



CS520

Project 1: Ghosts in the Maze

Kunjan Vaghela (kv353)
Rutgers University

Manan Shukla (ms3481)
Rutgers University

Mitul Shah (ms3518)
Rutgers University

October 14, 2022

Contents

1	Introduction	2
1.1	Problem Statement	2
1.2	Problem Environment	3
1.2.1	State of Cells	3
1.2.2	Ghosts	3
1.2.3	Agents	3
1.2.4	Parameters	4
1.3	Graph Traversal Algorithms:	4
1.3.1	Depth First Search (DFS):	4
1.3.2	Breadth First Search (BFS):	5
1.3.3	A* (A Star):	6
2	Implementation	8
2.1	The Environment	8
2.2	Agents	9
2.2.1	Agent 1 : Plan Once and Execute Blindly	9
2.2.2	Agent 2 : Plan at Every step	10
2.2.3	Agent 3 : Forecasting on Agent 2's knowledge	12
2.2.4	Agent 4 : Plan only when needed	14
2.2.5	Agent 5 : Plan with impaired sight	16
3	Analysis	19
3.1	Question 1	19
3.2	Question 2	20
3.3	Question 3	21







1

Introduction

1.1 Problem Statement

Quantified Maze terrain has cells/blocks. The cells are either blocked or unblocked (traversable) randomly. Maze index $[0,0]$ represents the Start/ Source Cell and Maze index $[n,n]$ represents End/ Destination Cell. Multiple ways/-paths exist from the Source cell to the Destination Cell. The Maze contains multiple randomly spawned Ghosts (at timestamp 0); they randomly traverse in four Cardinal directions to the adjacent cell, one cell per timestamp. Different agents (one at a time) spawned at Maze index $[0,0]$, traversing towards the Goal similarly to ghosts but with different constraints based on environment and prescientific knowledge. At any time, the Agent can 'see' the entirety of the Maze and use this information to plan a path. Agents try to solve the Maze avoiding the Ghosts, and the Agent dies if the Agent encounters the Ghost in the same cell. Depending upon the constraints, suitable Search algorithms are to be chosen and justified by the insights drawn from the empirical performance data.

Problem Representation:

START					
					
					
					
					GOAL

1.2 Problem Environment

The environment is a 51x51 Discretized Maze. Cells can be blocked or unblocked. There are multiple ghosts in the Maze. Maze has a Start Point [0,0] and Goal Point [50,50]. Only 1 Agent can traverse at a time.

1.2.1 State of Cells

- Unblocked : Cells traversable by the Agent.
- Blocked : Cells that are not traversable by the agent. The agent needs to consider them and avoid these cells while planning for a path.

1.2.2 Ghosts

There are multiple randomly spawned Ghosts in the maze. They can travel in Cardinal directions and traverse through all the cells in the grid, but they cannot move outside the 51x51 grid. Multiple ghosts can occupy the same cell. Ghosts are adversaries of Agents; if a Ghost encounters the Agent, the Agent dies. Ghosts are unbiased with their movement through the maze, 0.25 Probability of Moving in each direction. If the next potential cell is blocked, there is a 0.50 Probability that the ghost will enter it. Ghosts cannot spawn at Start Cell [0,0] or Goal Cell [50,50]. Ghost Cells that are not traversable by the Agent. The Agent needs to consider them and avoid these cells while planning for a path.

1.2.3 Agents

The agent starts at maze [0,0] and attempts to navigate to maze [50,50]. It can only traverse in cardinal directions and only between unblocked squares. It also cannot move outside the 51x51 grid. At any time, the agent knows the entirety of the maze and utilizes this to solve the maze.

Types of agents and corresponding traversing constraints:

- Agent 1: Agent 1 is aware of the blocked and unblocked cells in the maze and plans the entire path from source to destination at timestamp 0, ignoring the position of ghosts and their movement. It simply plans once, executes the shortest route, and is exceptionally efficient. But it has no contingency plan if it encounters the ghosts while traversing the maze and dies. The agent starts at maze [0,0] and attempts to navigate to maze [50,50]. It can only travel in cardinal directions and only between unblocked squares. It also cannot move outside the 50x50 grid.
- Agent 2: Like Agent 1, Agent 2 is aware of the Cell states. Agent 2 plans its path and proceeds after considering the ghost's position at every timestamp. It changes its course of action if it thinks it will encounter a ghost. This ability is the main difference between Agent 1 and Agent 2. At any time, the agent knows the entirety of the maze and utilizes this to solve the maze
- Agent 3: Agent 3 is a revamped version of Agent 2. It can use the experiences gathered by agent 2. It rethinks at every timestamp and has

the option to stay in the same cell if required. Agent 3 simulates Agent 2 multiple times for decision-making and selects the most promising move. This knowledge adds to the planning and helps sketch a more informed action.

- Agent 4: Agent 4 is a different strategy altogether. It takes visibility and strike parameters into account. It calculates path using A* algo and until it encounters ghost in the visibility parameter (set to 3 here), it should not change the path potentially. Further, we have also added one more check using the strike param. If the ghost is in the visibility+1 range, strike is set to 1 and this help Agent4 ensure that the ghost might be in near future. It stays on the path and if now this time too, it encounters ghost in visibility+1 range, strike is updated to 2 and it moves away from the ghost.
- Agent 5: Agent 5 loses sight of the ghost(s) when the ghost enters a blocked cell. Agent 5 can see the ghost if the ghost is present in the unblocked cell, but loses sight of the ghost as soon as it enters the blocked cell. We implemented Agent 5 based on Agent 4's algorithm and by impairing Agent 5's ability to see the ghosts inside blocked cells.

