Project Report - I

Ghosts in a Maze

Sri Krishna Kaashyap Madabhushi
Net ID: sm2599

Pranathi Vaddela
Net ID: pv250

October 15, 2022

GitHub: Link

Contents

1 Implementation

1.1 Assumptions

For this project, the following are our assumptions:

1 A particular cell in a grid can have more than one ghost.

2 A ghost can choose to remain inside a wall at a 50% probability.

3 A ghost can choose to go inside a wall or remain at the same cell at a probability of 50%.

4 At every timestep, an agent can choose to move or remain at the same cell.

1.2 Environment

The following are our assumptions to construct the environment for the project. Since an agent and ghost should be able to move, we have considered a 2-D grid to be our environment. Our environment generation code runs in the following way:

1 At every new iteration, a new grid of size *51 * 51* is generated.

2 The values of the grid is represented by numerical values, initially a value of 0 is assigned to each cell. This represents that all the cells are unblocked.

3 Once the grid is generated, we need to make the states of the cells either blocked or unblocked with a probability of *28%* for the blocked cell.

4 In order to get the probability to be 28%, we have constructed a new array of size 100, and have made 28 locations in the array blocked.

5 Now, we iterate through every cell and get a random location from the 1-D array. If the random location is blocked, then we block the cell of the maze, else the cell remains unblocked.

6 This follows the probability of 28% while maintaining randomness in our selection. At the same time, we ensured the start position and the end position is not blocked.

7 Now, we used BFS algorithm to check if there exists atleast one path from the starting location to the ending location. If the path exists, we return the grid. If not, we call the same function recursively until a valid grid is returned.

8 **Note:** *This BFS step also returns a precomputed 2-D matrix that is used in all the implementations of our agents. More details about it is explained in the later part of the report.*

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

We felt Breadth-First Search (BFS) to be more useful for checking the existence of the paths because, there might be cases in DFS where the number of iterations it takes to reach the destination is long. In BFS, the moment we reach the destination, we are certain that we have taken the shortest possible path. And also, since the grid is of size 51*51, the space it takes for BFS to run doesn't overflow. Secondly, we have used this same BFS step, iterating in a level order fashion in our precomputation step, explained in the later part of the report.

1.3 The Agent

We have not used a special datastructure to store the agents position. We have defined 4 individual classes for each agent and, the agent's current position is being tracked by the state variable of the class. At every time step, either the agent chooses to move in one of the 4 directions or decides to stay at the same cell.

1.4 The Ghost

In this project, we have used a hashmap to store the current ghost locations. Since a particular cell can have more than one ghost, the key of the hashmap would be the ghost position, the tuple (row, column) and the value of this key would be the number of ghosts in that particular cell.

At the same time, we are also representing the ghost in a grid. Everytime a ghost enters a cell, the value at the cell in the grid is incremented by 2. Therefore, the possible states of each cell are as follows:

1 Unblocked - Represented by a 0

2 Blocked - Represented by a 1

3 Ghost(s) at unblocked cell - Represented by a list of even numbers such that for n ghosts, the value at the cell would be *2*n*.

4 Ghost(s) at blocked cell - Represented by a list of odd numbers such that for n ghosts, the value at the cell would be *2*n + 1*.

Note:

At every time step, we use two conditions to make sure that the agent is alive.

1 Whenever we move the agent, we check if there exists a ghost in that cell. We check this using our ghostMap in a constant time.

2 Alternatively, when we move the ghost, we again check if the new ghosts position is the same as the agents position.

2 Precomputation

2.1 Why Precomputation?

The main idea behind the precomputation steps is to reduce the running time of all our agents. We have used this precomputed values throughout our implementations thereby reducing the running time of our algorithms drastically.

We tried to avoid computations that remain constant throughout the iterations and we have identified them to be:

1 The shortest distance in terms of number of steps from each unblocked cell to the destination.

2 The shortest path from each unblocked cell to the destination.

Through our various iterations, *number of steps* has been proven to be a better heuristic than the *manhattan distance* for all our distance heuristics. And by precomputing this beforehand and using this, we get the distance heuristic in a constant time reducing our overall runtime.

Secondly, having a shortest path stored from every cell has helped us avoid recomputation of a path. With this value, we always had the information of the next cell to which the agent should move inorder to reach the destination in the shortest path. This has been another factor that has helped us reduce the runtime of our agents exponentially. For example, in the case of agent 2, when there is a ghost in path and we take a detour, we didnt have to compute the path again since we had it stored.

2.2 Precomputation Steps

1 Initially, when the maze is generated, instead of running the BFS from the source to destination, we ran the BFS algorithm from **destination to source**.

2 If we are able to reach the source cell, then the maze is valid. However, while travelling from the destination, we traverse in a level-order fashion.

3 This means that at every iteration, we are moving 1 unit distance away from the previous cell. Therefore, we take a new array named *path*, that stores the distance to the destination for all the cells, while storing the parent cell.

4 By the end of this BFS, we have computed shortest path from all the unblocked cells at the same time, storing the distance from the destination.

