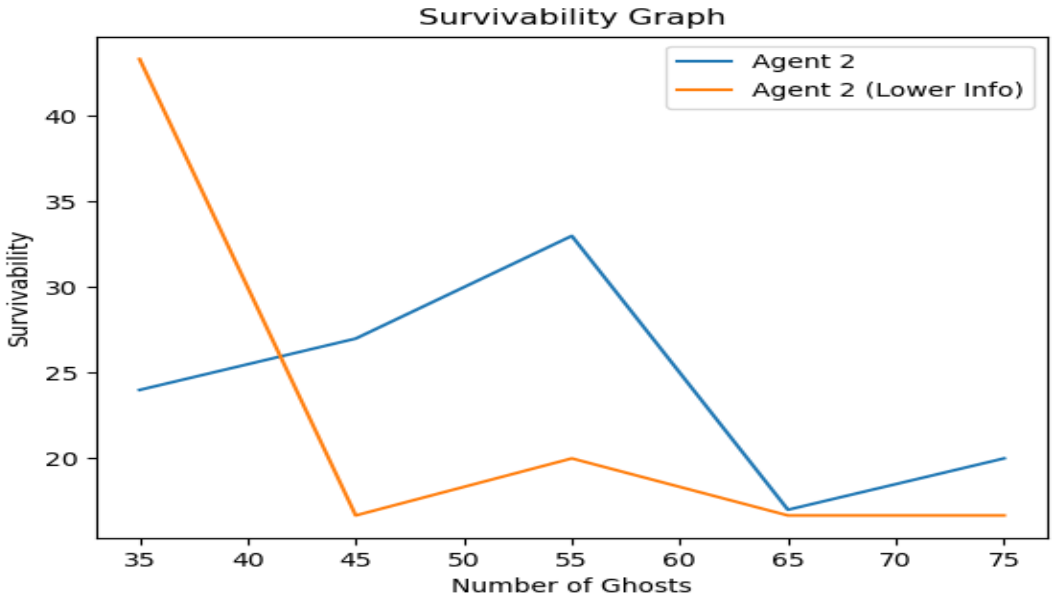
Agent 2 vs Lower Information Environment

Number of mazes – 50
Number of ghosts – [35,45,55,65,75]
Agent 2

Number of Ghosts	Survivability
35	24
45	27
55	32.99999999999999
65	17.000000000000004
75	19.999999999999996

Agent 2(With Low Info)

Number of Ghosts	Survivability
35	43.333333333333336
45	16.666666666666664
55	19.999999999999996
65	16.666666666666664
75	16.666666666666664



Agent 3 vs Lower Information Environment

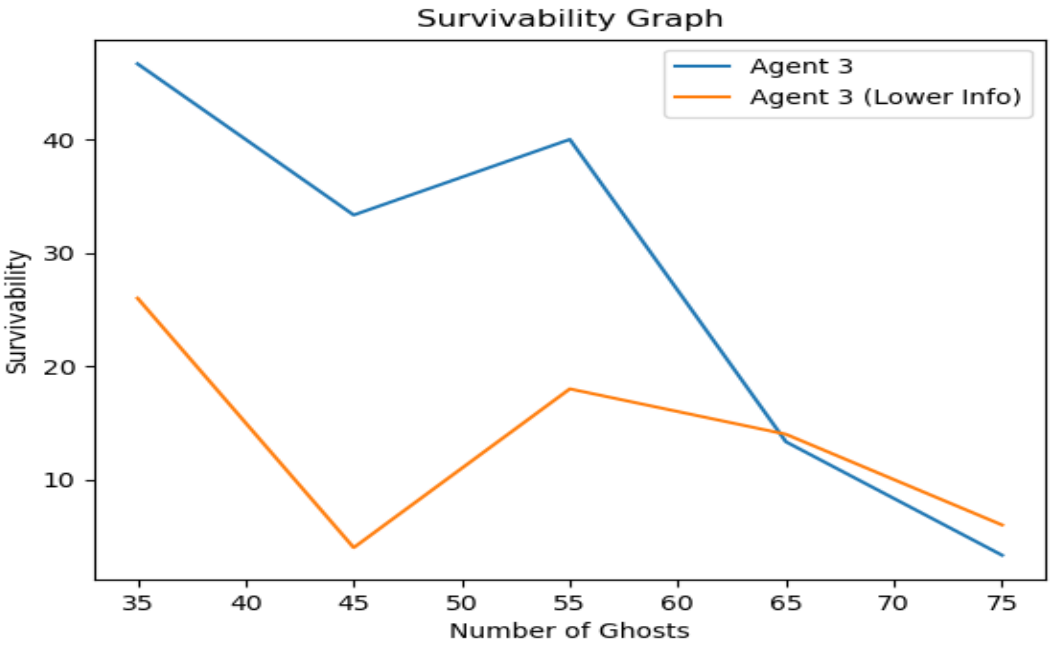
Number of mazes – 50
Number of ghosts – [35,45,55,65,75]

Agent 3 -

Number of Ghosts	Survivability
35	46.666666666666664
45	33.333333333333336
55	40.0
65	13.33333333333333
75	3.3333333333333326

Agent 3(With Low Info)

Number of Ghosts	Survivability
35	26
45	4.0000000000000036
55	18.000000000000004
65	14.000000000000002
75	6.000000000000005


Survivability Graph
Agent 3
Agent 3 (Lower Info)
Survivability
40
30
20
10
35
40
45
50
55
60
65
70
75
Number of Ghosts