1.2.4 Parameters

The Maze generated is based on the following parameters:

- Dimension : Dimension of the square Maze. [51x51]
- Density : Probability that a Maze cell is blocked [0.28]

1.3 Graph Traversal Algorithms:

1.3.1 Depth First Search (DFS):

DFS is an uninformed search algorithm for recursively traversing graph/tree data structures. The algorithm picks a path and explores the deepest node possible before traversing the other branch options encountered. If two (or multiple) nodes are encountered while traversing, DFS keeps these options in a fringe. Once the nodes in the current path are exhausted completely, DFS takes the element from the fringe following the LIFO (Last In First Out) principle. DFS can be used both to traverse and explore all the possible nodes of a graph/tree or can be used to search for a node in the data structure. If an infinite branch is present, DFS can get lost in the child nodes found and may never be able to find the goal node. Although this situation will never arise in our situation as the maze will be of finite dimension (here, 51x51). Thus, DFS can be used to find if a path exists between the two given nodes in the generated maze. Although DFS will not necessarily give the shortest route, DFS is quite efficient in searching for a course with less space complexity. Space Complexity of DFS: $O(b*d)$.

We implemented maze traversal via DFS in the following situations:

- 1) To find if a path exists from the start node to the goal node of the maze to ensure that the maze generated is valid for the agent to reach the goal node.
- 2) To check if each ghost spawned initially can be reached from the start node when the maze is created.

Maze traversal using DFS example:

S	1	2	X	
4	3	X		
5	X	10	11	12
6	7	8	X	13
X		9	X	E

Note: Here:

X= Blocked Cells.

S = Start position.

E = End Goal / Goal position.

Numbers denote the path taken by the algorithm to traverse through the maze.

1.3.2 Breadth First Search (BFS):

Like DFS, BFS is an uninformed search algorithm for recursively traversing graph/tree data structures. Before traveling to the deeper branch, the BFS algorithm explores all possible nodes at the current level. During traversal, if BFS finds child nodes of the current node, BFS will first traverse through all the existing nodes present (in the fringe) before picking up the child node for traversal. BFS achieves this using the FIFO (First In First Out) principle. Since BFS explores all the existing nodes in the fringe before exploring the next depth of nodes, BFS will always find the shortest path possible in the maze in between any two given nodes. But at the cost of exploring almost all the possible nodes in our maze before reaching the goal node. This is time-consuming and will not be suited for traversing the agent efficiently if the goal is only to survive and reach the goal node. Given its traversal technique, to get the position of the nearest ghost, we can use BFS. This is especially helpful with Agent 2 to satisfy the requirement of moving away from the nearest ghost when it cannot find a path through the maze. We implemented maze traversal via BFS to find the nearest ghost for Agent 2 if there is no path between the present node and the goal node.

Maze traversal using BFS example:

S	1	3	X	
2	4	X	14	16
5	X	10	12	13
6	7	8	X	15
X	9	11	X	E

BFS will take 17 steps to reach the end goal from the start cell in above maze.

S			X	
	G	X	4	G
	X	A	1	3
	6	2	X	
X		5	X	E

In the example above, among the two ghosts present in the environment, the nearest ghost coordinate was found on the 7th search.

Key:

S = Start position.

E = End Goal / Goal position.

X = Blocked cells.

G = Cell with Ghost.

A = Agent position.

Numbers denote the path taken by the algorithm to traverse through the maze.

1.3.3 A* (A Star):

A* search is an informed path search algorithm that can use a particular heuristic to get the best path between two given nodes. Given that the best heuristic is available for the given situation, the A* algorithm can plan at each step by calculating each cell's cost ($f(n)$). This is done by finding two components of the cell:

$g(n)$ – the path's cost from the start node to the end node.

$h(n)$ – the heuristic cost estimates the cost of the cheapest route from the current node to the end goal. Thus $f(n)$ is calculated for each cell using formula:

$$f(n) = g(n) + h(n)$$

We can use heuristic estimates like Euclidean distance or Manhattan distance for calculating $h(n)$. Since our agent cannot move diagonally, we used the Manhattan distance to calculate estimates. To start with, we instantiated the cost of all the new cells with infinity, and as the agent explores these cells, we update the given cells' cost based on the findings.

Manhattan distance as a heuristic is an admissible heuristic as the cost of reaching the goal node will be higher if the Agent has to move to another cell due to a blocked cell in between (that is Agent needs to turn around it and reach the goal node).

Manhattan Distance: Given two points a and b in a Euclidean space of dimension n, with coordinates (a_1, a_2, \dots, a_n) and (b_1, b_2, \dots, b_n) , the Manhattan distance between them is defined as: $|A-B| = \sum |a_i - b_i|$

Start position:

S	inf	inf	X	Inf
inf	Inf	X	inf	Inf
Inf	X	inf	inf	Inf
inf	inf	Inf	X	inf
X	inf	Inf	X	E

Note: Here:

X= Blocked Cells.

S = Start position.

E = End Goal / Goal position.

Numbers denote the path taken by the algorithm to traverse through the maze. To move to the next position, $g(n)$ will be 1, and $h(n)$ will be 7, hence total cost of 8 will be calculated. Next on moving to the next adjacent cell, $g(n)$ will be 2, and $h(n)$ will be 6. Now the cost of the next cell will be calculated, and in this way, the cost will be calculated till the goal node is reached. In this way, the A* algorithm will calculate the cost of each node sequentially taking the lowest-cost node first from the fringe. The above maze will have a heuristic similar to the below after A* performs the searches.

S	1+7	2+7	X	10+4
1+7	2+6	X	8+4	9+3
2+6	X	6+4	7+3	8+2
3+5	4+4	5+3	X	9+1
X	5+3	6+2	X	E

Priority queue data type in python is used to get the lowest cost node from the current node.

2

Implementation

2.1 The Environment

For implementing the Maze Environment we have used a Matrix with 51x51 dimensions. We have assigned values to each cell[index] such as 1 indicates if the cell is blocked and 0 indicates that is unblocked. For indicating Ghost's presence we subtract '10' from the corresponding cell value. This helps in easily handling 'Multiple ghosts present in a single cell' case and also in maintaining the value of index if the cell is blocked.