2.3 Example

Let's understand the precomputation steps by using an example. Imagine we have the following 10 x 10 maze with obstacles as below:

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0
\end{bmatrix}$$

We can clearly see that there exists a path from top left to bottom right corner for the above matrix. Our precomputed path matrix for the above maze would be as follows:

$$\begin{bmatrix}
(1,0,18) & (1,1,17) & (1,2,16) & (1,3,17) & -1 & -1 & (1,6,14) & (0,6,15) & -1 & -1 \\
(1,1,17) & (2,1,16) & (2,2,15) & (1,2,16) & -1 & (1,6,14) & (2,6,13) & -1 & -1 & -1 \\
-1 & (3,1,15) & (3,2,14) & -1 & -1 & -1 & (3,6,12) & (3,7,13) & -1 & -1 \\
(3,1,15) & (3,2,14) & (4,2,13) & -1 & (3,5,11) & (4,5,10) & (3,5,11) & (3,6,12) & (3,7,13) & (3,8,14) \\
-1 & -1 & (5,2,12) & (5,3,11) & -1 & (5,5,9) & -1 & (3,7,13) & (3,8,14) & (3,9,15) \\
-1 & -1 & (6,2,11) & (6,3,10) & -1 & (6,5,8) & -1 & -1 & -1 & (4,9,16) \\
-1 & (6,2,11) & (6,3,10) & (6,4,9) & (6,5,8) & (7,5,7) & (6,7,6) & (6,8,5) & (7,8,4) & -1 \\
(7,1,13) & (6,1,12) & (6,2,11) & (6,3,10) & -1 & (8,5,6) & -1 & -1 & (8,8,3) & (8,9,2) \\
-1 & (7,1,13) & -1 & (7,3,11) & -1 & (8,6,5) & (9,6,4) & (9,7,3) & (9,8,2) & (9,9,1) \\
-1 & -1 & -1 & -1 & -1 & -1 & (9,7,3) & (9,8,2) & (9,9,1) & 0
\end{bmatrix}$$

In the above matrix, every unblocked cell consists of a tuple in which the first two values signify the next (row, column) that the agent needs to go to reach destination in the shortest path along with the third value being the distance in terms of steps to reach destination in the shortest path.

Note: Throughout our implementation, this precomputed matrix is named **path**.

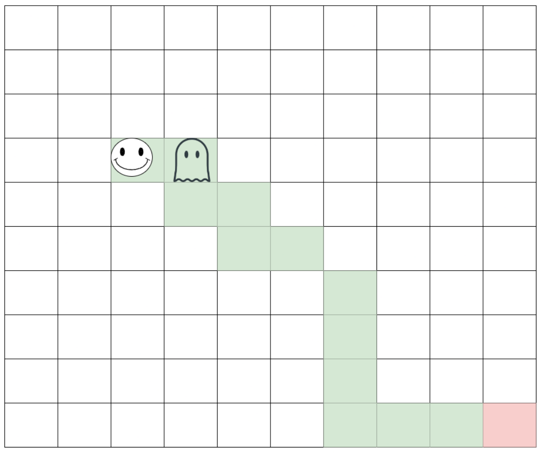
3 Agent 1

Since we have already precomputed the shortest path from all the cells, we start at the source *(0, 0)* and keep moving towards the destination until it either reaches the goal or gets killed by the ghost. Though Agent 1 is the fastest in terms of number of iterations and runtime, it doesn't take the changing environment into consideration and as the number of ghosts increase, the chances of getting killed is increased exponentially.

4 Agent 2

4.1 Agent 1 Drawbacks

The primary drawback of *Agent 1* is that it doesn't consider the ghosts in its path and as the number of ghosts increase, it ends up entering the cell with a ghost in it and dying in the process. This is depicted in the following image. At timestamp t1, lets say the agent is occupying a cell and trying to move in its shortest path, (depicted by green cells), even if there is a ghost in the next cell, it enters the cell and ends up dying. Therefore, our strategy for agent 2 is to address this draw back.


(a) Agent 1 at timestamp i

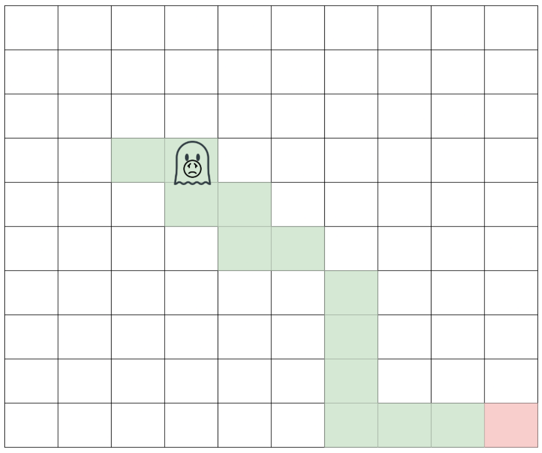

(b) Agent 1 at timestamp i+1

Figure 1: Agent 1 Drawback

4.2 Agent 2 Strategy

At every step, our agent 2 checks if there is any ghost obstructing its path. If there is, then it takes a detour to one of its adjacent cells, including taking a step back such that the new path either doesnt have a ghost in it or the ghost is farthest from the current cell. This is because, as we increase the number of ghosts, the likelihood of finding a path that has no ghosts in it reduces, thereby essentially taking the path in which the closest ghost is as far as possible.

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?

We dont always need to replan a new path at every timestep. It is efficient to move in the shortest path as long as there is no ghost in our planned path. We only need to replan if the current planned path is obstructed by a ghost.

4.3 Agent 2 Working Steps

1 Since we have a precomputed shortest path, we have taken the path and stored it in hashmap.

2 This step is done because whenever a ghost takes a step, we check if the new location is present in the planned path or not.

3 The presence of the ghost in the path is checked by the *isGhostPresentInPath* function that returns a True and the distance of the closest ghost in the path.

4 Our aim for the next step is to essentially find a path that has no ghosts in it or to maximize this distance as much as possible by considering all the 4 directions, including a step back.

5 The above steps are repeated until we either reach the destination or eaten by a ghost in the process.

5 Agent 3

5.1 Agent 2 Drawbacks

The primary drawback of *Agent 2* is that it only considers the current state of the ghosts but doesn't take into the account the future steps the ghost might take. For instance, though a particular ghost is in the path, the likelihood of the ghost staying in the same cell by the time the agent reaches the cell is low. Therefore, just because there is a ghost in the current path, we doesn't always need to replan but instead, pick a choice that has a higher success rate.

