CS 520 - Project 1: Ghost in the Maze

Submitted by Aditya Manelkar (am3369) and Anmol Arora (aa2640)

Design and Algorithms

Maze Generation

We randomly generated mazes using the constraints given in the project description. To check if the maze was solvable or not, we used a Bi-directional Breadth-First Search to check for possible paths. The space complexity is a good $O(b^{d/2})$ as compared to a simple Breadth-First Search's $O(b^d)$.

Q. What algorithm is most useful for checking for the existence of paths in a randomly generated maze? Why?

In order to identify the solvable mazes, those having paths leading from the start node to the goal node, we applied a Bidirectional Breadth-First Search. The reduction in space complexity that a Bi-Directional Breadth-First Search brings with it made it a *good* choice for us. Although it could be argued that using A* to locate a viable path through a maze would be a *better* idea, we wanted to try something different as we were planning on using A* extensively for each of the agents in our project. After hearing about the Kevin Bacon Problem in class, reading about it, and wanting to *avoid implementing the same algorithms* throughout our project, we chose to employ a Bi-Directional Breadth-First Search to check for the existence of paths in a randomly generated maze.

Ghosts

Ghosts were generated in the maze based on the rules provided in the writeup. For checking if a ghost was spawned on a path reachable from the start node, the Bi-Directional Breadth-First Search algorithm used to check for solvable mazes was re-used.

Conditions were applied to trigger ghost movements at every timestep, and to consider the run to end if a ghost were to enter the agent's cell or vice versa.

Agent 1

Agent 1 as outlined in the writeup was built to be a relatively naive agent.It would initially plan a path to course the maze using an A* Search Algorithm we implemented, and then later commit to moving as per the planned path (ignoring the ghosts later).

A* was the ideal algorithm to assess whether our agent could survive since it ensures that a path will be found if one exists. The A* algorithm extends all the nodes required for locating the

ideal path and avoids all extraneous nodes that are blocked - thus reducing the number of node searches.

What was our Heuristic?
In addition to common heuristics like Manhattan distance and Euclidean distance, Dr. Cowan had also mentioned that one could calculate heuristics by working through a relaxed version of the problem. The answer to the problem's relaxed version could then be applied to the real problem.

In our case, we relaxed the problem by eliminating all of the ghosts from it, then we located the best path through the maze from the present node to the destination node and set the heuristic equal to the number of steps it would take to get there. So after loading a maze, we would calculate these values and store them in a map that would be available to the agent throughout the run.

Agent 2

Agent 2 was modified in a way such that it would replan a better path during its movement as well to avoid running into ghosts (including the ghost avoiding logic if no good paths can be found).

Q. Does Agent 2 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?

Agent 2 does not have to replan every time. It will only replan when ghosts are discovered on the current planned path, and will continue on the same path if there are no ghosts on it.

Agent 3

For the design of our Agent 3, we went with the Monte Carlo approach. What we ended up doing is in a way where at every node we would check for utility values of its children and select the child with the best utility value.

Where did we get our utility values from?

Whenever we processed each neighbor/child node for our current node, we then ran 100 simulations of agent 2 through that node. For each simulation, if we managed to reach the goal node, we considered the outcome a "+1" and if the agent got caught by a ghost, the outcome was considered to be a "0". After running 100 simulations, we calculated the mean and set it as the utility for the child node. This process was repeated for all children from the current node.

Why perform 100 simulations?

After several runs of Agent 3, we realized that for a performance as good as or better than Agent 2, we had to run as many as 100 simulations. We initially started at 30 simulations per child node and then ramped up to 100.

Why run the simulations upto the end states?

We realized that just running simulations upto the next 10 moves was giving us many abnormalities. So we considered it best to always run up to an end state.

What issues did we face?

The biggest issue was the computational bottleneck we faced. We ended up doing most of the computation on our personal devices (which in retrospect was a bad idea).

How did we try to deal with the issue?
We tried to introduce multiprocessing to run simulations in parallel. But even in this, we ran into an interesting problem where the same memory was being referenced by all parallel instances for our agent and ghost objects (even when we would deepcopy them in their respective instances). Since we could not solve this problem in time, we decided to manually trigger multiple runs of different data points to at least reduce the time required by a factor of 5. Another arbitrary limitation we set up was the number of timesteps for which the simulations would run. This was done to prevent issues that would endlessly drag on simulations, like cycling between the same nodes or constantly running away from ghosts due to the absence of good paths.

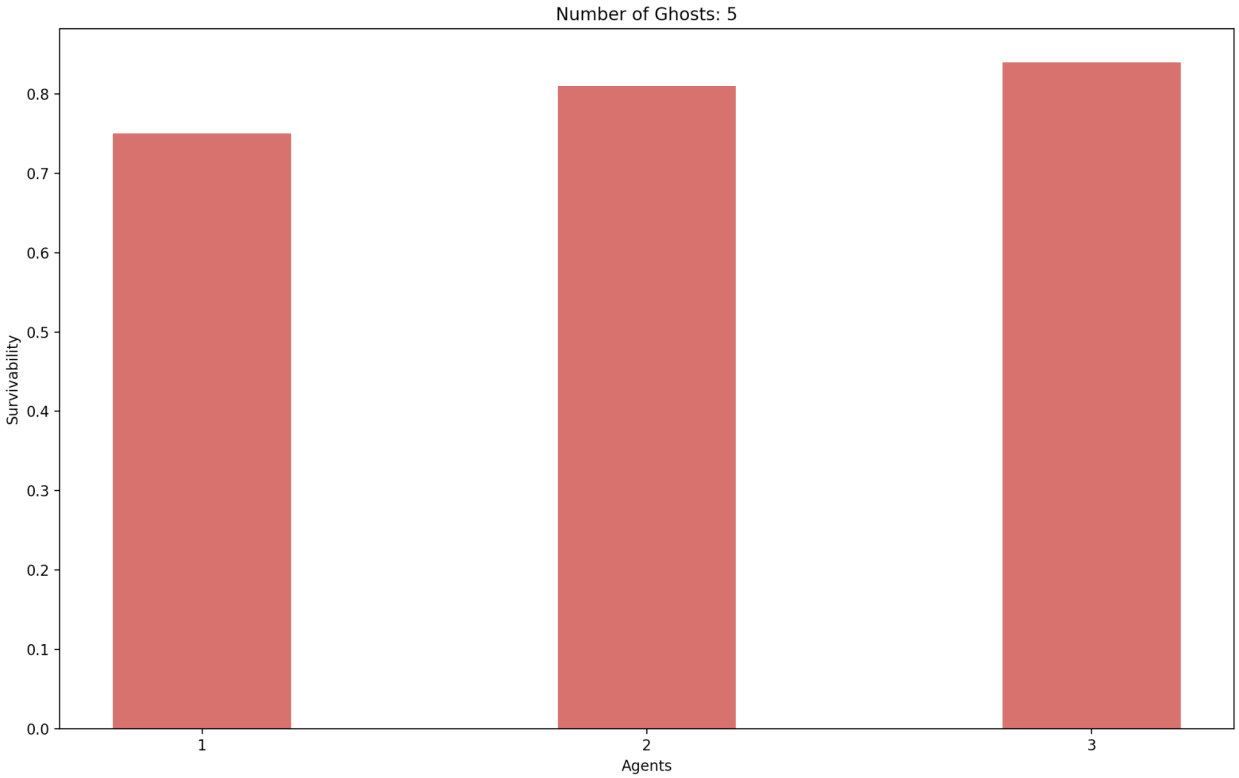
Q. If Agent 3 decides there is no successful path in its projected future, what should it do with that information? Does it guarantee that success is impossible?

As was hinted in the writeup, when our Agent 3 is unable to identify a path to the goal node in the current timestep, it immediately tries to move away from the closest ghost. While this indicates that there is no path to the goal node at that specific timestep, it does not imply that the maze cannot be navigated. What could happen is that by either staying in the same spot or moving away from the closest ghosts now could result in a non-zero utility being discovered in the following timestep (or even after 100 timesteps!) because the ghosts could have moved differently. What the simulations provide us is a basis for how the run could pan out starting with a specific board configuration for the current timestep.