Please look at the example for reference.

$$\text{Maze : } \begin{bmatrix} [START] & -10 & 1 & 0 & 1 & 0 \\ 0 & 0 & -10 & 0 & 0 & 1 \\ 1 & -10 & 0 & 1 & 1 & 1 \\ -20 & 0 & 0 & 1 & 1 & 0 \\ 0 & -10 & 1 & 1 & -9 & 0 \\ -19 & 0 & 0 & 0 & 0 & [GOAL] \end{bmatrix}$$

Start: [0,0]

Goal: [N-1,N-1]

Blocked Cells: [0,2],[0,4],[1,5],[2,0],[2,3],[2,4],[2,5],[3,3],[3,4],[4,2],[4,3],[4,4],[5,0].

Ghosts: [0,1],[1,2],[2,1],[3,0],[3,0],[4,1],[5,0],[5,0].

Special Case 1:[3,0] 2 Ghosts in the same cell.

Special Case 2:[5,0] Ghost present in the blocked cell.

Question 1. What algorithm is most useful for checking for the existence of these paths? Why?

When we start exploring the Maze from the Start Cell [0,0], we realize that it turns out to be a Graph traversal problem where there is a Parent-Child relationship between the Start node and corresponding direct traversable nodes. Our requirement was Checking only the existence of the path from the Start to the Goal and making sure that the randomly generated Maze doesn't have

a configuration such that it secludes the Goal[N-1, N-1] with Blocked Cells. For this, we have used Depth First Search (DFS) traversal algorithm. We selected DFS because it is highly efficient with Space, better than its counterpart Breadth First Search (BFS), and matches it in terms of performance. Choosing BFS would have been crucial if we had wanted to find the shortest distance, but that was not the case here.

2.2 Agents

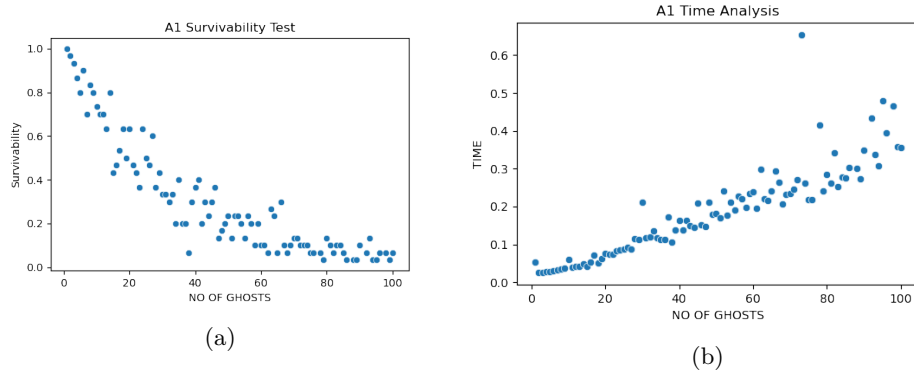
2.2.1 Agent 1 : Plan Once and Execute Blindly

Path Planning and Execution:

Agent 1 only plans the entire route before starting his journey and sticks to the course no matter what happens. Agent 1 is efficient and only works with the shortest possible path. Even if the agent realizes there is a Ghost in the next step, the agent will stick to the same path. As Agent 1 plans the shortest route, we initially implemented BFS to get the shortest path. BFS was working efficiently When the maze was substantially small. But as soon as the trials were run on larger grid sizes, we realized the bottlenecks of using BFS. As BFS works level by level, it does not leave any node unvisited, challenging the time and space constraints. Then we switched to A-Star (A*) algorithm, which handled larger grid sizes better. A-Star (A*) algorithm requires heuristic which provides estimate on how far the current position is from the goal position. We used Manhattan distance as the heuristic in the A* implementation purpose.

Pseudo-Code:

```
# Agent one traversal logic    -> Returns True if Agent 1 reaches the goal cell, and False if Agent 1 dies
def agentOneTraversal():
    a1 = start_pos            # Agent 1 coordinates denoted by this variable
    # Agent One gets the shortest A Star path using Manhattan Distance heuristic, ignoring the ghosts.
    aStarPathDetermined = aStar(my_grid)
    # Agent One traverses in below while loop until either it reaches the ghost, or when Agent 1 gets in the same cell as ghost.
    while (a1 != final_pos):
        nextLocA1 = aStarPathDetermined[a1]
        ghostmovement(my_grid)
        # if my_grid[nextLocA1] == 1:
        #     print('Agent is in Blocked Cell. Some Serious Error !!!!!!!!!!!!!!!')
        if my_grid[nextLocA1] != 0:
            print('Agent not in Open Cell. Ghost Encountered!')
            print(my_grid[nextLocA1])
            return False
        a1 = nextLocA1
    return True
```

Performance and Insights:

S

Figure 2.1: Agent 1 (a) Survivability v/s No. of Ghosts (b) Time Taken (s) v/s No. of Ghosts

- When the number of Ghosts is 0, Agent 1 performs its best since it does not alter its path depending on the ghosts movement. As ghost number increases, Agent 1's survival rate decreases and tends to 0 when Number of ghosts = 80.
- Average time of all successful runs for Agent 1= 0.188

2.2.2 Agent 2 : Plan at Every step

Path Planning and Execution:

Agent 2 is more cautious than Agent 1. It reconsiders and recalculates the route at every step it takes. This process provides Agent with the chance to avoid ghosts. With our earlier experience of A* versus BFS, we decided to implement A* again. We implemented Repeated A*; that is at every step we follow the A* procedure for determining the next action/direction. Since Agent 1 did not have any on-the-go thinking to be done, it was extremely light on time and space but using Repeated A* introduced heavy processing and computing load. We optimized the programming, but that ensued minimal improvement. We started thinking at deeper levels and found the answer in Multiprocessing. We used the 'subprocess' package in python, which drastically changed the performance.

