Date: 14th October, 2022

CS 520 : Introduction to Artificial Intelligence

Project 1: Ghosts in the Maze

TEAM MEMBERS:
Naga Vamsi Krishna Bulusu (nb847)
Sakshi Ghadigaonkar (ssg156)

Project Summary: A search algorithm is an algorithm designed to solve a search problem. Search algorithms work to retrieve information stored within a particular data structure, or calculated in the search space of a problem domain, with either discrete or continuous values.
This project focuses on the implementation and importance of search algorithms. Given a maze-like square grid (size:51x51) with probability 0.72 unblocked cells and probability 0.28 blocked filled with ghosts, we find the optimal path to the goal in four scenarios each having a different agent.

Programming language: JAVA

Environment: We created a Square Maze with the given probabilities of unblocked and blocked cells using a ***2D character matrix***. In the maze, *unblocked cells* are represented by the character '✱' and *blocked cells* are represented by the character 'X'. Since the maze is generated using a random function, we need to validate our maze before passing it to your agent code. For *validating our maze* we are using *Depth First Search* (DFS) *algorithm* since our aim here is to check if a path exists or not. DFS traverses in a depth-wise fashion so we are reducing the search space by exploring only the nodes that move us closer to the goal. However, in the worst case we might have to traverse all the nodes which is similar to Breadth First Search(BFS).

Time Complexity for traversing the maze using DFS & BFS is given by $O(n^2)$ where n is the size of the maze.
In the Best Case scenario, **Space Complexity** of DFS is better than BFS since BFS traverses all the levels one by one.

Agent: The start point for all agents is given by the coordinates (0,0) and the goal is at coordinates (50,50). The agent moves in the cardinal directions only in unblocked cells(✱) with the boundary of the maze.

Ghosts: The ghosts are spawned using a random function. We are keeping track of the *starting locations* of all the ghosts using a ***List(pair) data structure***. As mentioned in the problem statement, at every timestep the ghosts move to one of its neighbouring cells. If the neighbour picked by the ghost is blocked then the ghost decides whether to move or stay in the same cell by a probability of 0.5. The movement of the ghosts is also given by a random function.

👤	*	*	*	X	*	*
X	*	X	*	X	*	*
*	*	*	*	*	*	X
X	*	*	X	*	X	*
X	*	*	*	*	*	X
*	X	*	X	*	*	*
X	*	*	*	X	*	G

SAMPLE 7x7 MAZE

Description of AGENT -1:

Agent 1 chooses the shortest path to reach the goal. This agent is not concerned about the ghosts in his path which means that it only plans a path once and doesn't change its path if the path contains any ghosts.

Workflow:

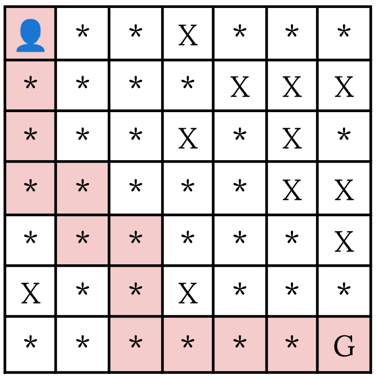
1. Agent-1 uses *Breadth First Search* (BFS) for finding the shortest path from the start node to the goal since we are doing a level-order traversal.
2. We created a function called '*startAgentOne*' to which we are passing parameters like the Environment, the start node and a boolean flag value '*moveGhost*' to enable ghosts movement.
3. We are maintaining a *hash set* to keep track of the visited nodes and a queue for exploring the neighbouring nodes.
4. After moving the ghosts we check if the agent is alive. If the agent is alive then we check if the agent has reached the goal or not.

