Project One Report

Author: Patrick Liang – pzl4

Design:

Firstly, I chose to use the Java programming language because it is fast, does not need memory management, and is able to multi-thread.

When designing this project, I decided I would need a Maze class, an Agents class, and a Main class to run the agents. I started with the Maze class.

Maze:

I would need a constructor, a maze generator, and a maze verifier for the Maze class. The maze verifier was the most complicated feature of this class. I had a couple of options: DFS, BFS, or A*. I ultimately went with BFS because although DFS is faster because of its better Space Complexity(O(b^(d*)) vs. O(b*m)), the maze verifier should be BFS because the downside of the Space Complexity was exactly what I wanted. Since I wanted to calculate the distance from each visitable cell to the jail cell, T, BFS is necessary because it searches every cell to be visitable.

Search:

For the Agents class, I initially planned for a findPath method and Agents 1-5. However, I quickly realized that these methods were insufficient for the assignment. One big issue I encountered was the lack of a tuple class in Java. A nice workaround that I found is to make a static class, which I call index. The static index class implements three different constructors, each proposed for a specific use case. I added a couple of getters to use for PriorityQueues in other functions in the Agents class.

For the search algorithm, which I call findPath, I originally designed it to be BFS because in a 2D array, the solution would be optimal and because I did not have the foresight of how computationally taxing Agent Three would be. However, after attempting to run Agent Three, I quickly realized that although the solution was optimal, the run time left much to be desired. From there, I implemented A*, with the heuristic I used to be the distance traveled so far, *g*, 1 + distance from the current cell to the end, *h*, cell distance found from verifyMaze in the Maze class is equal to *f*. After implementing A*, I found that all agents performed much better than BFS(regarding speed).

Agents:

Agent One is simple on its own. Once the path is found, travel on the path, and if the Agent either moves into a ghost or a ghost moves into the Agent, record it as dead.
Agent Two is where the assignment is not as straight - forward. Because of the notion of moveAway, the distances of ghosts to the Agent's current position are needed. This suggests that a PriorityQueue should be used to speed up the lookup of the closest ghost, which is used to optimize runtime for Agent Two. The order of the PriorityQueue of ghosts is determined by the distance to the Agent. To calculate the distance of ghosts most efficiently, I update the distance every time a ghost moves. I use the Manhattan distance to calculate distances because ghosts cannot travel diagonally. Even though the assignment says to replan at every step, it is not

necessary to replan at every step unless there is a ghost in the way of the current shortest path. One way to implement this would be to go through the path and verify if a ghost occupies any cell in the path every time the agent moves.

The design of Agent Three is relatively simple, simulate Agent Two eight times and decide which square to travel towards. In order to simulate all possible moves, I added the modifier {0, 0} to the cell calculator to simulate in place. The biggest challenge I encountered in Agent Three was the tiebreaker; what would be sufficient to break a tie of equal utility? With the guidance of Prof. Cowan, I decided to use the distance of the possible new move compared to the previous move with the greatest utility(e.g., {0, 1} distance to goal is eight and {0, 2} distance to goal is 7, it would pick {0, 2} as the new cell). Moreover, if the distances were equal, I would then opt to go in the direction of down(*row* + 1) or right(*col* + 1). If Agent 3 decides there is no successful path in any of its projected cells, move away from the current cell. Depending on how many simulations you are running, it does not guarantee that success is impossible. The more simulations that are happening, the more accurate the simulations become.

Agent Four is also relatively simple; change the heuristic so that the Agent values survivability over speed. With this objective in mind, I changed the heuristic from the shortest cell distance to the distance to the closest ghost. I would iterate through all ghost coordinates and calculate the Manhattan distance of the current cell and the ghosts. The heuristic would then choose the closest ghost and return the negative distance. The distance returned has to be negative due to the nature of the PriorityQueue, which orders in ascending order. This results in an *f* value that becomes smaller as the distance between the Agent and the ghost grows larger. After implementing the heuristic, I decided to experiment if it was possible for Agent Four not to need to simulate at every turn. The condition I chose was the distance of the closest ghost to Agent. A new path would be found if the distance were under a certain threshold. As I tested this condition, I found that the survivability of the Agent was drastically reduced if the distance was under 100, which is the max distance a ghost could be. So I concluded that Agent Four, just like Agent Two, would have to replan at every step.