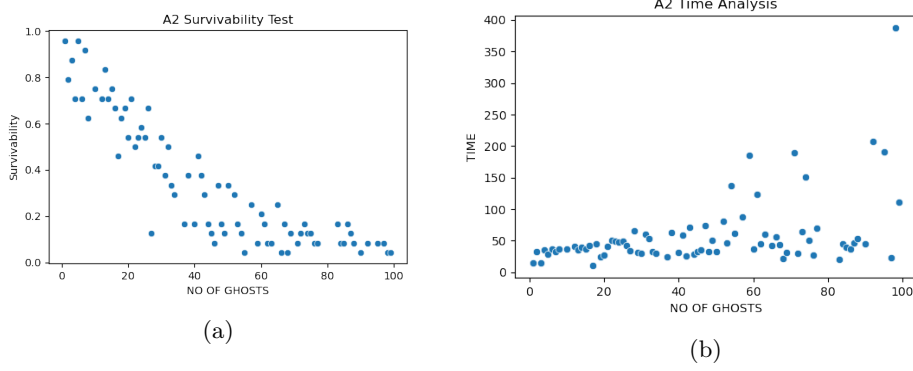
Question. Agent 2 requires multiple searches - you'll want to ensure that your searches are efficient as possible so they don't take much time. Do you always need to replan? When will the new plan be the same as the old plan, and as such you won't need to recalculate?

The maze is randomly generated, i.e., the Blocked and Unblocked Cell Maze Configuration is random. Ghost spawn at Timestamp 0, and their movement is also arbitrary. The random motion of ghosts introduces uncertainty. Hence we

need to re-plan at every step. But even this is not entirely true. What if the ghost moves to a cell which does not affect the agent's movement in the next timestamp? Then the re-routing is pretty redundant and would only add to the computational load decreasing the agent's efficiency. Possibly, the new plan calculated would be the same as the earlier plan if the earlier route is the best one. In our current implementation, complying with the specified requirements recalculates the A* path at every step. This step considers the possibility that the ghost's movement might sketch out a better (shorter) way than it had calculated earlier and may increase the agent's survivability. However, it adds to computational processing.

Pseudo-Code:

```
def agentTwoTraversal(start_pos = (0,0)):
    global agent2PathAndMetric
    a2 = start_pos # Agent 2 coordinates denoted by this variable
    nearestGhostPosition = tuple()
    # Agent 2 travels until it reaches the goal position, or until it gets in same coordinate as ghost position
    while (a2 != final_pos):
        # Agent 2 gets a new A Star path based on current location in each step, and checks a path which is free of ghost
        aStarPathDetermined = aStar(my_grid, 1, a2[0], a2[1])
        if len(aStarPathDetermined) == 0: # True if A Star path not present to goal node
            # Agent checks for the nearest ghost using BFS algorithm
            nearestGhostPath = breadth_first_search(my_grid, 1, a2[0], a2[1]) # Returns list of path to ghost
            # If the nearestGhostPath returned does not contain next cell, below 'if' will be triggered. Acts as safety mechanism
            if len(nearestGhostPath) == 1:
                return False
            nearestGhostPosition = nearestGhostPath[1] # This will be the next cell from the agent's current cell
            nextLocA2 = getAwayFromGhost(a2, nearestGhostPosition) # Passes current Agent 2 location and the next Path that Agent will
            # have to take to get to the nearest ghost
            # if nextLocA2 == a2: # Included to go towards ghost if there are no other paths present. Implemented this as Agent 2 cannot
            # stay at same location
            # nextLocA2=nearestGhostPosition
            if not nextLocA2 or my_grid[nextLocA2[0],nextLocA2[1]] < 0: # Returns False if there is no path or if index is out of bounds
                return False
        else:
            # If A Star path is present without ghost in path, agent will follow the A Star path
            nextLocA2 = aStarPathDetermined[a2]
        # Storing data of Agent 2 in path
        currCellToNextCellDirection = findDirection(a2, nextLocA2)
        agent2PathAndMetric[(a2[0], a2[1], currCellToNextCellDirection)] = False
        # Movement of ghost initiated
        ghostmovement(my_grid)
        # Returns False if agent steps in same cell as ghost
        if my_grid[nextLocA2] != 0:
            return False
        a2 = nextLocA2
    return True
```

Performance and Insights:

S

Figure 2.2: Agent 2 (a) Survivability v/s No. of Ghosts (b) Time Taken (s) v/s No. of Ghosts

- Agent 2 takes far longer time than Agent 1 to complete the maze. Since the Agent checks for the best path at each step, time required to complete the maze was exponentially greater than Agent 1's time at similar settings.
- Average time of all successful runs for Agent 2 = 58.238

2.2.3 Agent 3 : Forecasting on Agent 2's knowledge

Path Planning and Execution:

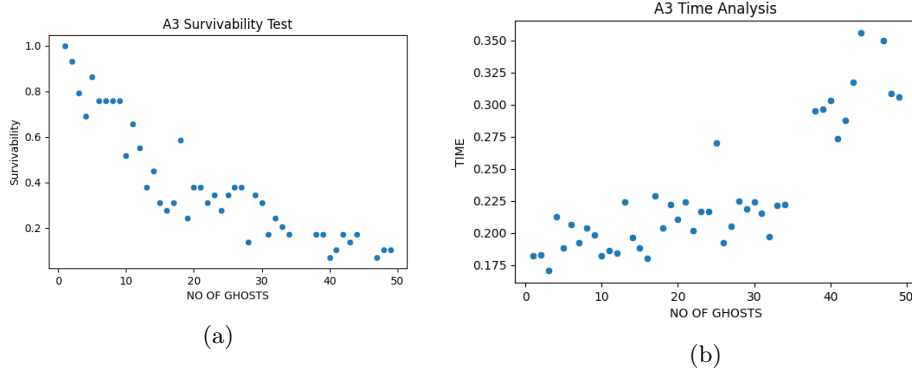
Agent 3 is an amped up Agent 2 which has the ability to forecast. It reconsiders and recalculates the route at every step it takes. This process provides Agent with the chance to avoid ghosts. With our earlier experience of A* versus BFS, we decided to implement A* again. We implemented Repeated A*; at every step, we follow the A* procedure for determining the next action/direction. Since Agent 1 did not have any on-the-go thinking to be done, it was extremely light on time and space but Using Repeated A* introduced heavy processing and computing load. We optimized the algorithm, but that ensued minimal improvement. We started thinking at deeper levels and found the answer in Multiprocessing. We used the 'subprocess' package in python, which drastically changed the performance. Agent3 uses the experience of Agent2. Agent3 works as per the following: Agent2 is ran for multiple runs and it's position level data is collected and stored. When Agent3 reaches on a given position, it is checked if that cell is already explored. If the cell has no history in Agent2: It runs Astar and takes the move (mimicks Agent2 Behaviour) If that cell is not explored, It checks the valid directions it can move. For all the valid directions, survivability are compared and the max survivability direction is chosen and movement is made. If cell is explored, This suggests it can be in loop or it is here due to ghost movement. So, it mimics Agent2's behaviour OR it stays in the place in case of no valid direction available.