5.2 Agent 3 Strategy

By handling the above drawback that agent 2 has, our main idea is to predict the ghost movement and take a step using that information. Therefore, at every step, we run our *Agent 2* for 10 iterations in each direction, and based on the success rate we get, we choose the step that has the maximum success rate.

5.3 Agent 3 Challenges and Wiggling Condition

While implementing our *Agent 3*, we have initially taken a recursive step to call our Agent 2 multiple times because the recursion stack stores the previous state of the Agent 3 and resumes the execution for the next step with the old states after finishing Agent 2's execution. However, the biggest challenge we faced while implementing our Agent 3 was the wiggling condition. There have been some instances where our agent started wiggling between two cells due to the ghost position and after a while, it was exhausting all the recursion stacks, thereby throwing maximum recursion depth error. One of the choices we had was to increase the recursion depth to *5000* but that didn't help.

5.4 Handling Wiggling Condition

Therefore, to handle the above challenge of wiggling, we had to limit the number of times the agent visits a cell. To do this, we have used a visited set and a special heuristic that would initially grow slowly but after a point, would grow exponentially. And we have assigned this value as our penalty for every step.

Therefore, our heuristic to select a step is modified as **failure rate + penalty + distance to goal**. Since we have already computed the distance to the goal previously using our precomputation steps, we would use the same computed value, and would be passing the number of times a cell is visited into our *getPenalty()* function that would return the penalty of taking the step. Finally, we have stored the heuristic value in a **Min-Heap** and chose the step with minimum heuristic value.

5.5 Why Wiggling is nice?

By choosing the penalty heuristic that would not completely avoid the wiggling, we are considering the wiggling cases upto a limit. This is because the agent wiggles because it cant find a right direction to move. While the agent is wiggling, the ghosts also move and might create a new path that would other be blocked if we avoid wiggling completely.

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 Agent 3 decides that there is no successful path in its projected future, (i.e.), by the success rate is very low or 0 for all the adjacent cells, its ideal for it to remain at the same cell and wait for the ghosts to move. Cases like these occur when all the 4 directions are blocked by the ghosts and choosing any direction would lead to death. However, it doesn't guarantee that the success is impossible. What it actually means is that the probability of finding a success in these paths is very low because every ghost's move is probabilistic and it is unlikely that the ghosts take one of the previous steps when we decide to move.

5.6 Agent 3 Working Steps

1 Initially, it starts from *(0, 0)* and runs Agent 2 in all possible directions 10 times.

2 In this case, on assuming that *(1, 0)* and *(0, 1)* to be a valid unblocked cells and doesnt have a ghost, we run agent 2 10 times at each of these cells and compute our heuristic value, **(failure rate + penalty + distance to destination)**.

3 We add these to a **Min Heap** and pick the step that has the minimum value.

4 **Note:** Since we are running the Agent 2 10 times in every possible cell, though our Agent 2's runtime is very low, our Agent 3 took a considerable amount of time for complete iteration.

6 Drawbacks of Agent 2 and Agent 3

The primary drawback of Agent 2 and Agent 3 is that it considers and replans if there is a ghost in its planned path. However, it doesn't consider the ghosts in the surrounding viscinity. As the number of ghosts increase, the chances of a ghost not lying in the path but just around the agent increases, thereby killing the agent. This is depicted in the example figure 2.

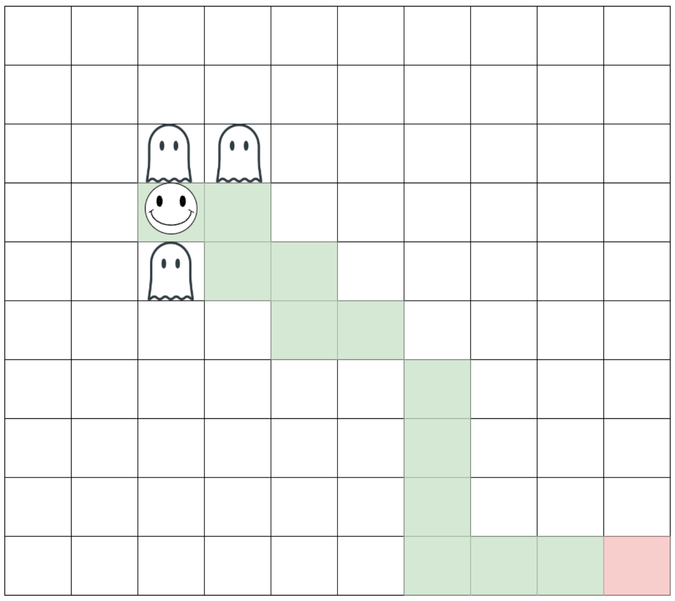


Figure 2: Drawbacks of agent 2 and 3

For the cases like these, the agent thinks that the path is safe but it is highly likely that one of the ghosts kill the agent in the next step. As the number of ghosts increase, it is more likely that the agent is surrounded by the ghosts and end up making a wrong decision in choosing the next step.

The second drawback of these agents is that it also considers the ghosts that are in the path but far from the current cell. The ghosts that are far from the current cell have a minimum or no effect on the agent because by the time we reach the ghosts position, it is highly likely that the ghost moves away from that cell thereby clearing the path for us.

7 Agent 4

7.1 Improving the Drawbacks

The main purpose of this agent is to overcome the two drawbacks that the previous agents have. Therefore, the main idea of this agent is that instead of re-planning when there is an agent in the path, we create a *circular vicinity* around the agent and takes a path that moves away from the ghost in the vicinity thereby avoiding contact with the ghost. This approach also doesn't consider the ghosts that have minimal or no effect on the agent thereby boosting both the performance as well as the survivability of the agent.