5. If the agent has reached the goal, we *reconstruct the path* from the goal to the start node.

Pseudo Code for ‘startAgentOne’ function:

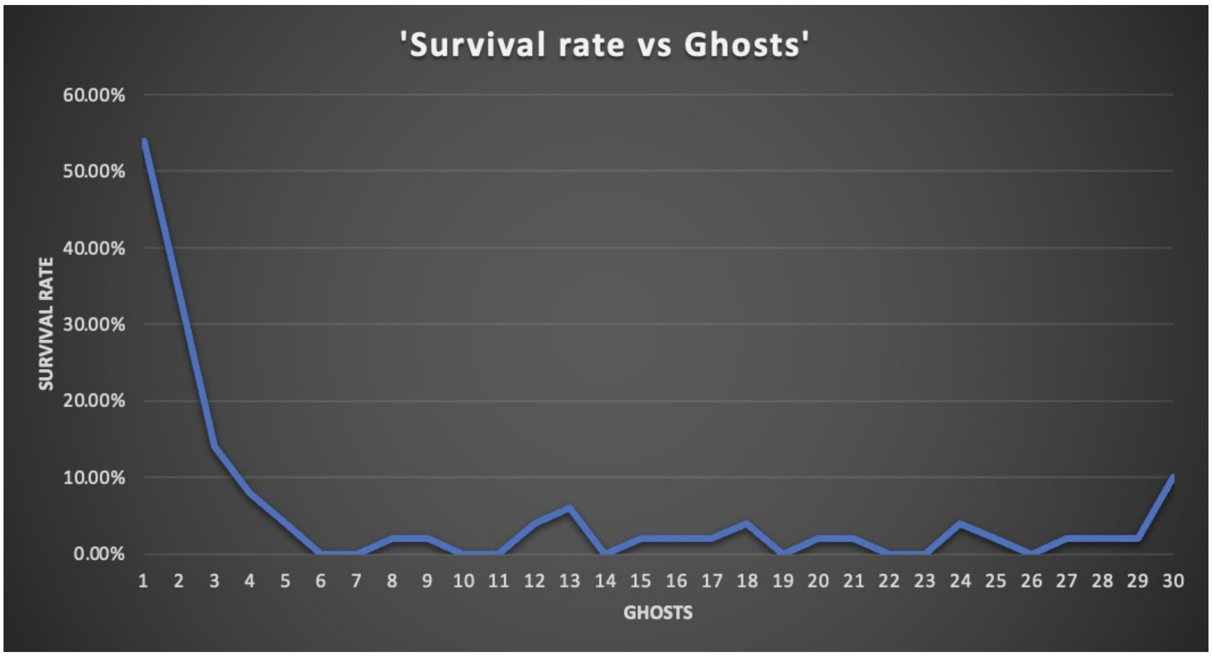
```
Function startAgentOne(environment, start, boolean moveGhost){
    Create HashSet 'visited';
    Create LinkedList 'queue';
    add start node to visited & queue;
    //we are using environment.parent to keep track of the parent child references
    environment.getParent().put(start, null);
    Set reachedGoal == false;
    while(queue is not Empty){
        for i=1 to i< queue.size();i++{
            pop the top node from queue;
            if moveGhost is true: move ghost;
            //check if agent is alive
            if(moveGhost && environment.getGhosts(contains node)) return false;
             // Check if agent reached Goal
             if(node coordinates== grid size-1){
                reachedGoal = true;
            }
            visitNeighbours(environment, visited, pair, queue);
        }
    }
    if agent reachedGoal then reconstructPath(Environment);
    return reachedGoal;
}
```

Representation of AGENT-1:



Data Analysis of AGENT-1:

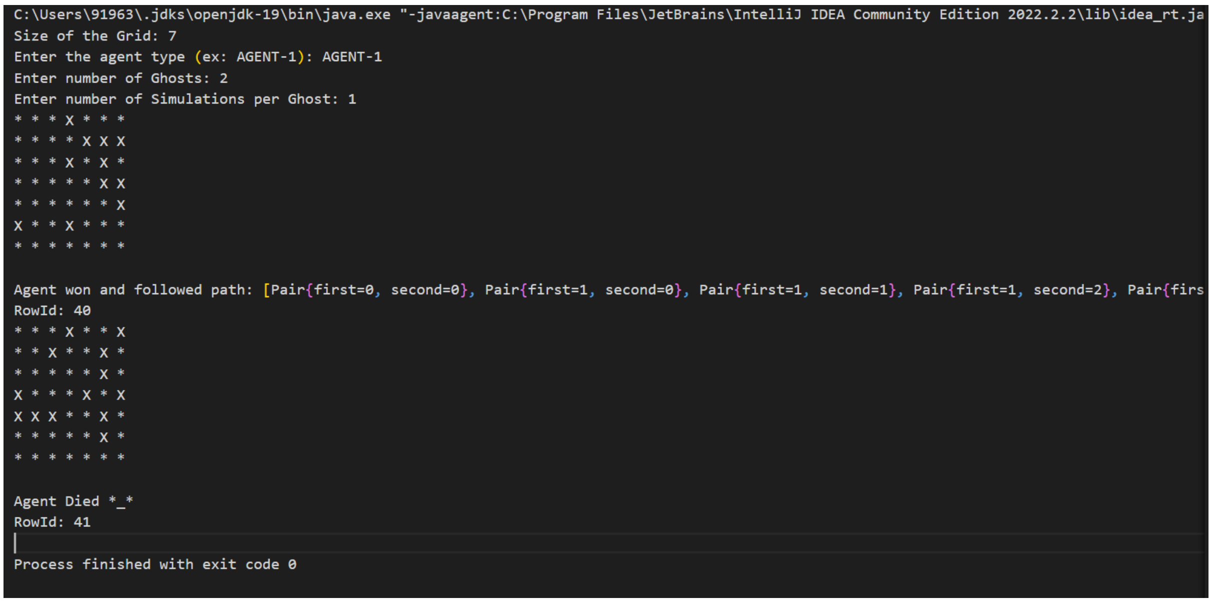
For populating the data, we are running the simulate function on parameters like the environment, number of simulations. We are calculating our agent's survivability when there are 'n' numbers of ghosts where n is an integer between 1 and 30 for 50 simulations.



Pseudo Code for the Simulate function of AGENT-1:

```
Function Simulate(environment, start){
    if(startAgentOne(environment, start,moveGhost: true)){
       print: "Agent won and followed the path:" + grid;
       return 1;
    else{
       print : "Agent died *_*";
       return 0;
    }
}
```

Output of AGENT-1:



```
C:\Users\91963\.jdks\openjdk-19\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2022.2.2\lib\idea_rt.ja
Size of the Grid: 7
Enter the agent type (ex: AGENT-1): AGENT-1
Enter number of Ghosts: 2
Enter number of Simulations per Ghost: 1
* * * X * * *
* * * * X X X
* * * X * X *
* * * * * X X
* * * * * * X
X * * X * * *
* * * * * * *

Agent won and followed path: [Pair{first=0, second=0}, Pair{first=1, second=0}, Pair{first=1, second=1}, Pair{first=1, second=2}, Pair{firs
RowId: 40
* * * X * * X
* * X * * X *
* * * * * X *
X * * * X * X
X X X * * X *
* * * * * X *
* * * * * * *

Agent Died *_*
RowId: 41

Process finished with exit code 0
```

Description of AGENT -2:

Agent-2 takes into consideration if there are any ghosts in its path and recalculates the path at every move depending on the information available to it. For getting the path, we are recursively calling Agent-1 and checking if there are any ghosts in the path. If there are ghosts in the path then, at every timestep we again recursively Agent-1 and recalculate a new path. Internally ,we are using the BFS algorithm for Agent-2 as well since we are calling Agent-1 for getting the path. The aim here is to make 'intelligent' decisions before making a new move.