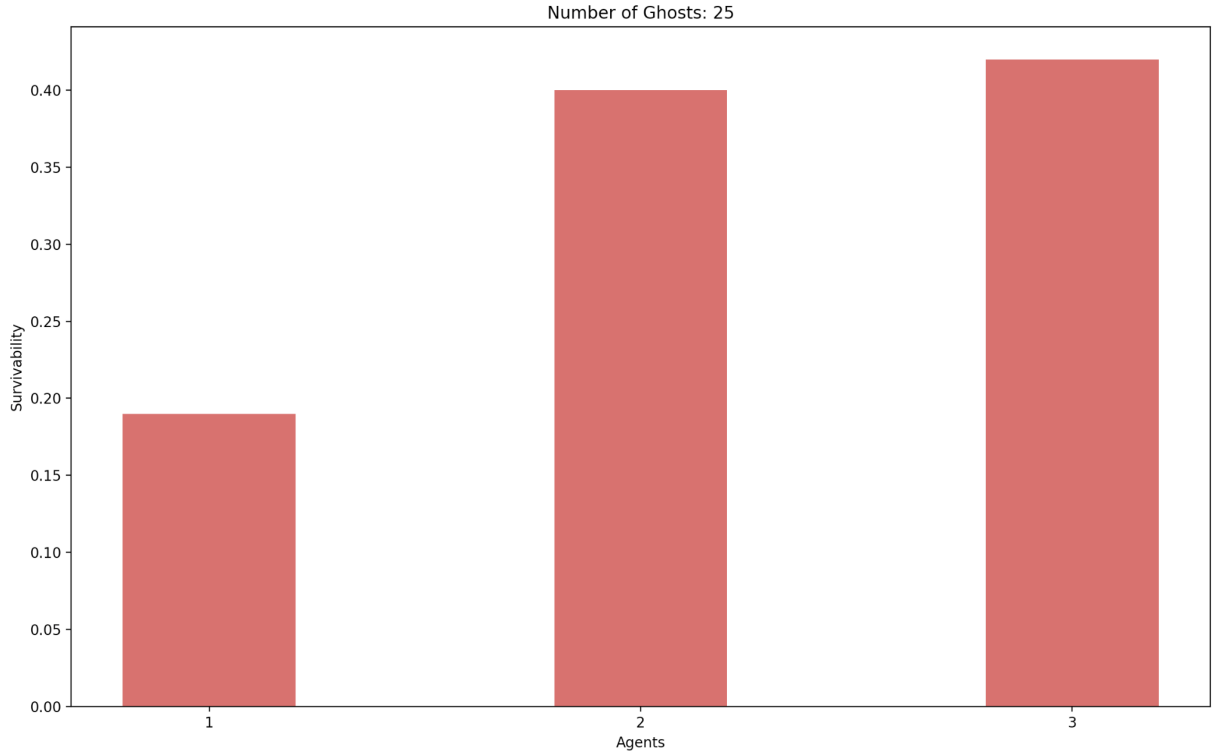
Q. How did Agents 1, 2, and 3, compare? Was one always better than others, at every number of ghosts?

The agents showed some general trends that indicated in general that the models were getting better, but there was no clear winner for all the data points selected.

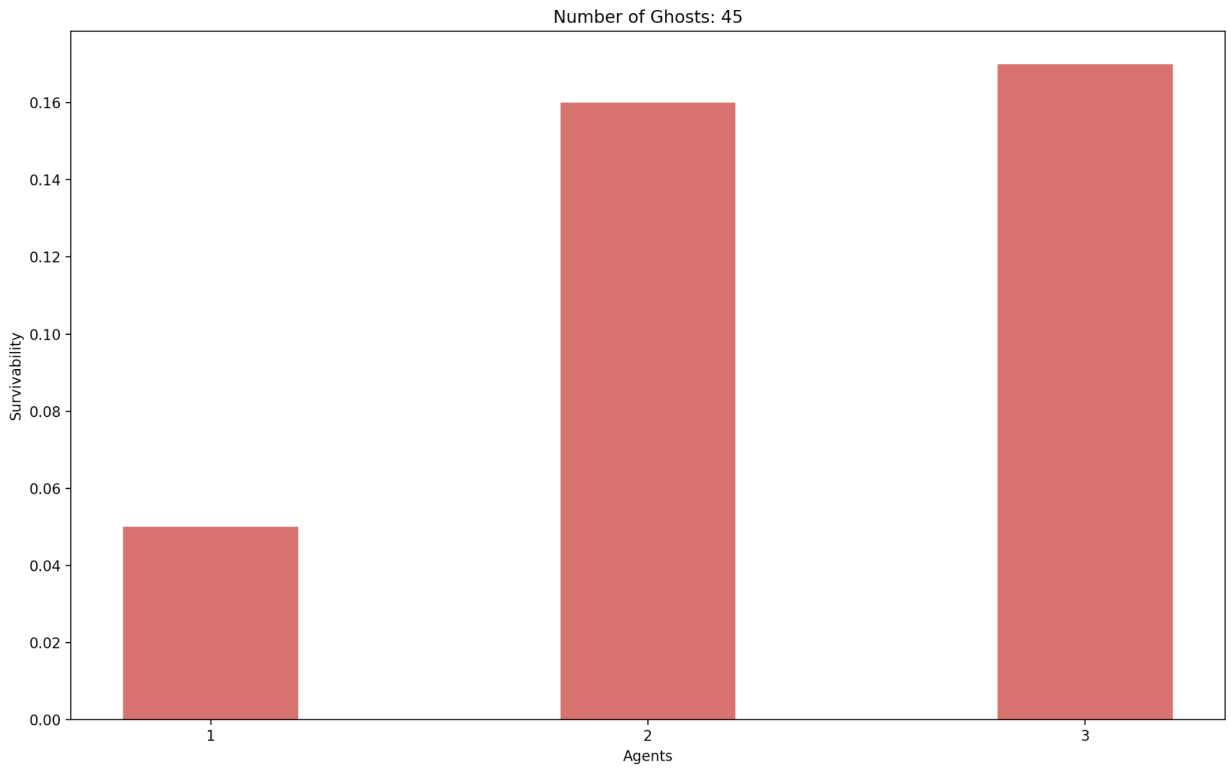
Comparing Agents 1, 2 and 3 for 5 ghosts in a maze



Comparing Agents 1, 2 and 3 for 25 ghosts in a maze



Comparing Agents 1, 2 and 3 for 45 number of ghosts



Q. If you are unable to get Agent 3 to beat Agent 2 - why? 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?

While our Agent 3 is not performing worse than Agent 2, it is also not clearly/decisively beating it. There could be few possible reasons for this:

(1) We are not doing enough tests on data points.
(2) Even though we are doing 100 simulations per data child (to the full possible depth) to calculate utility values, we probably still aren't calculating enough values to deal with bad RNG.
(3) We are always stopping agent 3 once it has made 1000 moves - which is an arbitrary limitation, but a required one nonetheless due to our computational bottlenecks. So we aren't even exploring the full extent of agent 3 due to hardware limitations.

Agent 4

Q. What possible ways can you improve on the previous agents? Does each planned path really need to be the shortest possible path? How could you factor in distance to ghosts? What do they do well? What do they do poorly? Can you do it better? Challenge yourself!

Our agent 4 takes into account two of the points highlighted here: (1) the planned path does not necessarily have to be the shortest path and (2) there is more we can do with the data from ghosts.

One thing we caught on to was that neither agent 2 nor agent 3 used the information from ghosts other than if they were coinciding with the paths while deciding/planning them. Distances from the nearest ghosts were used as an alternative to planning - when all plans failed, we defaulted to reacting to the closest ghost. What we felt was missing here was that the ghosts' regions were not taken into account during the planning.

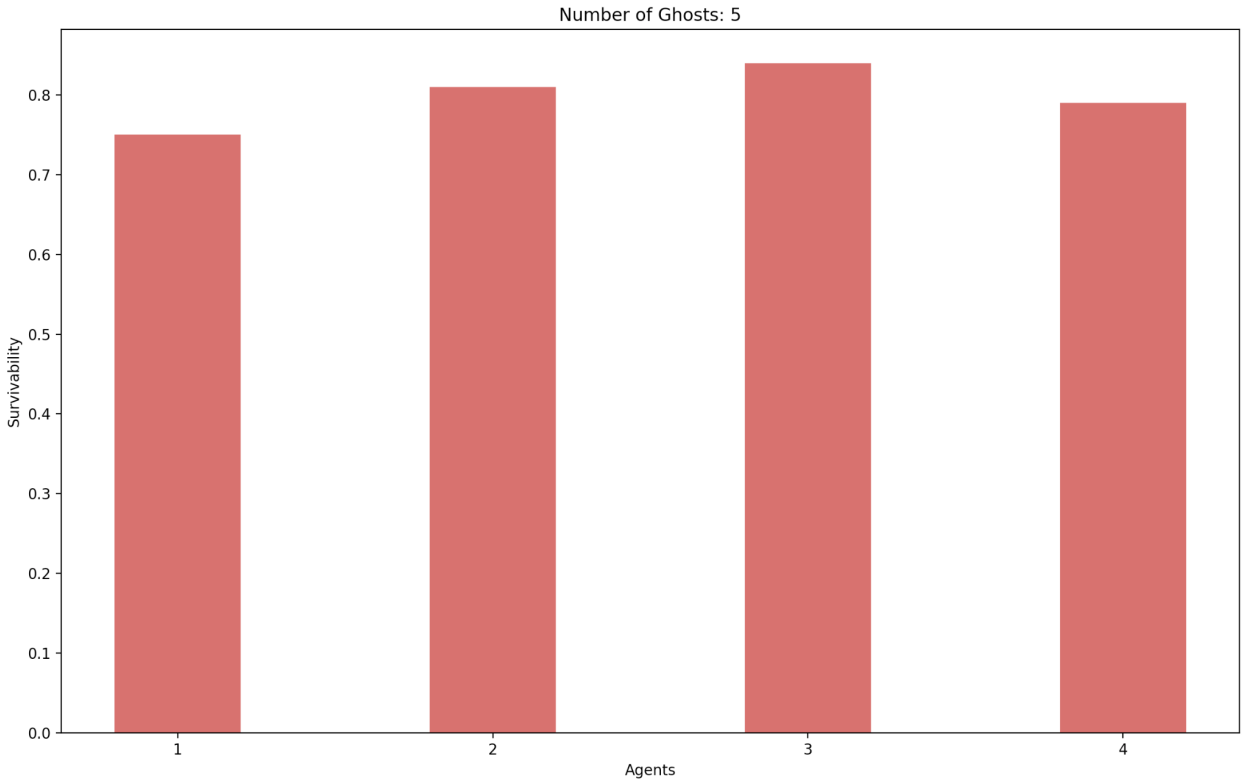
So we added the concepts of weights to punish/discourage paths that possibly went close to ghosts - we increased weights closer to a ghost's peripheral cells to increase the g-score value calculated in the A* algorithm. This was helpful in 2 ways - (1) because the planned path ran a lower risk of becoming immediately useless in the next time step/simulation step, and (2) we could integrate ghost regions better in the planning phase as well. Naturally, if our planning algorithm were to avoid available cells because they were in close proximity to ghosts, we would not always get the best possible paths. But we could save a considerable amount of time avoiding replanning (especially in the simulations in agent 3).