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?

From the requirements, it is clear that Agent3 should have better success rates than Agent2 since it has Agent2's experience and knowledge. For each Number of ghosts, we ran Agent2 25 times for each maze configuration and collected the movement data for each cell visited. Using this data, we are finding the next best possible move based on survivability from each cell, which acts as a heuristic to decide for Agent3. We are not running Agent2 on runtime at every step, as it will be highly costly. If Agent3 does not get the data from agent2's data collected, it runs, simulates Agent2 at runtime, and takes the decision. If it does not get the successful path to the goal, it waits at the cell OR takes a random move and then continues finding a new approach. If it does not get the successful path to the goal, it waits at the cell OR takes a random move and then continues finding a new direction as per its rules. Having no successful path may not necessarily mean that it can't succeed. It is possible after staying in place for a few timestamps. It may be able to find the path according to ghost movement.

Pseudo-Code:

```
def agentFourTraversal():
    global a4PathTaken
    a4 = start_pos # Agent 4 coordinates denoted by this variable
    aStarPathDetermined = aStar(my_grid, 0, a4[0], a4[1])
    # A Star path search without considering the ghost to get the shortest path possible to the end
    strike, visibility = 0, 3
    while (a4 != final_pos):
        if a4 in aStarPathDetermined:
            # If A Star path is present for the agent from the current cell, follow the path
            nextLocA4 = aStarPathDetermined[a4]
        else:
            # Find A Star path without considering the ghost, to get the shortest path possible to the end
            aStarPathDetermined = aStar(my_grid, 0, a4[0], a4[1])
            nextLocA4 = aStarPathDetermined[a4]
        # Check if any ghost is present in aStar's Visibility (+ 1 more depth) cells
        ghostPresentNearVisibility = checkOpenCellsForAgentFour(a4, aStarPathDetermined, visibility)
        if ghostPresentNearVisibility:
            # If ghost present in/near A Star path from the current cell, increments a counter to track consecutive ghost encounters
            strike += 1
            if strike==1:
                a4GhostPositionNearby = checkAdjacentCoordinatesForGhost(a4) # List of ghost coordinates at adjacent cells
                if a4GhostPositionNearby != []: # If there is a ghost in nearby cell
                    a4AllowedDirections = [1,2,3,4]
                    # Get list of invalid adjacent directions, which will lead to agent going out of environment
                    invalidDirections = getInvalidAdjacentDirectionsToGoTo(a4)
                    for i in invalidDirections:
                        a4AllowedDirections.remove(i)
                    placeholderA4GhostPositionNearby = a4GhostPositionNearby[:]
                    # Will remove cell coordinates/directions which will take the agent nearer to the ghost
                    for i in placeholderA4GhostPositionNearby:
                        restrictedDirection = findDirection(a4, i)
                        a4AllowedDirections.remove(restrictedDirection)
                    placeholderA4AllowedDirections = a4AllowedDirections[:]
                    # To remove blocked cells from the list of directions that can be taken
                    for i in placeholderA4AllowedDirections:
```

Performance and Insights:

S

Figure 2.3: Agent 3 (a) Survivability v/s No. of Ghosts (b) Time Taken (s) v/s No. of Ghosts

- Agent 3 performs better than Agent 2 in terms of survivability since Agent 3 uses data collected from Agent 2 to make decisions for each cell. Agent 3's survival rate decreases and tends to 0 when Number of ghosts = 80.
- Average time of all successful runs for Agent 3 = 0.23

2.2.4 Agent 4 : Plan only when needed

Path Planning and Execution:

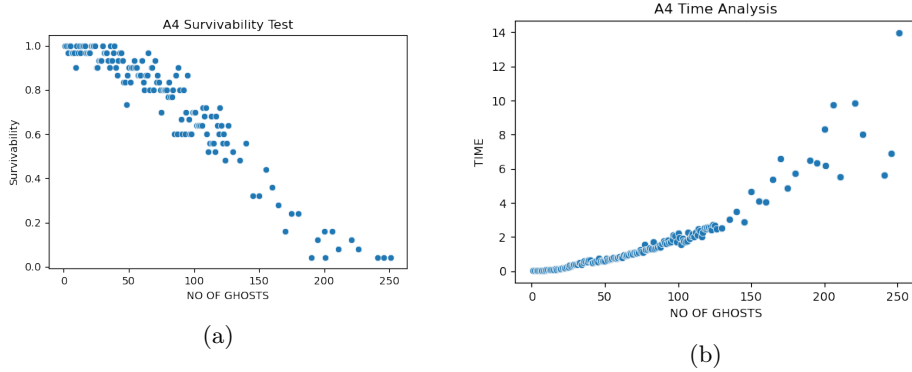
Question. Agent 4 is a free strategy - develop your own agent to solve the problem that beats Agent 3. How can you balance intelligence and efficiency?