Workflow:

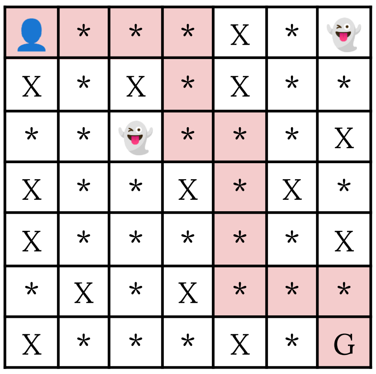
1. We need to pass a path to our Agent-2. So we first call Agent-1 to get this path.
2. If Agent-2 doesn't get a path from Agent-1 then it will call Agent-1 one more time. If there is no path then our Agent-2 will *move away from the nearest ghost*.
3. If we get a path from Agent-1, our Agent-2 takes a step forward. Simultaneously, invoke our moveGhost function and check if the agent is alive.
4. If our agent is alive, then we recursively call Agent-2 until our Agent-2 reaches the goal or our Agent-2 dies.
5. If our Agent-2 has reached the goal then we reconstruct the path from Goal node to start node.
6. For the move away from the nearest ghost function, we calculate the *Manhattan distance* from the current node where our agent is to the nearest ghost node.

Pseudo Code for ‘startAgentTwo’ function:

```
Function startAgentTwo(environment,current, boolean pathExists){
    moveGhosts;
    //check if agent is alive
    if(environment.ghosts.contains(current)) return false;
    // check if agent reached goal
    if(node coordinates== grid size-1) return true;
    // check if ghost is in path
    for(Pair p : environment.path){
        if (environment.ghosts.contains(p)) {
            pathExists = false;
            break;
        }
    }

    //If there is no path then choose next available path
    if(!pathExists) {
        pathExists = agentOne.startAgentOne(environment, current,
false);
    }
    if (pathExists is true):
        moveAhead and set current to the new node in the path;
    else:
        moveAwayFromNearestGhost and set current to the new node;
    return startAgentTwo(environment,current,pathExists);
}
```

Representation of AGENT-2:



👤	*	*	*	X	*	👻
X	*	X	*	X	*	*
*	*	👻	*	*	*	X
X	*	*	X	*	X	*
X	*	*	*	*	*	X
*	X	*	X	*	*	*
X	*	*	*	X	*	G

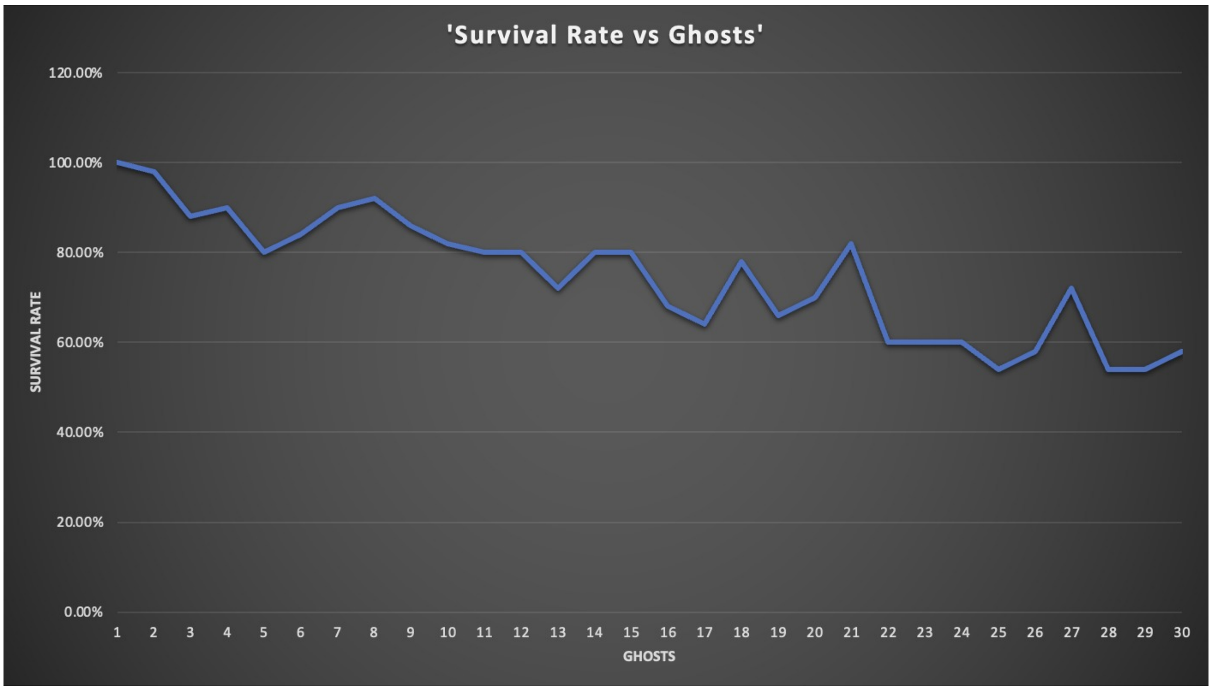
a) Initial path passed to Agent-2 by Agent-1 with two ghosts in the maze.