Agent Five is simply an add-on to Agent Four with no awareness of ghosts in walls. The change I made for Agent Five to survive more optimally was adding an array of last-seen positions of ghosts, lastSeen; lastSeen only updated when a ghost was moving outside a wall. Moreover, instead of using PriorityQueues, I used arrays to store ghosts in both arrays because I could then synch both arrays when updating positions. lastSeen is useful to keep track of because if a ghost enters a wall, at least you know where the ghost last entered from; thus, you avoid that area even if you do not know where the ghost is headed.

Agents Helper methods:

addGhosts:

Added ghosts randomly, within visitable cells, and stored in a PriorityQueue

moveAway:

Agent moves away based on the distance of each neighboring cell to the closest ghost.

moveGhosts:

Moves ghosts randomly and updates the distance to the Agent’s current position at each turn.

getDistance:

Calculates the Manhattan distance between two coordinates.

indMin:

Finds the smallest distance from an array of ghosts.

copyGhosts:

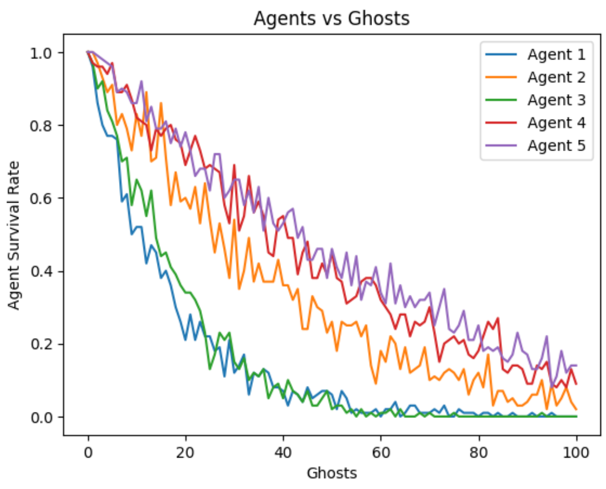
Creates a copy of ghosts with no references.

RunAgents:

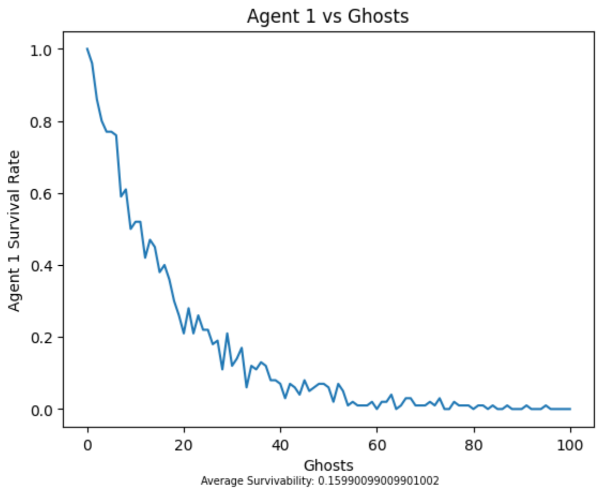
Utilized Java’s Thread class to run 101(0-100 ghosts) instances of all Agents simultaneously to reduce the time of runs. I ran 100 iterations of each Agent for an *x* number of ghosts. In addition, I generated a new random maze for each Agent at each iteration. I print out every survival rate for each Agent at the end. I then utilize piping in the Linux terminal to store survivability data into a .txt file. I then use Python to parse the data and generate graphs.