Another general change we made was to increase the number of total timesteps we would allow the agent to run for. Initially, due to computational bottlenecks, we had limited an agent to making at most 1000 moves in a maze (after which we were considering the run a failure). But in agent 4, we increased the maximum number of possible moves to 2500. We believed this would also help increase the survivability of the agent.

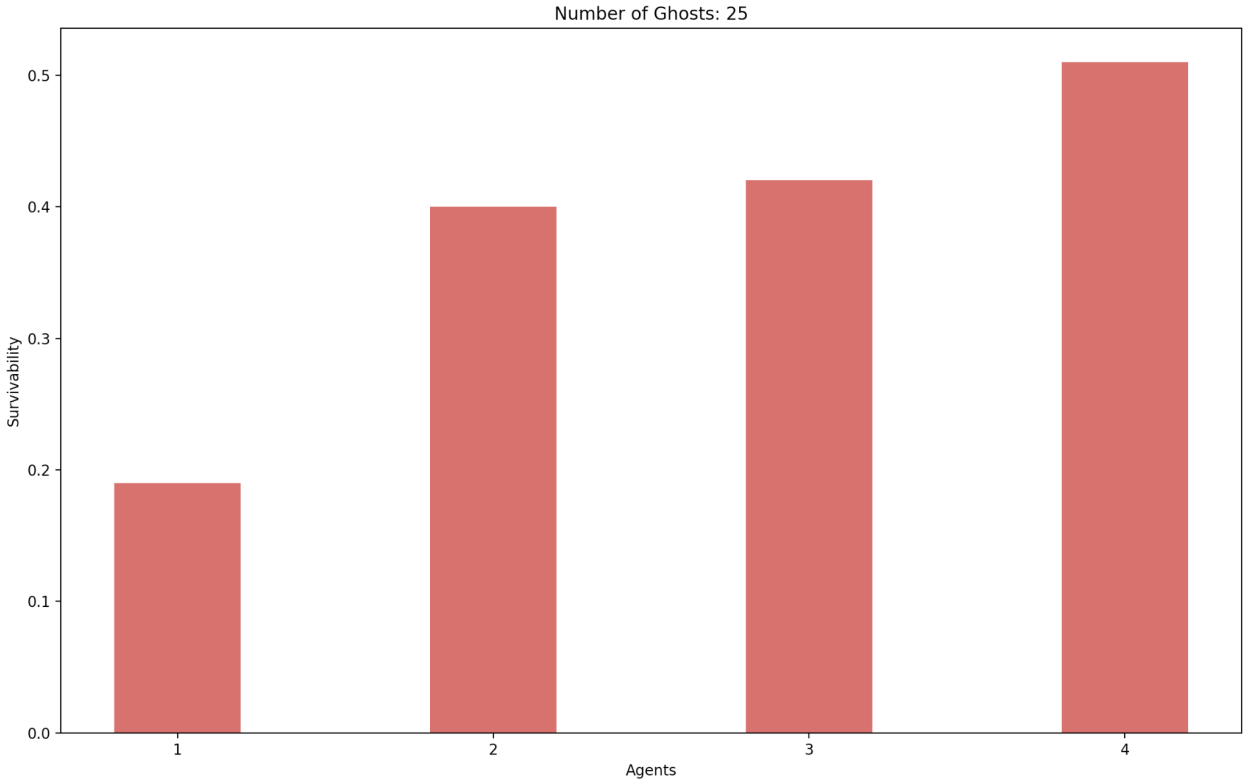
Q. How did your Agent 4 stack up against the others? Why did it succeed, or why did it fail?

Our Agent 4 stacked pretty well against the others (as can be observed in the graphs section). It was the best performer for data points which had considerably high number of ghosts and in general did not pose many outlier values across all agent tests (steps of 5, 25 and 45 ghosts).

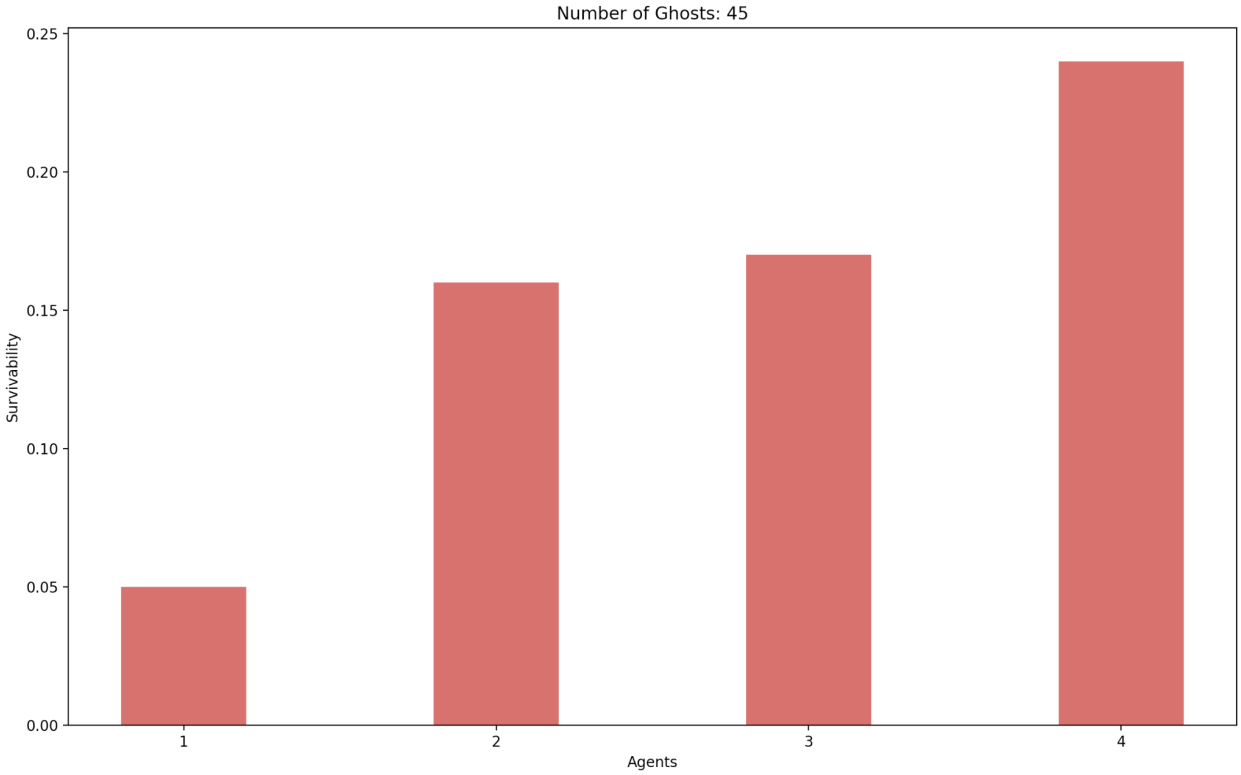
Comparing Agents 1, 2, 3 and 4 for 5 number of ghosts