*	👤	*	*	X	*	*
X	*	X	*	X	*	👻
*	*	*	👻	*	*	X
X	*	*	X	*	X	*
X	*	*	*	*	*	X
*	X	*	X	*	*	*
X	X	*	*	*	*	G

b) Agent-2 moves forward and the ghosts change their position. Now one ghost is the path hence our Agent-2 will calculate a new path.

Data Analysis of AGENT-2:

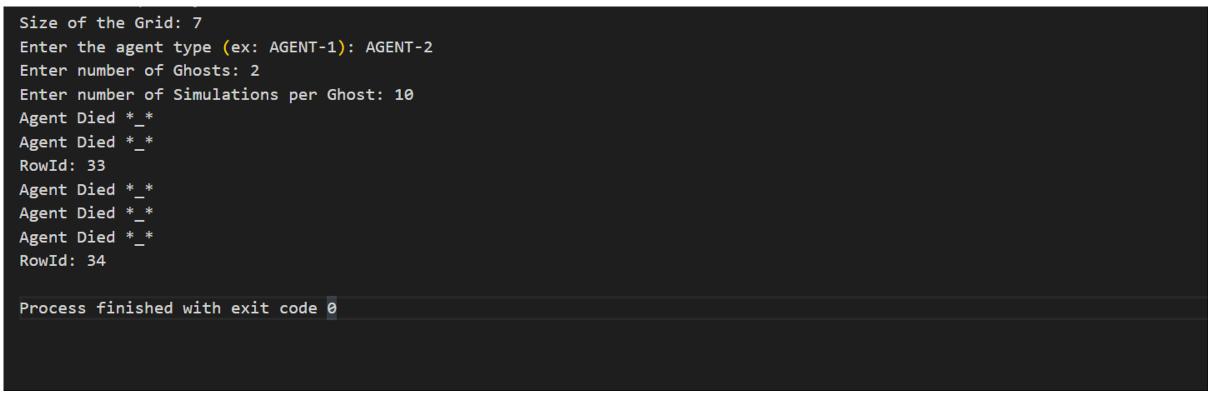
We are calculating our agent's survivability when there are 'n' numbers of ghosts where n is an integer between 1 and 30 for 50 simulations.



Pseudo Code for the Simulate function of AGENT-2:

```
Function simulate(environment,start){
    if (!pathExists) then set pathExists as false;
    if (reachedGoal) then set reachedGoal as true
    if(reachedGoal) return 1;
    else print ("Agent Died *_*");
    return 0;
}
```

Output of AGENT-2:



```
Size of the Grid: 7
Enter the agent type (ex: AGENT-1): AGENT-2
Enter number of Ghosts: 2
Enter number of Simulations per Ghost: 10
Agent Died *_*
Agent Died *_*
RowId: 33
Agent Died *_*
Agent Died *_*
Agent Died *_*
RowId: 34

Process finished with exit code 0
```

Description for AGENT-3:

For every step,Agent-3 calculates the success rate for every possible move it can make. Agent-3 internally makes use of the behaviour of Agent-2 for calculating the success rate for the possible neighbouring cells and selects the neighbouring cell with the highest success rate.

Workflow:

1. When the Agent-3 is at the start node, it calculates the success rate for all of its neighbouring cells.
2. To calculate the success rate Agent-3 calls Agent-2 for every cell over 5 simulations.
3. If the success rate for all its neighbouring cells is the same then Agent-3 breaks the tie using Euclidean Distance.
4. If our Agent-3 doesn't have any choice between the neighbouring cells then it will use Agent-2 behaviour and move away from the nearest Ghost.
5. For storing the success rate we are using a Hash Map wherein the Key is the coordinate pair and value is the success rate.