7.2 Agent 4 Working Steps

1 Initially, the agent tries to move in the shortest path while creating a circular vicinity boundary around itself.

2 If there is any ghost entering the boundary, it computes the cell that has the highest distance from the ghost and takes this step.

3 Since we have precomputed shortest paths from every cell in the maze, the agent continues to move in the new shortest path.

4 With the increase in the number of ghosts, the likelihood of having multiple ghosts in the vicinity increases and the agent picks the path that moves away from all these ghosts.

5 Therefore, to measure the next step it should take, we have designed a custom heuristic that takes the number of ghosts in the direction, distance between the agent and ghost, and the distance to the destination into consideration.

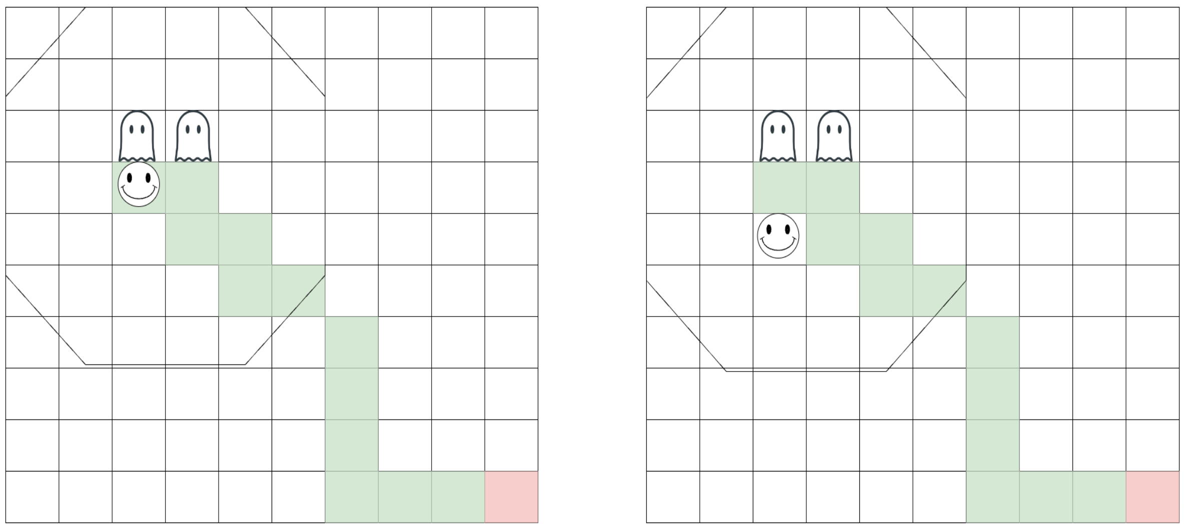

(a) Agent 4 at timestamp i (b) Agent 4 at timestamp i+1

Figure 3: Agent 4 Smart Decision

8 Analysis

8.1 Computational Bottlenecks

The main computational bottleneck we faced was for Agent 3 and Agent 6(Described in the later part of the report) in terms of time. It was because we have not completely avoided the wiggling condition. And also, since they were the boot-strapped versions of Agent 2 and 4. To reduce the effect of wiggling, we have introduced a new parameter *penalty* into the program.

Secondly, with the increase in the number of ghosts, the size of our *ghostMap* that stores all the locations of the ghosts increased thereby increasing the space required to store the ghosts. However, it was an expected behaviour and the hashmap helped us reduce the overall execution time of the program by giving us the constant access times.

Finally, our first draft computed the path from the current cell and the distance to the goal for each iteration(or whenever we had to). We have reduced this time exponentially using our precomputation step that has been used in developing all our agents. Since we have embedded the precomputation within the validation of maze, we got extra accessible information without any extra computation.