Comparing Agents 1, 2, 3 and 4 for 25 number of ghosts



Comparing Agents 1, 2, 3 and 4 for 45 number of ghosts



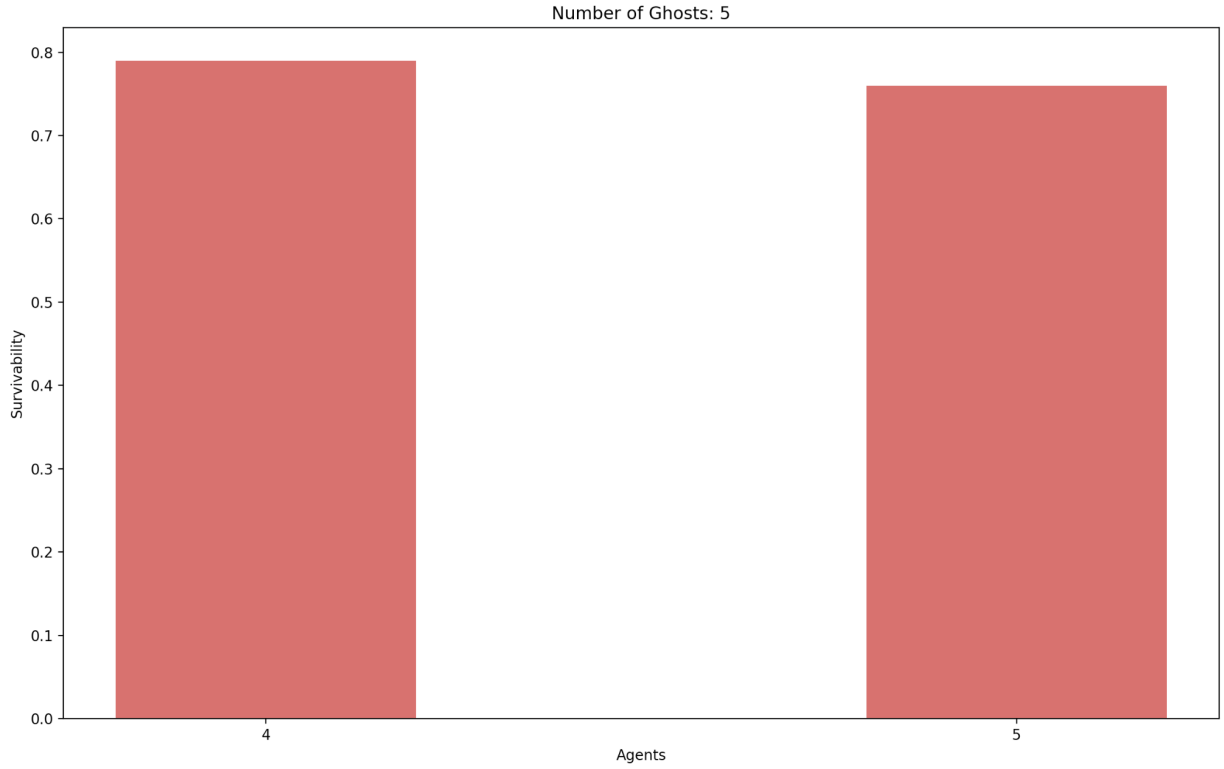
Agent 5

Q. 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?

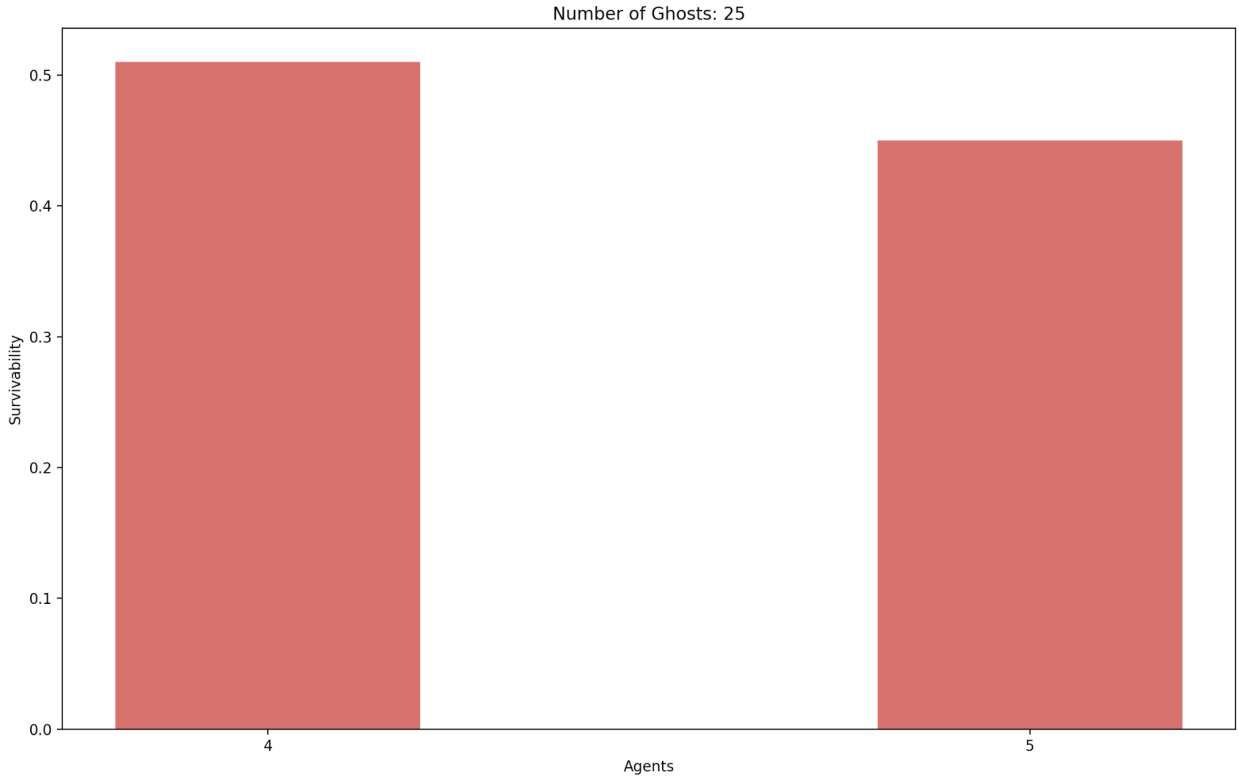
To accomplish the conditions for agent 5, changes were made in the path planning functions, which now could not punish paths passing close to ghosts within blocked cells. Also, the ghosts within blocked cells would not be considered while fleeing from ghosts in the absence of available paths.

Agent 5 created in this way could be compared directly only with agent 4 (as other agents did not make much use of additional information from ghosts). In general it performed slightly worse than agent 4 - which was expected due to it being less informed.

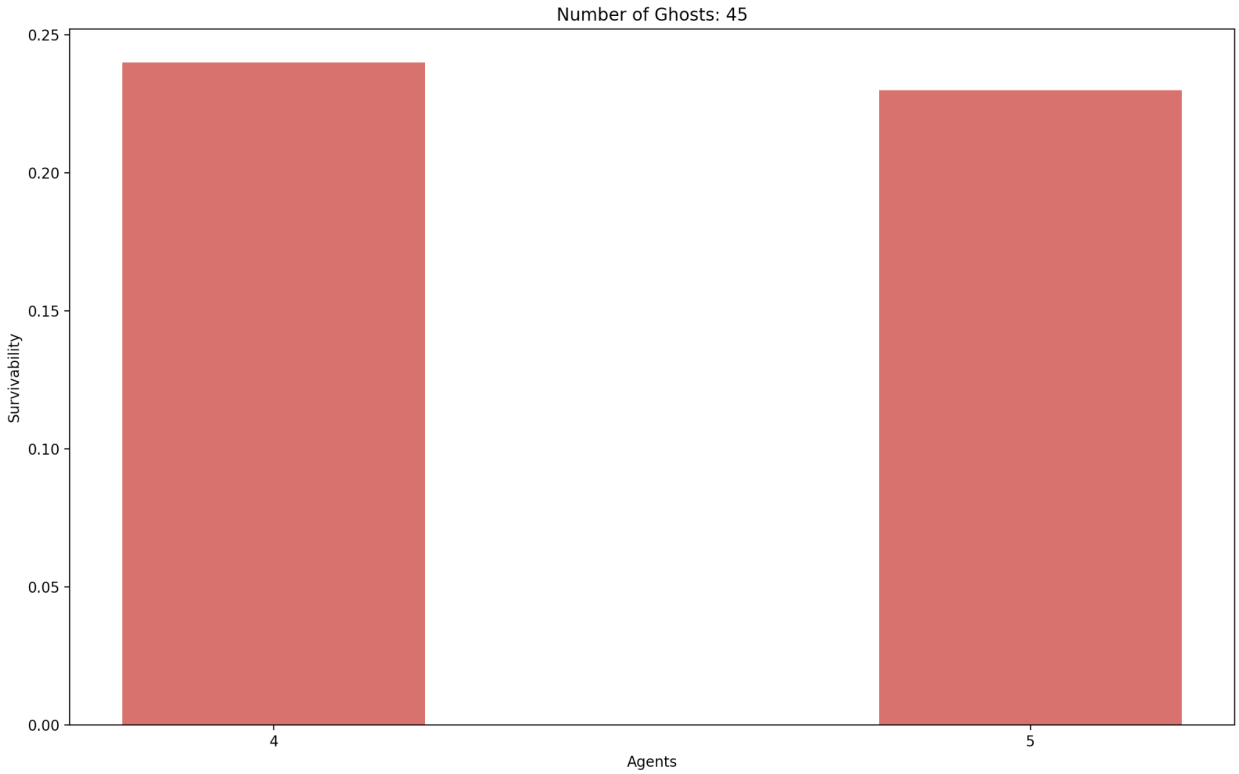
Comparing Agents 4 and 5 for 5 number of ghosts



Comparing Agents 4 and 5 for 25 number of ghosts



Comparing Agents 4 and 5 for 45 number of ghosts



Path Visualization

In general we have added steps in the README.md on our git repo on how to replicate a step-by-step visualization while running the agent. The following are the regions in general visited by the agents in green (and the ghosts in red for context). In cases where we have overlaps of paths, the display will show the path under the most recent traversing object (so if a ghost crossed a path taken by an agent in a previous time step, then we will print the ghost's path in the final image).