Bottleneck:

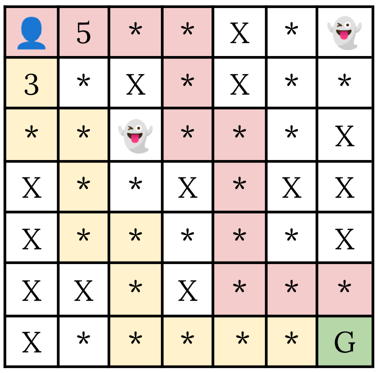
When the success rate as well as the euclidean distance for all the neighbouring cells of Agent-3 are the same then we face a wiggle effect. To solve the wiggle effect we have assigned penalties when we revisit a previously visited node. We decrement this penalty value from the success rate of the revisited cell and evaluate our next move on the basis of the new success rate.

Pseudo Code for 'startAgentThree' function:

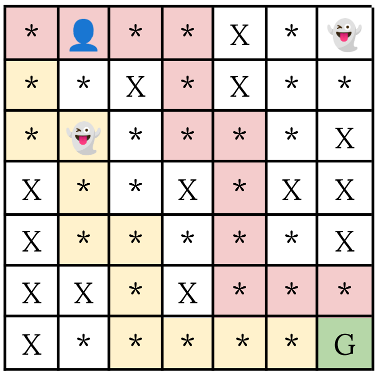
```
Function startAgentThree(environment,current,goal,visited map){
    Initialize current success to MIN_VALUE;
    set strongCell as current;
    moveGhosts;
    if (environment.ghosts.contains(current)) return false;
    if(current.equals(goal)) return true;
    //running 5 simulations for every neighbouring cell
    Map<Pair, Integer> map = agentTwo.simulation(environment, current,
5);
    if(map.size >0){
        for (Map.Entry<Pair, Integer> mapElement : map.entrySet()){
            set cell to the Hash map Key;
            set success to the Hash map value;
            if(node is revisited) success --;
            if(success > current success){
                set current success to success;
                set strongCell to cell;
            }
            else if(success == current success){
                calculate the EuclideanDistance;
            }
            else{
                moveAwayFromNearestGhost;
            }


        }
    }
    current = strongCell;
    if(visited.containsKey(current))  {
        visited.put(current, visited.get(current) + 1);
    } else {
        visited.put(current, 1);
    }
    print(map and the node to which the agent moved to);
    return startAgentThree(environment, current, goal, visited);
}
```

Representation of AGENT-3:



a) From the start node, the success rate of one of its neighbours is 5 and the other neighbour is 3.