For Agent 4, we wanted to balance intelligence and efficiency. Which is why Agent 4 implementation is quite different from all the above agents. We wanted to balance the previous Agent's strategies, and then came up with the idea of visibility (set to 3). Until now, at every step, the Agents (except agent 1) were re-calculating its route from his current position to the goal node. Then it takes an informed action on whether to move on to this path. This increased the computational processing to a great extent. Hence, we made our algorithm where we first calculated A* and get the visibility attribute. Visibility allows the Agent to see 3 steps ahead (including the Adjacent cells of these potential positions). Initially, A* path of the maze without taking ghosts into consideration is calculated, and then Agent checks its Visibility vicinity (in the next A* path cells) at each step. Agent moves ahead if no ghosts are present in its vicinity. For the first instance, when the Agent encounters a ghost in its vicinity, the Agent will not move ahead and stay in the same position for the next timestamp. If, in the next timestamp, the ghost moves closer to the ghost, then the Agent runs

away from the ghost. To implement this efficiently, we introduced a flag named 'strike' which increments by 1 whenever a ghost is encountered in vicinity. If `strike==1` and there are no ghosts in the adjacent cells, agent stays in the same cell hoping for the ghost to move away from the shortest path. Otherwise, agent moves away from the nearest ghost. Strike is set to 0 when the ghost moves away from vicinity. This process is repeated till the Agent reaches the goal node or one of the ghosts gets into the same cell as agent's.

Pseudo-Code:

```
def agentFourTraversal():
    global a4PathTaken
    a4 = start_pos # Agent 4 coordinates denoted by this variable
    aStarPathDetermined = aStar(my_grid, 0, a4[0], a4[1])
    # A Star path search without considering the ghost to get the shortest path possible to the end
    strike, visibility = 0, 3
    while (a4 != final_pos):
        if a4 in aStarPathDetermined:
            # If A Star path is present for the agent from the current cell, follow the path
            nextLocA4 = aStarPathDetermined[a4]
        else:
            # Find A Star path without considering the ghost, to get the shortest path possible to the end
            aStarPathDetermined = aStar(my_grid, 0, a4[0], a4[1])
            nextLocA4 = aStarPathDetermined[a4]
            # Check if any ghost is present in aStar's Visibility (+ 1 more depth) cells
            ghostPresentNearVisibility = checkOpenCellsForAgentFour(a4, aStarPathDetermined, visibility)
            if ghostPresentNearVisibility:
                # If ghost present in/near A Star path from the current cell, increments a counter to track consecutive ghost encounters
                strike += 1
                if strike==1:
                    a4GhostPositionNearby = checkAdjacentCoordinatesForGhost(a4) # List of ghost coordinates at adjacent cells
                    if a4GhostPositionNearby != []: # If there is a ghost in nearby cell
                        a4AllowedDirections = [1,2,3,4]
                        # Get list of invalid adjacent directions, which will lead to agent going out of environment
                        invalidDirections = getInvalidAdjacentDirectionsToGoTo(a4)
                        for i in invalidDirections:
                            a4AllowedDirections.remove(i)
                        placeholderA4GhostPositionNearby = a4GhostPositionNearby[:]
                        # Will remove cell coordinates/directions which will take the agent nearer to the ghost
                        for i in placeholderA4GhostPositionNearby:
                            restrictedDirection = findDirection(a4, i)
                            a4AllowedDirections.remove(restrictedDirection)
                        placeholderA4AllowedDirections = a4AllowedDirections[:]
                        # To remove blocked cells from the list of directions that can be taken
                        for i in placeholderA4AllowedDirections:
                            for i in placeholderA4AllowedDirections:
                                nextCell = getNextCoordinatesToMoveTo(a4, i)
                                if (not checkForOpenPosition(nextCell)):
                                    # checkForOpenPosition(nextCell) will return False if the cell is blocked.
                                    a4AllowedDirections.remove(i) # If blocked, Agent cant move to this direction
                                if a4AllowedDirections == []: # Stay at same location as Allowed Direction from checks is 0
                                    nextLocA4 = a4
                                else: # Choose random direction from allowed directions
                                    directionToMove = np.random.choice(a4AllowedDirections)
                                    nextLocA4 = getNextCoordinatesToMoveTo(a4, directionToMove)
                            else:
                                nextLocA4 = a4
                                # Stay at same position as ghost is not at immediate next step. Ghost can go away to another direction
                        else:
                            # Move away from the AStar path as ghost encountered nearby
                            nextLocA4 = getAwayFromGhost(a4, nextLocA4)
                    else:
                        strike = 0
                        nextLocA4 = aStarPathDetermined[a4]
                        # Since no ghost in visibility. Agent will go on with AStar path.
                # Agent dies if agent is present with the ghost
                if my_grid[nextLocA4] != 0:
                    return False
                a4 = nextLocA4
                ghostmovement(my_grid) # Movement of each ghost present in my_grid
                a4PathTaken.append(a4)
    return True
```


Performance and Insights:

S

Figure 2.4: Agent 4 (a) Survivability v/s No. of Ghosts (b) Time Taken (s) v/s No. of Ghosts

- Agent 4 performs really well as compared to the other agents. Agent 4's survivability is more than 50% where other agents are nearing 0. Its survivability tends to 0 when Number of ghosts is near 250!
- Average time of all successful runs for Agent 4 = 1.83

2.2.5 Agent 5 : Plan with impaired sight**Path Planning and Execution:**

Question. Redo the above, but assuming that the agent loses sight of ghosts when they are in the walls, and cannot make decisions based on the location of these ghosts. How does this affect the performance of each agent? Build an Agent 5 better suited to this lower-information environment. What changes do you have to make in order to accomplish this?

For Agent 5, we created an algorithm that does not have information about the ghosts in the wall. When we stacked this idea with other Agents, we assessed that only Agent 4's algorithm would behave differently in this restricted environment. Agent 1 is a fearless agent who will not consider ghosts while planning the path. Agent 2 performs A* searches repeatedly to get a path without ghosts until the goal node. Since A* search ignores the blocked path anyway, it does not consider the cells with ghosts. Agent 3 uses Agent 2's data to move to the cell, and if the path is blocked, it behaves like Agent 2 and ends up searching for an A* path without ghosts. Hence all Agents 1, 2, and 3 will behave the same in this lower-information environment.