Agent 1

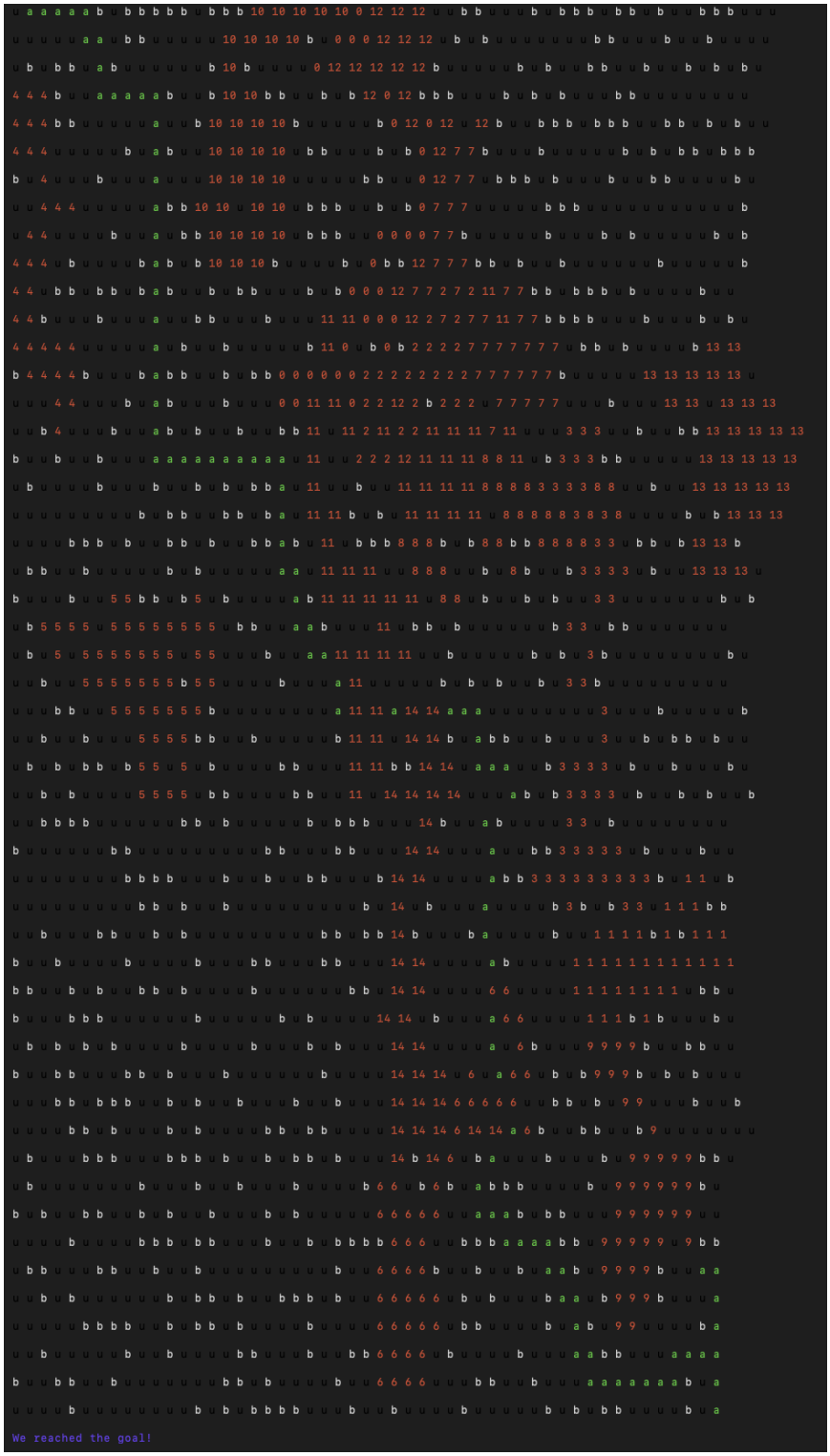
Here the agent pre-plans and continues on the path even if it means getting caught, as shown below:

Agent 2

Here the agent re-plans and hence is able to navigate the maze fairly when too many ghosts aren't blocking it:

Agent 3

Here the agent builds on Agent 2 and makes more informed decisions:

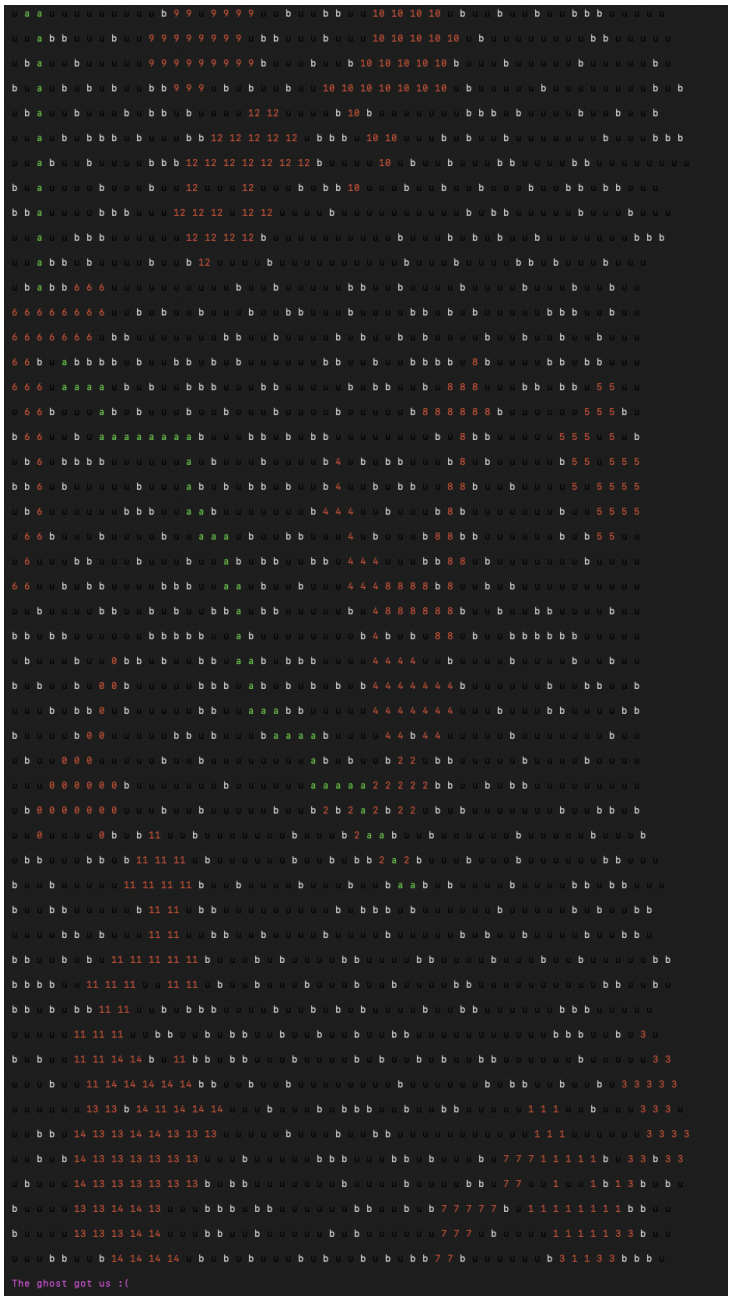


Agent 4

In Agent 4 we deployed a way to punish paths that are planned closer to ghost regions. This can be seen below at the start, where the agent sees it a more beneficial move to go down and continue doing so instead of right:

Agent 5

In Agent 5, we have no way of knowing or punishing movements in regions of ghosts in blocked cells. This can be seen in the way the agent gets caught below:



The region where the agent gets caught in an empty maze looks something like:

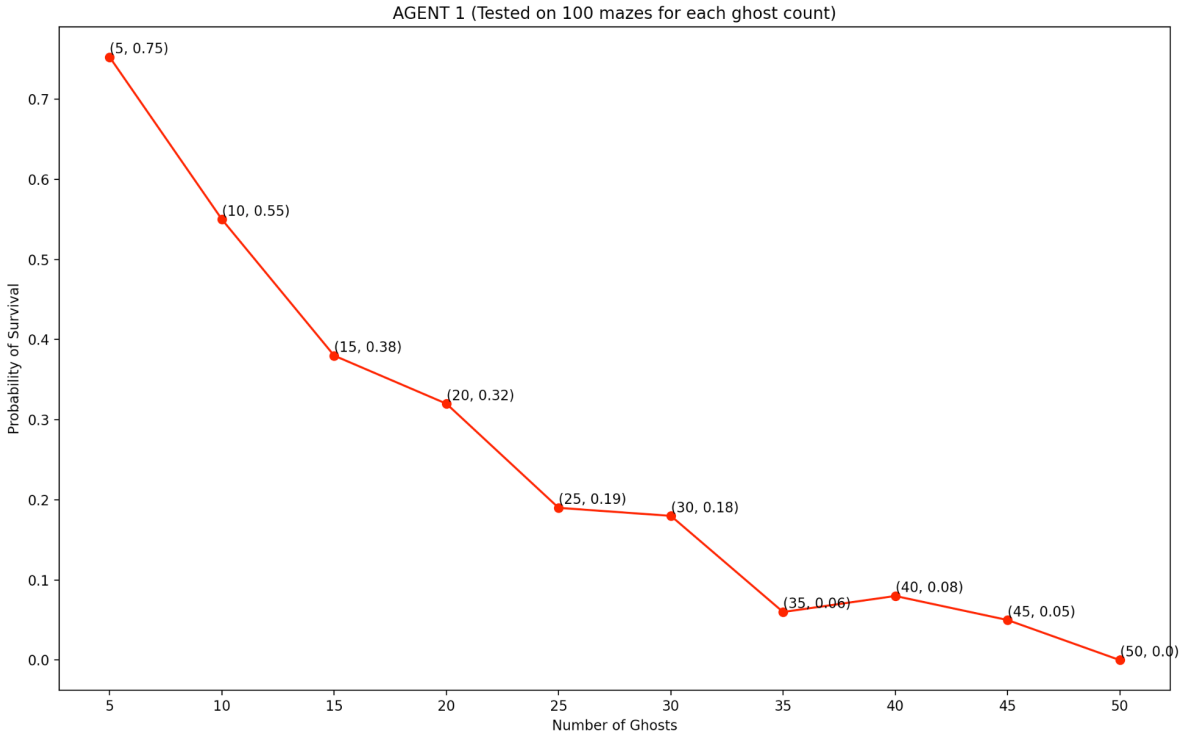
In which you can see that even if ghost 2 was in the region in one of the blocked cells, agent 5 would not be able to know about it and continue on moving deeper.

Graphs

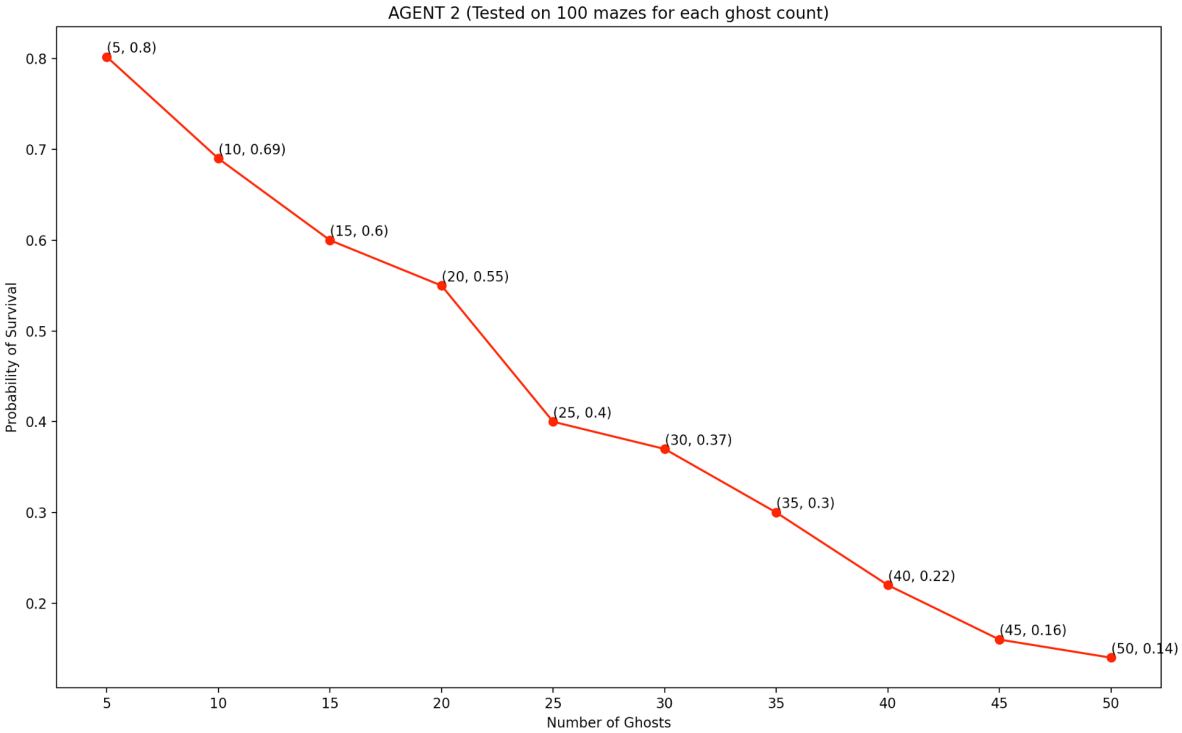
Q. When does survivability go to 0?

In general what we had observed for agents 1 and 2 was that the survivability went down very close to 0 for 70 ghosts (when we were checking by increasing the number of ghosts in steps of 5). But for better representation, we have displayed graphs for all agents upto 50 ghosts - because there were few cases where there were outliers.