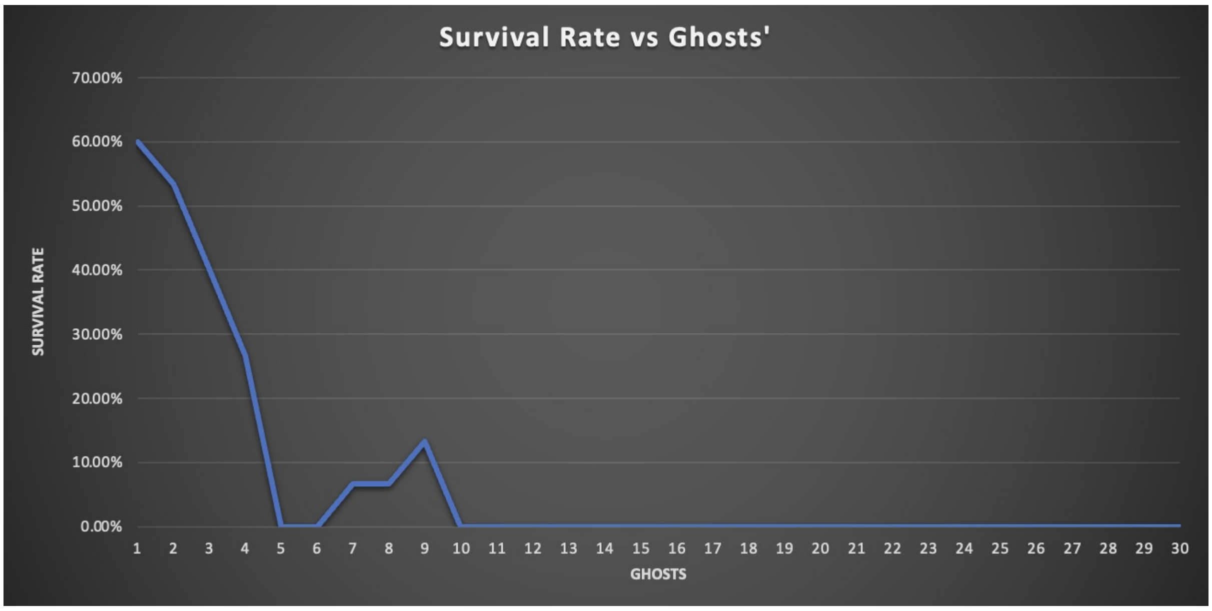


b) Agent-3 chooses the neighbour with higher success rate

Data Analysis of AGENT-3:

We are calculating our agent's survivability when there are 'n' numbers of ghosts where n is an integer between 1 and 30 for 15 simulations.

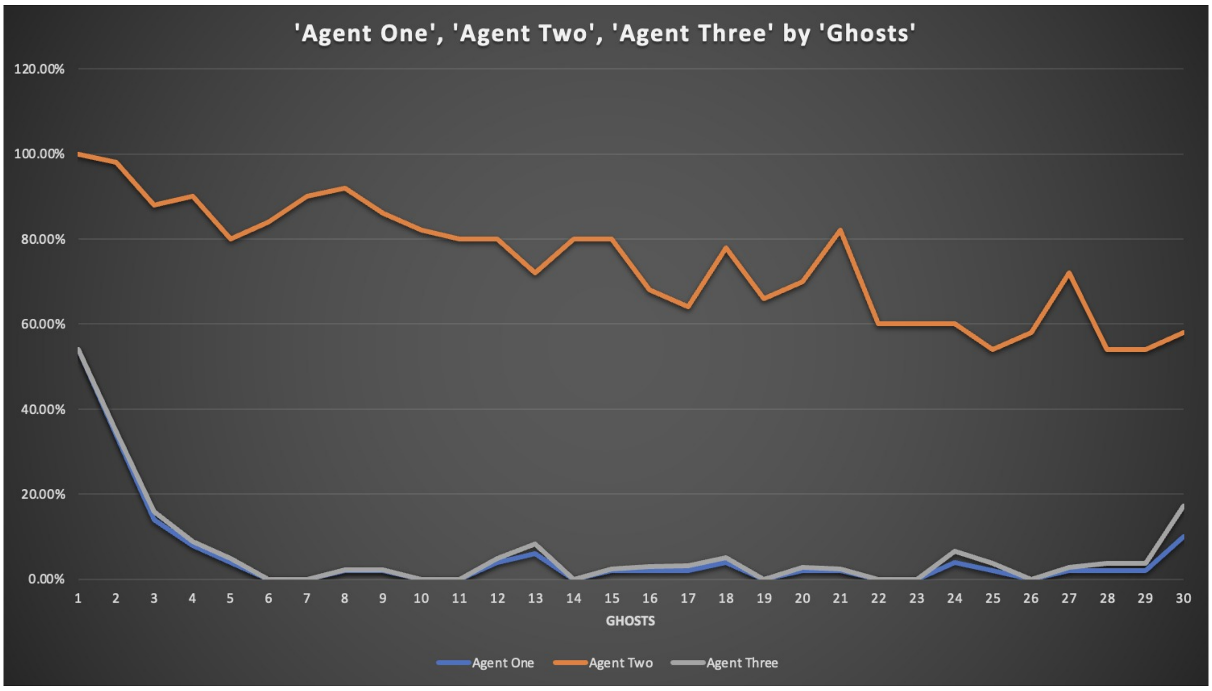
Bottleneck: We are facing performance issues with our Agent-3 because of the wiggle effect. Our Agent-3 keeps backtracking to the previous node which decreased its survival rate and increased the computational time exponentially.