As per our initial analysis before implementing the algorithm for Agent 5, we believed that Agent 4's algorithm would work well even in this lower-information environment as the possibility of ghosts moving into and through the blocked

cells was thin (considering 50% probability whenever ghost's random movement was supposed to be through the blocked walls). But in the results that followed, we found that Agent 5's average survivability was not even better than Agent 1's. These results were surprising as Agent 4 performed 125% better than Agent 1. Still, on running the tests, the result indicated that Agent 4's algorithm does not perform well when the information about the ghost is reduced. Agent 5 performed the worst when compared to other agents in terms of survivability.

Pseudo-Code:

```
def agent3Traversal(nr_of_ghosts):
    a3 = start_pos          # Agent 3 coordinates denoted by this variable.
    a3CellExplored = []     # Storing each position taken by Agent3, so that we can decide if it is forming a loop in path or not?
    while (a3 != final_pos):
        print(a3)
        if a3 in a3CellExplored:      # If Agent3 is again visiting already visited cell, take the move as per A star, else stay in place.
            aStarPathDetermined = aStar(my_grid, 0, a3[0], a3[1])
            nextCell = aStarPathDetermined[a3[0],a3[1]]
            if my_grid[nextCell[0]][nextCell[1]] != 0:
                a3 = a3              # Stay in place if the nextCell has ghosts.
            else:
                a3 = nextCell
        else:
            a3AllowedDirections = [1,2,3,4]      # Initiated a direction list, invalid or blocked directions will be removed from this list.
            invalidDirections = getInvalidAdjacentDirectionsToGoTo(a3)      # Gets list of invalid adjacent directions, which will lead to agent going out of environment
            print('invalidDirections : '+str(invalidDirections))
            for i in invalidDirections:
                print('Removing value '+str(i)+' from a3AllowedDirections '+str(a3AllowedDirections))
                a3AllowedDirections.remove(i)      # Removes invalid directions from allowed directions for the agent from current position
            nearbyGhostPositions = checkAdjacentCoordinatesForGhost(a3)
            if len(nearbyGhostPositions) != 0:
                for pos in nearbyGhostPositions:
                    a3AllowedDirections.remove(findDirection(a3, pos))
            placeholderAllowedDirections = a3AllowedDirections[:]
            for dir in placeholderAllowedDirections:
                nextCell = getNextCoordinatesToMoveTo(a3, dir)
                if (not checkForOpenPosition(nextCell)):
                    a3AllowedDirections.remove(dir)

            print("Allowed Directions "+ str(a3AllowedDirections))
            directionToTake = moveAsPerAgent2(nr_of_ghosts, start_pos, a3AllowedDirections)      # Take one allowedDirection with the maximum survivability.

def agent3Traversal(nr_of_ghosts):
    a3 = start_pos          # Agent 3 coordinates denoted by this variable.
    a3CellExplored = []     # Storing each position taken by Agent3, so that we can decide if it is forming a loop in path or not?
    while (a3 != final_pos):
        print(a3)
        if a3 in a3CellExplored:      # If Agent3 is again visiting already visited cell, take the move as per A star, else stay in place.
            aStarPathDetermined = aStar(my_grid, 0, a3[0], a3[1])
            nextCell = aStarPathDetermined[a3[0],a3[1]]
            if my_grid[nextCell[0]][nextCell[1]] != 0:
                a3 = a3              # Stay in place if the nextCell has ghosts.
            else:
                a3 = nextCell
        else:
            a3AllowedDirections = [1,2,3,4]      # Initiated a direction list, invalid or blocked directions will be removed from this list.
            invalidDirections = getInvalidAdjacentDirectionsToGoTo(a3)      # Gets list of invalid adjacent directions, which will lead to agent going out of environment
            print('invalidDirections : '+str(invalidDirections))
            for i in invalidDirections:
                print('Removing value '+str(i)+' from a3AllowedDirections '+str(a3AllowedDirections))
                a3AllowedDirections.remove(i)      # Removes invalid directions from allowed directions for the agent from current position
            nearbyGhostPositions = checkAdjacentCoordinatesForGhost(a3)
            if len(nearbyGhostPositions) != 0:
                for pos in nearbyGhostPositions:
                    a3AllowedDirections.remove(findDirection(a3, pos))
            placeholderAllowedDirections = a3AllowedDirections[:]
            for dir in placeholderAllowedDirections:
                nextCell = getNextCoordinatesToMoveTo(a3, dir)
                if (not checkForOpenPosition(nextCell)):
                    a3AllowedDirections.remove(dir)

            print("Allowed Directions "+ str(a3AllowedDirections))
            directionToTake = moveAsPerAgent2(nr_of_ghosts, start_pos, a3AllowedDirections)      # Take one allowedDirection with the maximum survivability.
```

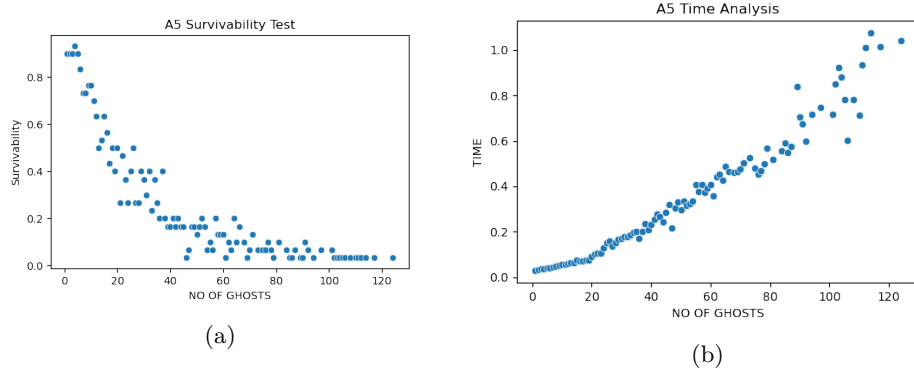
Performance and Insights:

Figure 2.5: Agent 5 (a) Survivability v/s No. of Ghosts (b) Time Taken (s) v/s No. of Ghosts

- Agent 5 underperforms all the Agents on which we ran the same simulations. This may be because of the lower visibility available for Agent 5. It is likely that the ghosts entered the blocked cells, which were not visible to Agent 5, and were able to get out (considering the probability factor) only after a few turns, and it is too late for Agent 5 to maybe take any action by then. Agent 5's survival rate decreases and tends to 0 when the Number of ghosts = 100.
- Average time of all successful runs for Agent 5 = 0.369

3

Analysis

3.1 Question 1

Graphs comparing the performance of each agent for different numbers of ghosts. When does survivability go to 0?