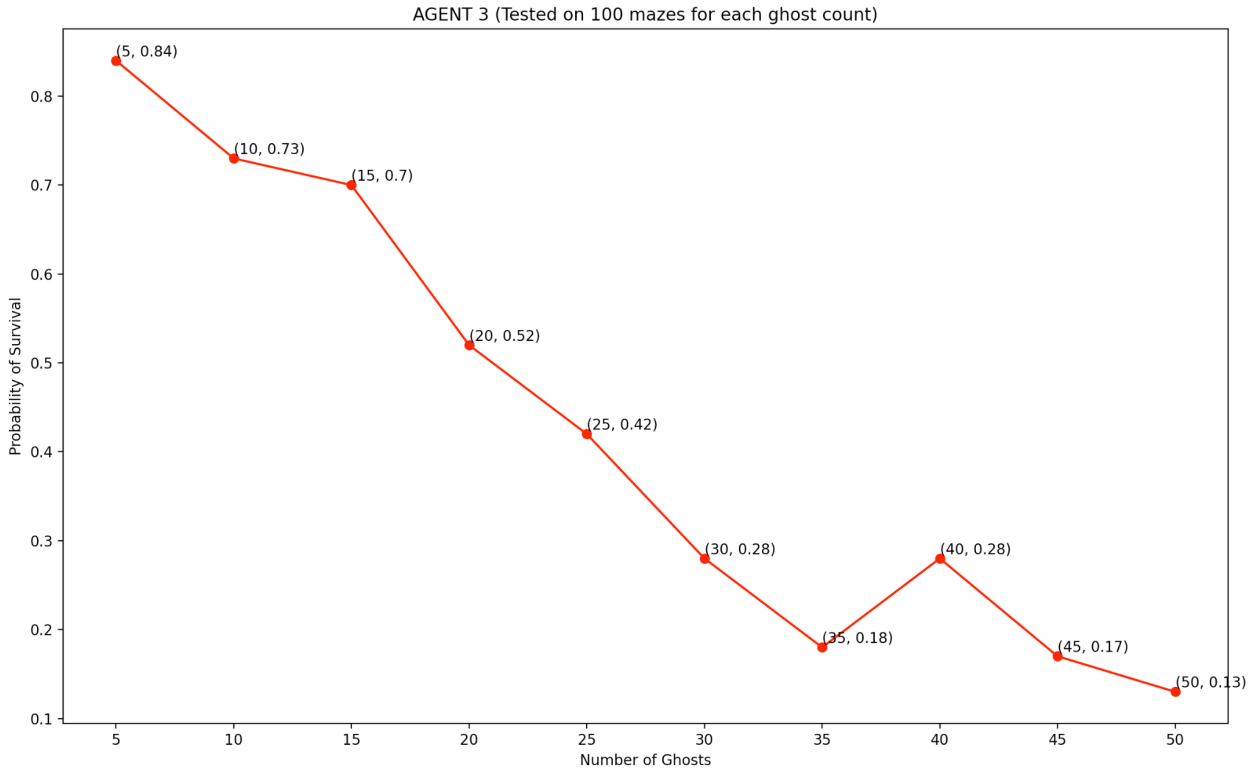
Agent 1



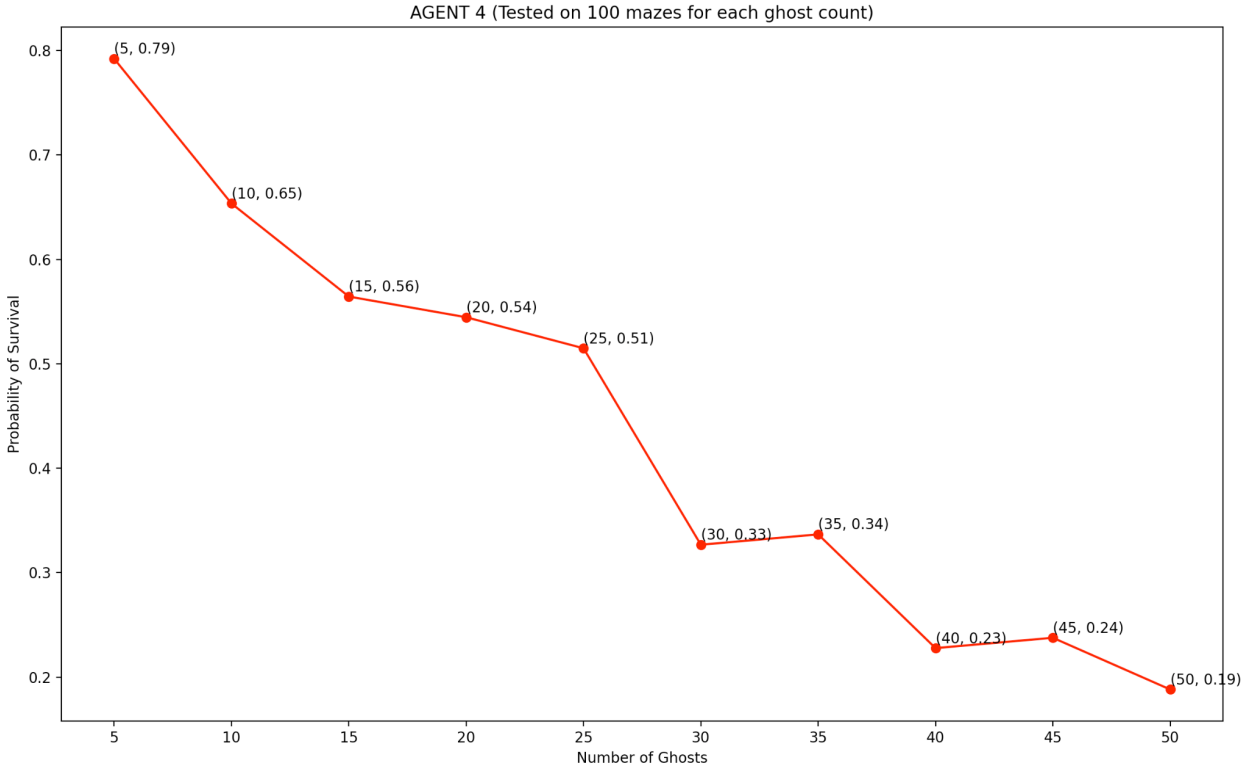
Agent 2


AGENT 2 (Tested on 100 mazes for each ghost count)
Probability of Survival
Number of Ghosts
0.8
0.7
0.6
0.5
0.4
0.3
0.2
5
10
15
20
25
30
35
40
45
50
(5, 0.8)
(10, 0.69)
(15, 0.6)
(20, 0.55)
(25, 0.4)
(30, 0.37)
(35, 0.3)
(40, 0.22)
(45, 0.16)
(50, 0.14)

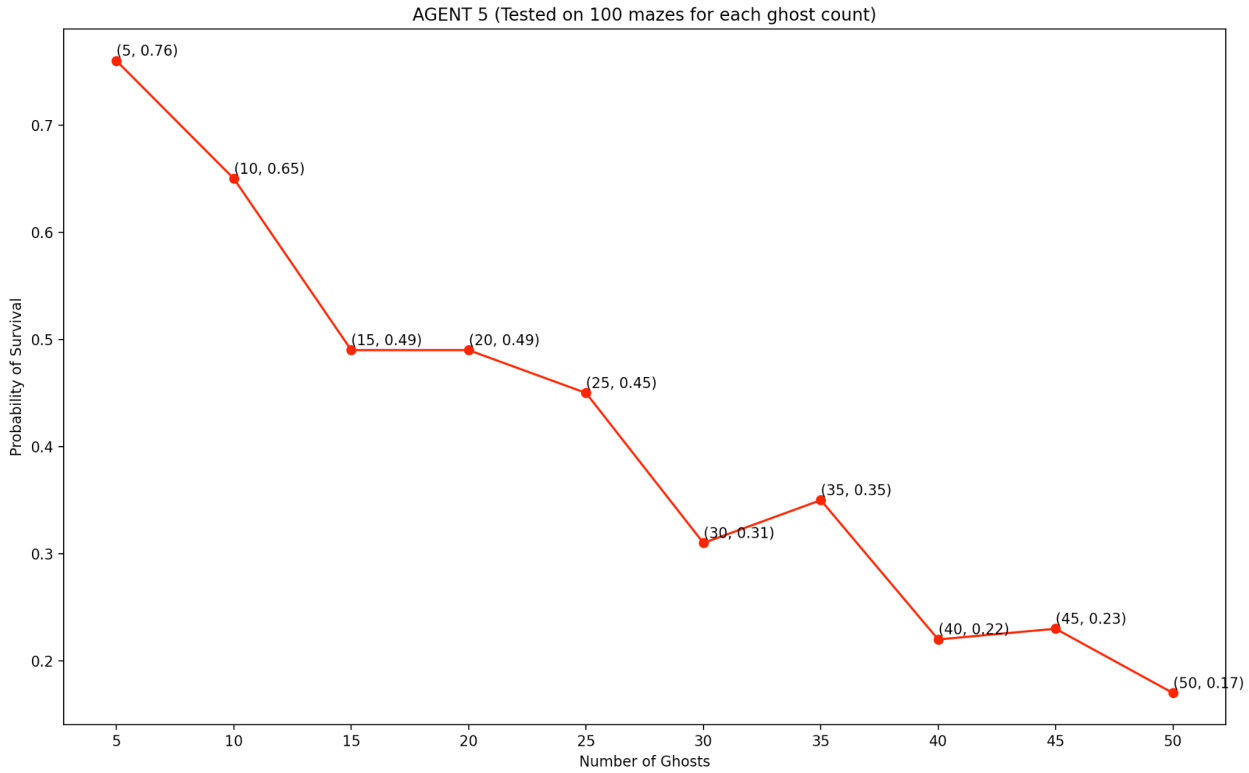
Agent 3


AGENT 3 (Tested on 100 mazes for each ghost count)
Probability of Survival
Number of Ghosts
(5, 0.84)
(10, 0.73)
(15, 0.7)
(20, 0.52)
(25, 0.42)
(30, 0.28)
(35, 0.18)
(40, 0.28)
(45, 0.17)
(50, 0.13)
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
5
10
15
20
25
30
35
40
45
50

Agent 4


AGENT 4 (Tested on 100 mazes for each ghost count)
Probability of Survival
Number of Ghosts
(5, 0.79)
(10, 0.65)
(15, 0.56)
(20, 0.54)
(25, 0.51)
(30, 0.33)
(35, 0.34)
(40, 0.23)
(45, 0.24)
(50, 0.19)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
5
10
15
20
25
30
35
40
45
50

Agent 5



Code Breakdown

We have many individual code files used for this project. The below sections will highlight codes or parts of codes used for each of the major requirements of the project. The full commented codes, maze CSV files, the stats gathered, and even the environment setup files for replicating our setup can be found in the zipped folder included in the submission. It also has an environment.yml file to recreate the environment we used (if you're using anaconda/miniconda) and the README.md file also has the steps.

Maze Generation

What we tried to achieve with the maze creation was an automated pipeline that would generate N (51 * 51) mazes and would store them as individual CSV files in a maze_csv folder. There were three main aspects to this pipeline:

maze.py

This code takes command line inputs and generates a random maze with the 0.28 blocking rule mentioned in the project write-up, then checks if the maze is correct (by using a function from

bdbfs.py), and then if the maze can be traversed from start to goal, then it saves the maze created as a CSV file (by using a function from csvops.py).

bdbfs.py

The driver code here is the Bi-Directional BFS that we apply in the function **doesPathExist** that checks if the maze has some optimal path between start and goal.

csvops.py

The driver code in the context of matrix creation is **generateCsv** which is used to create a CSV file based on the matrix provided.