8.2 Graphs Comparing the Performance

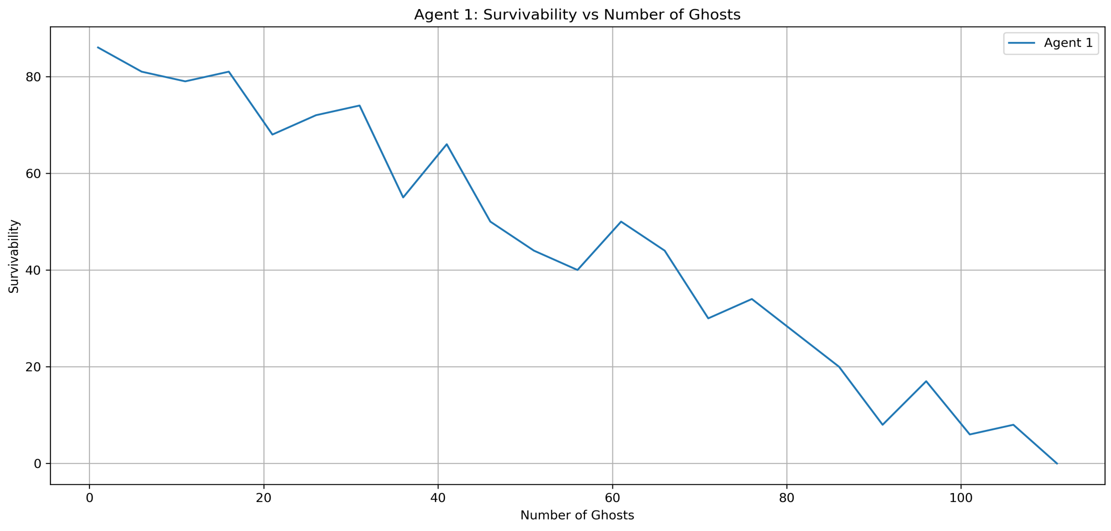


Figure 4: Agent 1 Success Rate vs Number of Ghosts

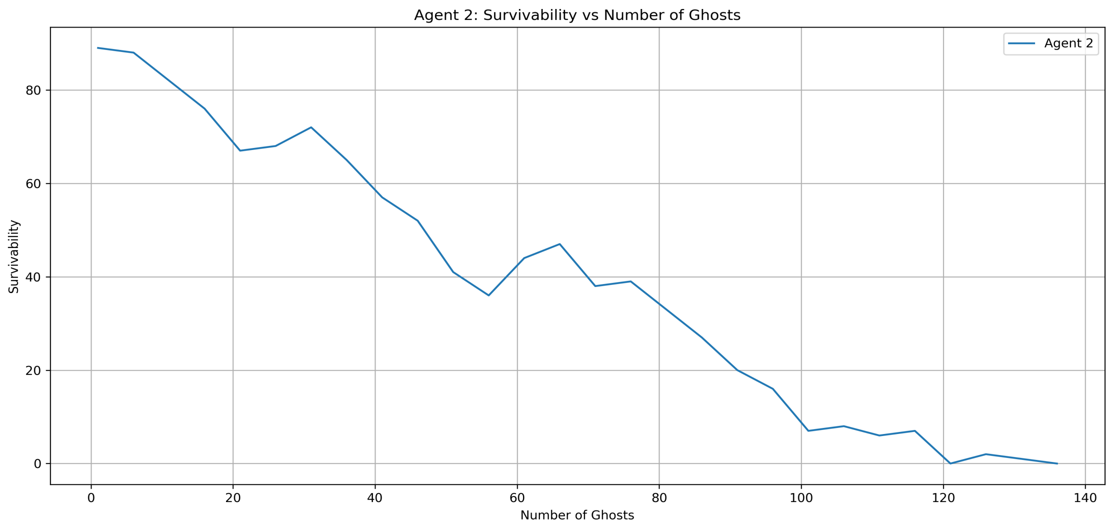


Figure 5: Agent 2 Success Rate vs Number of Ghosts

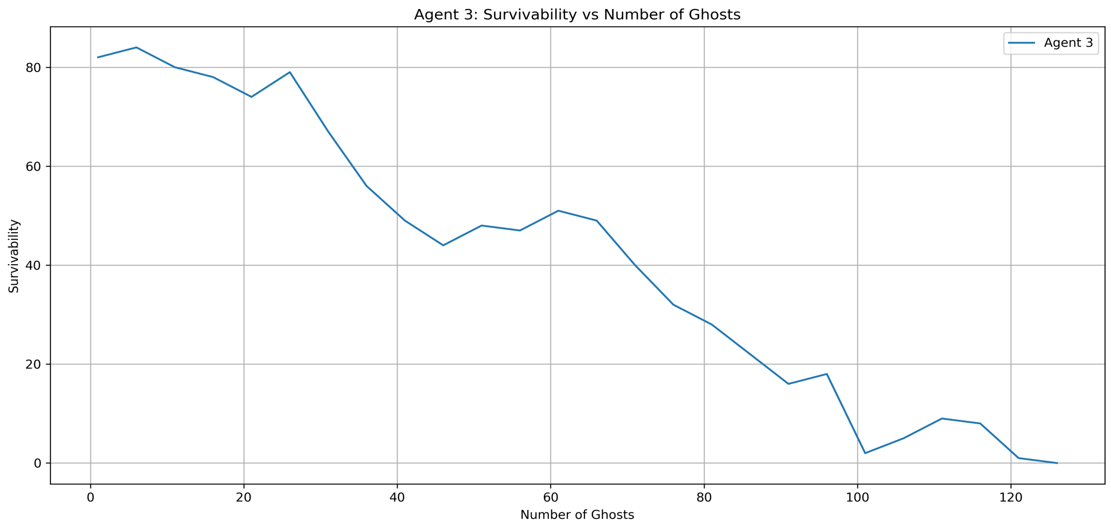


Figure 6: Agent 3 Success Rate vs Number of Ghosts

Note

To find more accurate results, we have run the each step for 100 iterations. Since our Agent 3 took considerably more time as compared to other agents, we have increased the step size for our agents by 5 for a better comparison.

The survivability of each agent goes to 0 at the following number of iterations:

- **Agent 1:** 111 ghosts.
- **Agent 2:** 136 ghosts.
- **Agent 1:** 126 ghosts.

8.3 Agent 1, 2 and 3 Comparison

Firstly, we have observed that there was no single agent that always performed better than the other in all the cases. However, our Agent 2 outperformed Agent 1 in most of the cases. The drawback of Agent 1 was that it wasn't considering the ghosts position and as the number of ghosts increased, the likelihood of the Agent 1 dying increased drastically.

Though Agent 3 performed more computation and had more information with respect to the maze as well as the ghosts future movements, our Agent 3 always couldn't outperform Agent 2 because Agent 3 didn't have a definite path in mind and was using Agent 2's results to decide the next path. Hence, although it concludes to choose a next step, the agent is unclear about the complete path and the probability of it dying in the chosen path is still quite high because of the random probabilistic movement of ghosts.

Finally, though every agent had its own drawback, there wasn't a number for which one agent was *always* performing better than the other, but for some random iteration, the probability of Agent 2 performing better than 1 and 3 was higher.