Agent	Avg Survivability	Survivability Range*	Avg Survival Time
Agent1	0.307	0.29, 0.326	0.188
Agent2	0.341	0.319, 0.364	58.238
Agent3	0.392	0.364, 0.42	0.23
Agent4	0.75	0.735, 0.765	1.83
Agent5	0.292	0.272, 0.311	0.369

*Survivability Range is considering 95% Values of Normal Distribution of Avg Survivability

Avg Survivability = No of Success / No. of experiments

Survivability Range:

$[Avg\ Survivability - (1.96 / (2 * \sqrt{N})) , Avg\ Survivability + (1.96 / (2 * \sqrt{N}))]$

- All the agents' survivability will decrease from one to zero as the number of ghosts increases. Also, the average time taken to reach the goal increases as the number of ghosts increases.
- Agent 1's average survivability is the second least. As Agent 1 plans only at the start without considering any ghost and executes that plan, it cannot make changes in the path even when a ghost enters the path.
- Agent 2's average survivability is better than Agent1's. Agent 2 can re-plan at each step without considering ghosts and executes that plan. As it can re-plan at every step, it will better understand the current ghost positions and thus will have better chance of survival (more than Agent 1's).

- Agent 3's average survivability is better than Agent 2's. Agent 3 can forecast the future by building on the knowledge of Agent 2 and staying in a particular position. The average time of Agent3 is more as it has to run simulations of Agent 2 for each index for which the data is already not present in Agent 2's data collection.
- Agent 4's average survivability is more than Agent 3's. In fact it outperformed all the other agents and had a survival rate of approx 50% at 100 ghosts. At similar number of ghosts, all the other agents were nearing 0 in terms of survivability. This results must be because of Agent 4's strategy, which considers visibility and strikes criteria to reach the goal. Strike criteria helps Agent 4 to assess and take the path which avoids clashing with ghosts.
- Time taken by Agent 4 is less than Agent 3, as it does no simulations of prev agents as done by Agent 3. Also, Agent 4 only re-plans if it encounters a ghost in its visibility criteria of the estimated path, which reduces the re-planning time, and thus Agent4 takes less execution time than Agent 3.
- Agent 5's survivability was the least. This must be because of the lower visibility available for Agent 5. It is likely that the ghosts entered the blocked cells before they entered the Agent 5's visibility, and were able to get out (considering the probability factor) only after a few turns, by which time it was too late for Agent 5 to take any action.

3.2 Question 2

Computational bottlenecks you ran into and how you dealt with them. To get this working with a large number of ghosts at a large dimension is going to take some work?

- For Agent 1. Initially, the design was to implement BFS, the shortest path algorithm, but it has exponential space complexity. While the A* algorithm runs are computationally less intensive and give the shortest distance, we went for implementing A* for the shortest path algorithm.
- Theoretically, and by doing runs, we are proving that the A* also performs better than BFS in our case. Time taken by Agent 3 to execute serially was huge. The computational bottleneck in Agent 2 led to a design change for Agent 3.
- We did not run Agent 2 at runtime in Agent 3 every time. Agent 2's function was paralyzed using the subprocess module.
- The MultiProcessing function spawns X process of Agent 2 for X no. of ghosts, with each process collecting data for each no. of ghosts. It waits until all X processes are completed. The race condition of writing metrics in the file for data collection is avoided by writing the metrics of each process in a separate file and merging it later.

- For Agent 4: To reduce complexity, introduced visibility attribute to check only to a limited length of 3+1. Visibility allows the Agent to see 3 steps ahead and the Adjacent cells of these potential positions. Thus, instead of the whole path, only a particular set of cells at a chosen depth are considered for making the decision.
- For Agent 4 and Agent 5: While collecting data for pattern recognition, we realized that Agent 4 and Agent 5 can withstand a higher number of ghosts which is why more iterations were needed to reach their threshold. With the increased difficulty (number of ghosts), the time required for each iteration grew substantially. Hence we had to increment the number of ghosts by 5 instead of 1.
- We were planning to use the SciPy array to store matrices as it has better space optimization than the NumPy array. Still, since the grid scale was only 51x51, there was not much difference in practical implementation.

3.3 Question 3

How did Agents 1, 2, and 3, compare? Was one always better than others, at every number of ghosts?

- All the agents' survivability will decrease from one to zero as the number of ghosts increases. Also, the average time taken to reach the goal increases as the number of ghosts increases. Agent 1's average survivability is the least as it plans only at the start without considering any ghost and executes that plan.
- Agent 2's average survivability is better than Agent 1's. Agent 2 can re-plan without considering ghosts and executes that plan at each step. As it can re-plan at every step, it will better understand the current ghost positions and thus will have to survive more than Agent 1.
- Agent 3's average survivability is better than Agent 2's. Agent 3 can forecast the future by building on the knowledge of Agent 2 and staying in a particular position. The average time of Agent 3 is more as it has to run simulations of Agent 2 for each index for which the data is already not present in Agent 2's data collection. This makes it slower compared to Agent 2 and Agent 1.
- Agent 4's average survivability is the highest. As discussed above, Agent 4 has special implementation which makes handling cases more efficient.
- Agent 5's average survivability is less than Agent 4, as it loses information about the ghosts in the blocked cell. Its execution time is almost equivalent to that of Agent 4.