Pseudo Code for the simulate function of AGENT-3:

```
Function simulate(environment,start){
    set goal node;
    Map<Pair, Integer> visited = new HashMap<>();
    boolean reachedGoal = startAgentThree(environment, start, goal,
visited);
    if(reachedGoal) return 1;
    print("Agent Died");
    return 0;
}
```

Comparison graph between AGENT-1, AGENT-2, AGENT-3:



Description for AGENT-4:

The aim here is to have a balance between intelligent decisions and efficiency. So we are making use of A* *algorithm with two heuristics* for Agent-4. One heuristic is calculating the distance from the start node to the current node and from current node to goal.The second heuristic being the penalty score assigned to the agent when the agent is moving towards the ghost.

Workflow:

1. Initially, we are using a priority queue to keep track of the unvisited cells and a Hashset to keep track of the visited cells.
2. We are also maintaining a Map to keep track of the cost functions. In the Map, the Key is the current cell pair and the value is a wrapper node that contains cost functions.
3. We begin by pushing the start cell pair into the priority queue and add the start pair to 'visited' Hashset.
4. We will then move the ghosts and check if the agent is alive and reached the goal.
5. If the agent didn't reach the goal then we explore the neighbouring cells at the top of the priority queue.
6. Every time we pop a cell pair from the priority queue,we select the cell with the minimum value of cost function. This is internally handled by our min heap priority queue.
7. If the agent reaches the goal then we reconstruct the path and return true.

Code for ‘startAgentFour’ function:

```
boolean startAgentFour(Environment environment, Pair start, Pair goal){

    // Directional Arrays
    int[] X = {0, 0, 1, -1};
    int[] Y = {1, -1, 0, 0};

    Node node = new Node(start, 0, (int)computeHeuristic(start, goal,
environment));

    // To keep track of unexplored nodes
    PriorityQueue<Node> priorityQueue = new PriorityQueue<>(new
CustomComparator());

    priorityQueue.add(node);

    // To keep track of explored nodes
    Set<Pair> visited = new HashSet<>();

    // To keep track of Nodes
    Map<Pair, Node> map = new HashMap<>();


    // To keep track of Path
    environment.parent.put(start, null);

    // To store distances
    map.put(start, node);


    while (!priorityQueue.isEmpty()){

        Node top = priorityQueue.poll();

        moveGhosts(environment);

        if(environment.ghosts.contains(top.cell)) return false;

        if(top.cell == goal) {
            reconstructPath(environment);
            return true;
        }
        // If already visited
        if(visited.contains(top.cell)) continue;

        // add node to the explored node
        visited.add(top.cell);

        // Visit neighbours or children
        for(int i = 0; i < 4; i++){
            int newX = X[i] + top.cell.first, newY = Y[i] +
top.cell.second;
            if(checkBounds(newX, newY, environment.grid.length)){
                Pair child = new Pair(newX, newY);
                if(environment.grid[newX][newY] == '*') {
                    // F(x) = G(x) + H(x) { G(x) -> Distance from
current to child, H(X) -> Distance from child to goal + penalty
                    int cost = top.g + 1 + (int)computeHeuristic(child,
goal, environment);
                    if(visited.contains(child)) {
                        Node childNode = map.get(child);
                        if(childNode.f > cost){
                            childNode.f = cost;
                            map.put(child, childNode);
                            priorityQueue.remove(childNode);
                            environment.parent.put(child, top.cell);
                        }
                    }
                    else {
                        Node childNode = new Node(child, top.g + 1,
cost);

                        priorityQueue.add(childNode);
                        map.put(child, childNode);
                        environment.parent.put(child, top.cell);
                    }
                }
            }
        }
    }
    return false;
}
```