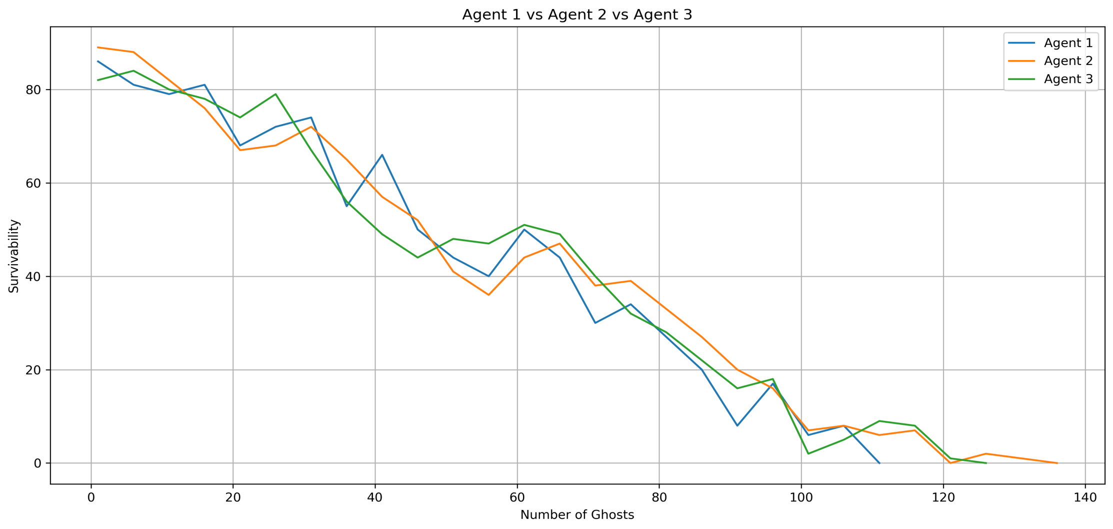


Figure 7: Agent 1 vs 2 vs 3 Performance Comparison

Theoretically, Agent 3 has all the information that Agent 2 has, and more. So why does using this information as Agent 3 does fail to be helpful?

Theoretically, Agent 3 has all the information that it requires to make a more informed decisions. However, the primary reason that it couldn't beat Agent 2 was because Agent 3 over-exploited Agent 2's behaviour to find the optimum path. The information that the Agent 3 has was far more than required to make an informed decision. And, as the number of ghosts rise, the behaviour of the ghosts become more random. That is, though we took the success rate based on 10 iterations in each direction, the likelihood of the ghosts following the similar moving pattern on the 11th iteration is very low.

8.4 Agent 4 Comparison

Our Agent 4 was able to outperform all the previous agents(1, 2 and 3) because the primary limitation of agent 2 and 3 is that it considered ghosts only in the path that it is travelling and ignored the other ghosts in the vicinity. Therefore, the probability of getting killed by a ghost around the agent increased drastically with the increase in the number of ghosts.