Main

Everything that happens to run agents on mazes, or try simulations goes through the code main.py. It has many facets to it, but the common ones will be covered here and any specific points will be addressed under specific agent sections. Another code useful here is csvops.py, because it has the function to load CSV files for maze runs. Lastly all statistics related to survivability of the agent are also captured and stored using the main driver code.

main.py

This code loads the CSV files for mazes saved in the past to python arrays, and automatically spawns ghosts and an agent depending on values passed to it. Once the spawning is done, it runs a full “game” for the agent trying to traverse the maze with a specific number of ghosts in it. After multiple tests, it also automatically generates CSV files with survivability statistics for the agents for different numbers of ghost spawns and stores them in ./agent_csv/agent{num}.

csvops.py

The driver code in the context of matrix creation is **readCsv** which is used to load CSV data files in matrix arrays.

Ghosts

The important codes for the ghost objects are in ghost.py and the spawning location is checked using the BD-BFS algorithm in bdbfs.py

ghost.py

This code has the ghost class (for the ghost objects in the maze) which has necessary attributes like **row** and **col** indicating its position on the maze and methods like **spawnGhost** and **moveGhost** that spawn the ghost and move it respectively. The **spawnGhost** method makes use of the BD-BFS algorithm in **bdbfs.py** to check if the ghost is spawned in a location reachable from the agent.

Agent 1

Agent 1 is the approach where we plan a path at the start after spawning, and then just move along that planned path without any concern for the surrounding environment (i.e. the position of the ghosts on the maze). While the approach is fairly straightforward, where we have planned the path using an A* algorithm in the **planPath** method of the **agent** class in **agent.py**, there are other things worth mentioning in **main.py** and **agent.py** that form the basis of our pipeline for agents 2 through 5.

main.py

Here we have quite a lot of things happening:

(1) The code accepts few prompted arguments like the start and end maze numbers, the maximum number of ghosts, the step size by which we want to increase the volume of ghosts, and finally the agent number that needs to run on the data points. Say we were to pass start and end maze numbers as 1 and 100, maximum number of ghosts as 10 with a step size of 1, and the agent number as 1, then the main.py code would loop through ./maze_csv/maze1.csv to ./maze_csv/maze100.csv - chunk them in batches of 10 - and run agent 1 starting with 1 ghost upto 10 ghosts for each chunk.

(2) For each chunk/batch of mazes we are processing for a specific number of ghosts (say G) for a specific agent (say A), we are creating a ./agent_csv/agentA/GhostX.csv file that stores data in terms of 0's and 1's on whether the agent failed or survived on a maze. These CSV files are then used to plot the data separately as shown in the Design and Graphs sections respectively.

(3) For each run on an individual maze for a specific number of ghosts, we have a while loop that is going to be our stub going ahead where we will be also adding new agents. The loop basically actually invokes methods that move the agent and ghost objects and it also checks if the run has ended (i.e. the agent has either reached the goal, has been caught, or has exceeded the maximum number of timesteps). In later agents, things like replanning, simulations, etc. are also invoked in this loop.

This workflow helped us plug other agents fairly easily and also automated a lot of our testing efforts (we could just keep our machines on charging and keep the code running for hours if required).

agent.py

The code has the main agent class with some attributes and methods that will be discussed in detail (because they are just re-used for the other agents - with some new methods introduced by each agent):

(1) We have the attributes **row** and **col** which indicate the position of an agent and **name** which indicates the type of the agent (i.e. agent1, agent2, etc.).

(2) We also have another attribute called **heuristics** which is a map that stores the number of steps it would take the agent from any node on a specific maze to the goal node of the maze. This attribute is always calculated at the start of every run invoking an agent class method called **getBaseHeuristics** which uses another method called **calcHeuristsics**

to find optimal path lengths from each of the nodes to the goal node using a BD-BFS. The values in the **heuristics** map are then used as measures to plan paths.

(3) The **planPath** method is used to apply an A* algorithm to decide a path using the **heuristics** map we have available in the agent object to return a list of possible steps that can be taken to reach from the start to the goal node.

(4) There are also helper methods **isValidMove** and **createPath** that are used to assist checking for only valid neighbors and finally create a list for the path planned respectively in **planPath**.

Agent 2

Agent 2 just makes sure that at a given timestep its current path does not get blocked off by ghosts. It first checks if there is even a need to replan the path decided before starting the run using a new method in the **agent** class in **agent.py** called **doWeReplan** and then either replans using the same **planPath** method that Agent 1 uses (but this time from its current position) or does not replan. In case there are no paths available (ghosts blocking all possible routes from current timestep), then the agent just moves furthest from its nearest ghost using a new method **stayAwayFromGhosts** in agent class. There are additional steps added in the running loop to include workflows for Agent 2 in **main.py**.

agent.py

The new **doWeReplan** method in agent class checks if the path available to the agent object (through any previous timestep) has any ghosts blocking it for the current timestep. The **stayAwayFromGhosts** method first finds the ghost closest to the agent object's current position and then helps decide which neighboring cell to the agent object would increase its distance to the ghost the most.

Agent 3

Agent 3 as discussed in the design section uses a Monte Carlo approach for calculating utility values for each of the valid neighbors of the current location of the agent, and then moves to the neighbor with the highest utility. The way this utility is calculated is by averaging the outcomes of 100 simulations of Agent 2 from each neighbor to a simulation ending state (which could be the goal node or the node at which a ghost catches the agent). The code that helps realize this is the function **monteCarlo** in **main.py**. When no utilities worth exploring are available, Agent 3 then defaults to the same behavior as Agent 2 (i.e. it checks for paths that currently do not have any ghosts on them, or it simply tries to move furthest from its nearest ghost).

main.py

There are additional steps added in the running loop to include workflows for Agent 3. For the simulations, we find valid neighbors of the agent's current cell using the **findChildren** function and then run the function **monteCarlo** on them. Based on the utility values we either select one of the children or default to the Agent 2 workflow.

Agent 4

Agent 4 improves on Agent 3 in 2 ways: one is that it is allowed longer runs with an increased number of permitted timesteps (even if it computationally hurts us - and it did, by almost doubled the time for each run), and second is that it takes into consideration ghost regions and increases path weights in those regions to punish selecting such paths.

What the latter achieves is a situation where we may not have to replan as much in average cases, and in the process we may give up an “optimal” path for a less dangerous one. The codes that are changed for Agent 4 are **agent.py** with a new **planWeightedPath** method that adds weighted paths to the g-score and another method **calcStepWeight** which actually calculates the weight of each path. In the past few agents every cell we would move to would have a uniform weight (of 1) - so we did not want to change that workflow. There are also changes made to **main.py** to invoke the necessary methods in the Agent 4 workflow.

agent.py

There is a new method **planWeightedPath** introduced to calculate weighted path using data only from all ghosts on the maze. This method uses another method **calcStepWeight** to calculate weights - which assigns compounded values for regions around the ghosts (i.e. when we have a cell shared in two separate ghost regions, we want to increase its weight more to make a path through that cell *more* unfavorable).

Agent 5

Our Agent 5 is just a slightly modified Agent 4 who cannot make use of any data from ghosts within blocked cells. So this means that any calculation we do for paths in **agent.py** are slightly modified. There are also naturally some changes in **main.py**, but they are just to introduce the new agent in the workflow.

agent.py

There is a new method **planWeightedPathVisible** introduced to calculate weighted path using data only from visible ghosts and there were small changes made to **calcStepWeight** to change its behavior and ignored walled ghosts based on a new input **invisibleCheck**.

Data Visualization

There were 2 types of data visualizations done - one in **main.py** to visualize the agent/ghost paths or for real-time display, and the other is the data visualization for the graphs in the report which is done in **data.py**.

main.py

In this code we are printing to the terminal the positions of the agent and ghosts using print statements and coloring them differently using the colorama module using the **dispMaze** function.

data.py

In this code we are using pandas and matplotlib to plot the survivability of the agents for different numbers of ghost spawns (which are gathered and stored in ./agent_csv/agent{num} by **main.py**).

comparingAgents.py

In this code we are using pandas and matplotlib to plot bar graphs between the survivability of different agents for a particular number of ghost spawns (which are gathered and stored in ./agent_csv/agent{num} by **main.py**).

References

Using colorama:
https://www.geeksforgeeks.org/print-colors-python-terminal/

Visualizing the mazes:
https://medium.com/swlh/fun-with-python-1-maze-generator-931639b4fb7

Questions regarding data structures:
https://stackoverflow.com/questions/4230000/creating-a-2d-matrix-in-python

BFS code understanding (referred as an idea for BD-BFS):
https://www.geeksforgeeks.org/breadth-first-traversal-bfs-on-a-2d-array/

CSV reference:
https://www.pythontutorial.net/python-basics/python-write-csv-file/

Understanding of A*:
https://en.wikipedia.org/wiki/A*_search_algorithm

Concepts of BD-BFS/Monte Carlo:
Dr. Cowan's Lectures