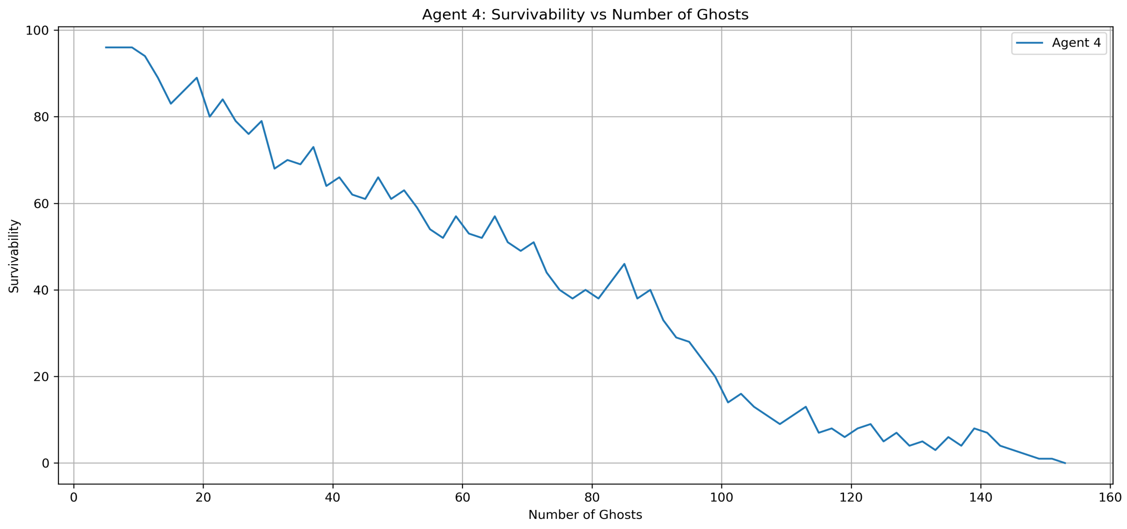


Figure 8: Agent 4 Success Rate vs Number of Ghosts

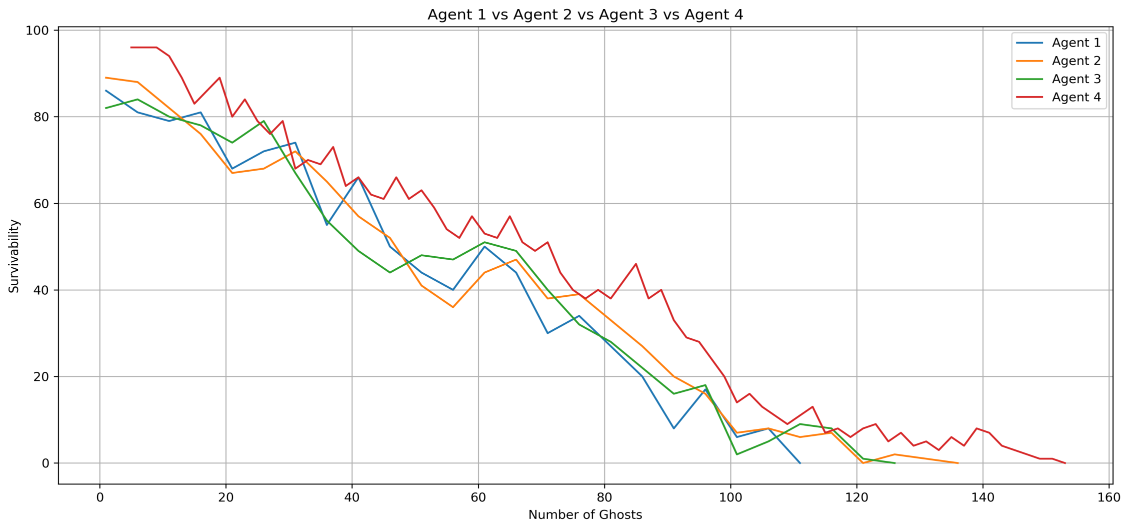


Figure 9: Agent 1 vs Agent 2 vs Agent 3 vs Agent 4

9 Agent 5

We have used the same idea of Agent 4's vicinity while ignoring the ghosts inside the wall. That is, if there is a ghost in a wall in the vicinity, we ignore the ghost and assume that there is no ghost. Our Agent 5's performance was very close to Agent 4's but slightly lower. This was an expected behaviour because the case of ghost coming through the wall and killing the agent wasn't being considered.

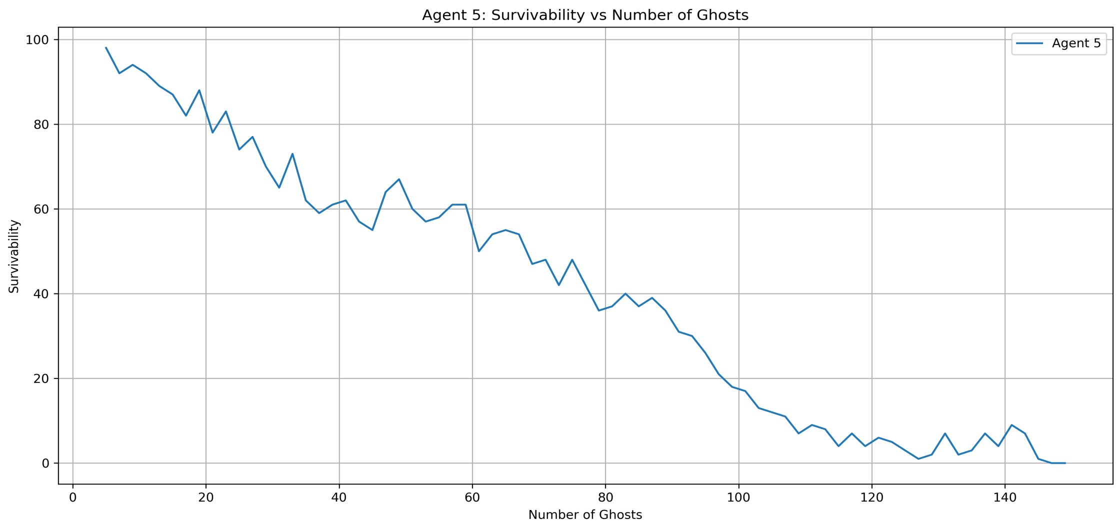


Figure 10: Agent 5 Success Rate vs Number of Ghosts

Since our representation of grid is both simplistic and scalable, (blocked cells are always odd and unblocked cells are always even); we didnt have to make many changes to our Agent 4. For every ghost in our vicinity, we checked whether the value at the cell is even or odd. If the value is odd, then we didn't consider that ghost to be in the vicinity.

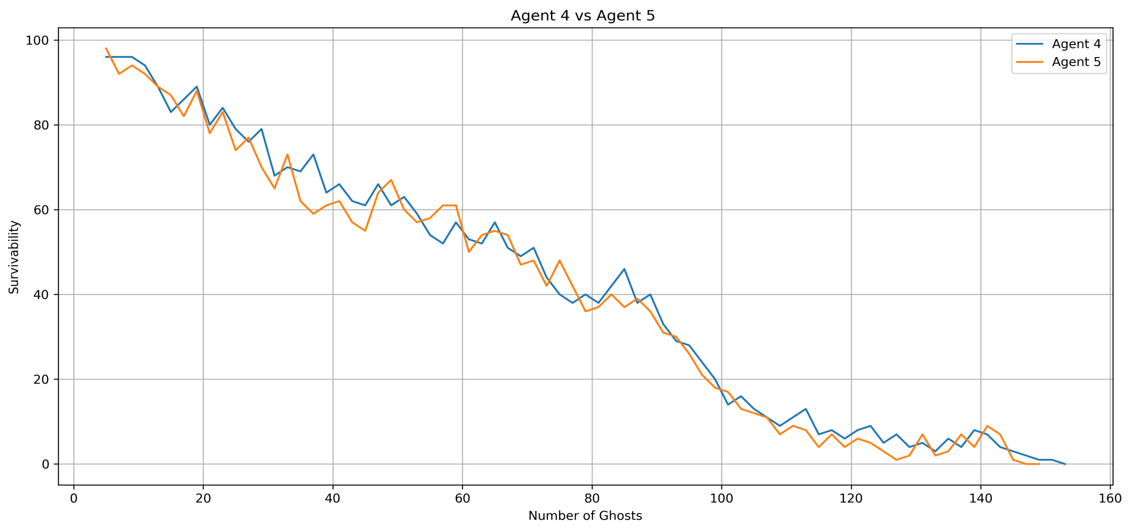


Figure 11: Agent 4 vs Agent 5

10 Agent 6

Since our agent 4 was outperforming all the other agents, we decided to implement an Agent 6 using Agent 4. The idea behind developing the Agent 6 was to analyse the behaviour and performance of Agent 4 if more information is given to it. Therefore, similar to Agent 3; at every step, we run our Agent 4 for 10 iterations in each direction, and based on the success rate along with a custom heuristic, we choose the step that has the minimum value of the heuristic.

We have also considered our key takeaways from Agent 3 of having a penalty and have used the same heuristic as Agent 3. Our Agent 6 outperformed the agents (1, 2, 3 and 5). However, it couldn't outperform Agent 4 in terms of survivability and also, the execution time for Agent 6 is considerably high(Similar to Agent 3) compared to other agents. Since the running time was high, in the interest of time, we have taken a bigger step size for Agent 6 as compared to Agents 4 and 5.

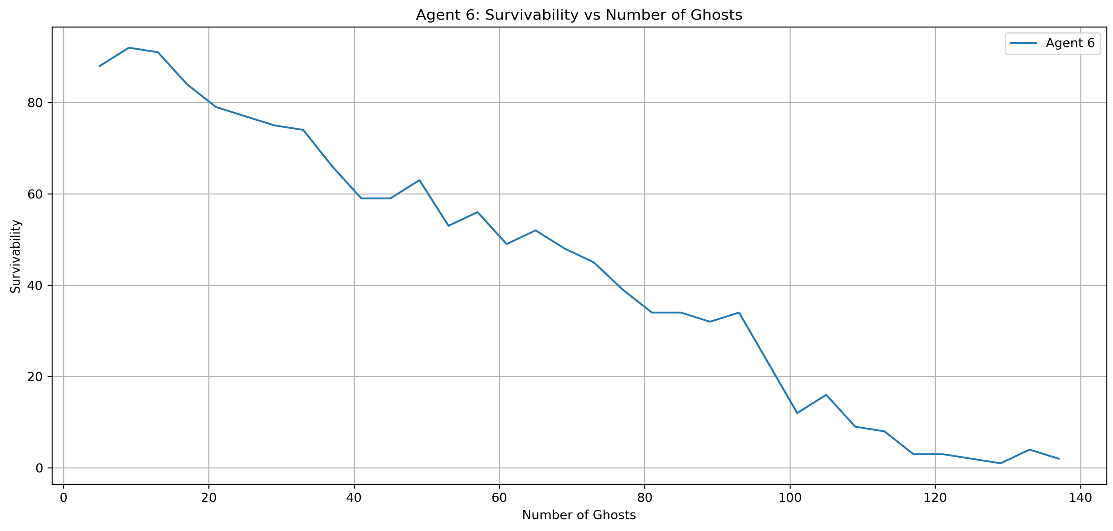


Figure 12: Agent 6 Success Rate vs Number of Ghosts

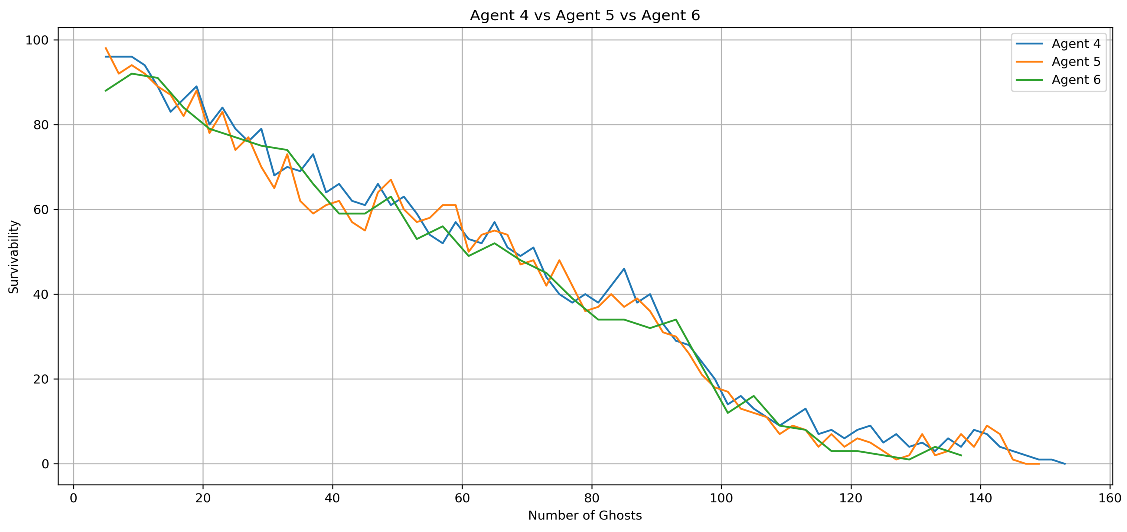


Figure 13: Agent 4 vs Agent 5 vs Agent 6

11 Conclusion and Takeaways

Among all of our implementations, our Agent 4 was able to perform better than all other agents including Agent 6. The key learnings and take aways from this project has been:

1 Understanding the search algorithms and implementing them.

2 Finding the right balance between exploration and exploitation. Realizing that having the maximum information doesn't always leads to the best decision.

3 It is always important to draw a balanced line between exploration and exploitation. Both, over exploration and over exploitation would affect our decision making.

4 We believe that the main problem with both Agent 3 and 6 were that they were designed as *over-exploitation* models and ended up taking bad decisions impacting the survivability of the agent.

5 We also believe that by understanding the limitations Agent 2 has, we were able to draw a balance between exploration and exploitation, that is, exploring and exploiting the cells in the vicinity to understand the neighbourhood better, eventually taking better decisions.

6 We have also concluded that Agent 4 **travels furthest** in the matrix before dying as compared to any other agents by looking at the cell in which the agent dies.

7 In terms of computations, we believe all our agents(1, 2, 4 and 5) have been implemented to be time efficient, (giving out results in the order of milli-seconds) and scalable enough to be able to expand to the mazes of much bigger sizes.

8 Our precomputation step has helped us to achieve this time complexity and we have derived the inspiration for it from the technique, **Dynamic Programming**.

9 We have used one of the noted ideas known as **bottom-up dynamic programming** where instead of starting a computation from the source, we start it from the destination that gives us more information about the environment.