Graphs:

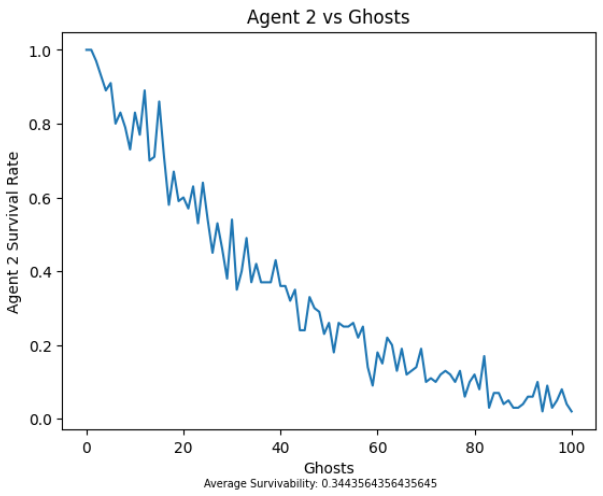
All Agents vs. Ghosts:



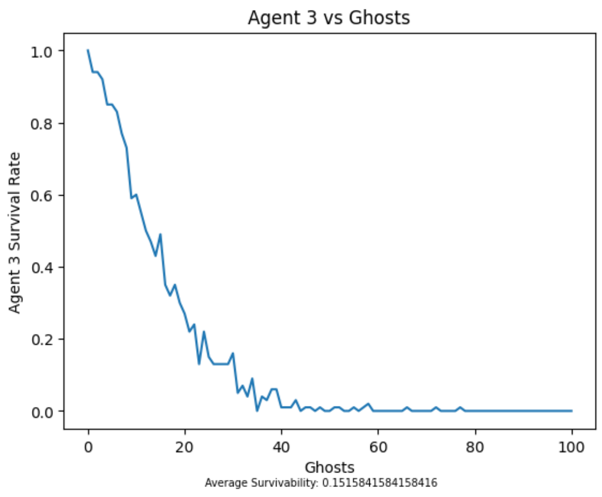
Agent 1:



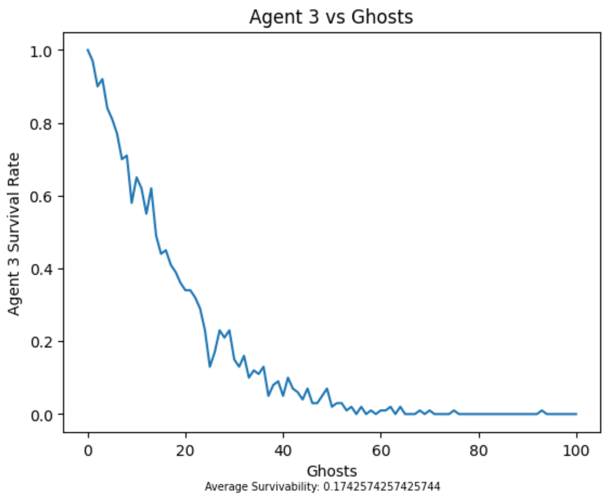
Agent 2:



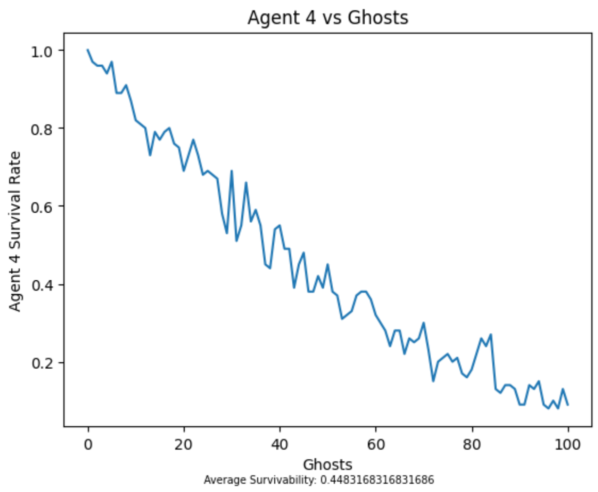
Agent 3, 8 Simulations:



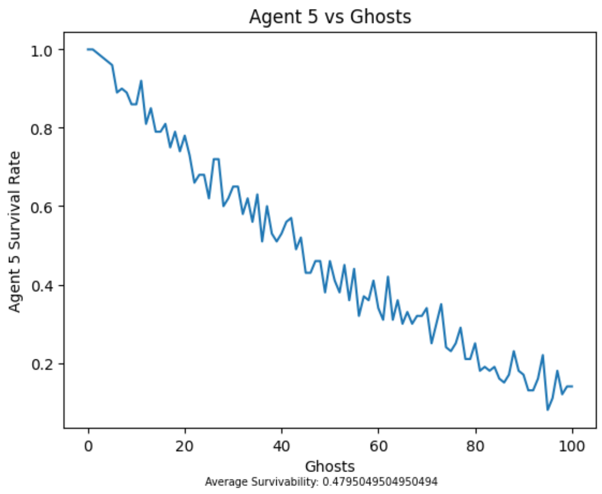
Agent 3, 3 Simulations:



Agent 4:



Agent 5:


Agent 5 vs Ghosts
Agent 5 Survival Rate
1.0
0.8
0.6
0.4
0.2
0
20
40
60
80
100
Ghosts
Average Survivability: 0.4795049504950494

Analysis:

For Agents One, Two, and Three, Agent Two outshone the other two by far. With an average survival rate of 0.34, it is more than twice as likely to live compared to Agent One and Agent Three. This makes sense compared to Agent One, which has no foresight and replanning ability. However, it does not make any sense compared to Agent Three(8 simulations), which is the worst Agent with a survival rate of 0.152. Just like Agent One, Agent Three peters once the number of ghosts exceeds 10. Before that, Agent Three performs equally as well as Agent Two.

My hypothesis for Agents Three is worse than Agent Two in average survivability is that the ghosts are too random to account for. More importantly, running Agent Two to completion as a sign of utility is probably misleading regarding the nearby ghosts. In addition, the more simulations you run, the higher likelihood that there will be one or two runs that survive. These runs are "lucky" in that it escapes nearby ghosts and does not run into many ghosts afterward. Because of this, Agent Three may head in the direction of nearby ghosts, thus causing an early death when it should default to move away. These lucky simulations are also why I hypothesize it is worse than Agent One. Because Agent Three may blindly run into a ghost where it thinks survivability is the best. This hypothesis is only for when the number of ghosts exceeds 10, where the lucky simulations stand out more. I believe if I used Agent 4 as a utility indicator Agent Three would outperform all other Agents.

This hypothesis is supported by the data I conducted with three simulations per turn, which generated a .02 increase in average survivability(0.17).

Agent Four is the second-best Agent. With an average survival rate of 0.45, nearly 50% of all runs survive. It succeeded because it uses Agent Two's constant replanning in addition to a heuristic that prioritizes survivability rather than the shortest distance. However, it still considers the shortest distance**.**

Blindness:

Agent One:

It would not be affected since it does not account for ghosts, to begin with.

Agent Two:

It would be marginally affected. This is because it replans at every step and because ghosts are likely to move back out of the wall where they can be seen.

Agent Three:

It would be marginally affected because it utilizes Agent Two as a measure of utility.

Agent Four:

It would be more affected because it utilizes ghosts' positions to plan its path, and if a ghost is in a wall, it may value the path where the ghost can pop out and kill the Agent. However, because of replanning, this effect would be mitigated.

Agent Five:

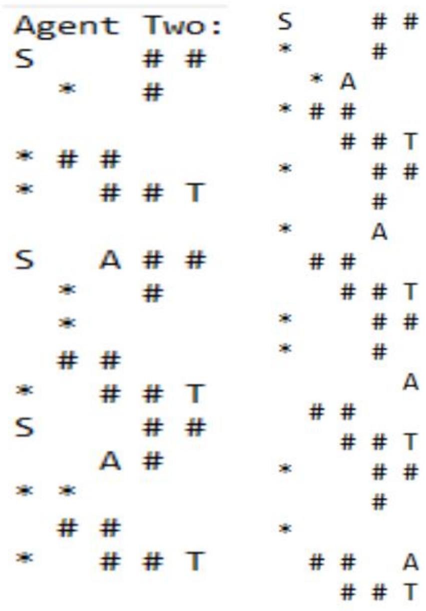
Agent Five is designed to remember ghosts' positions only when they are outside a wall. So, for example, if a ghost moves into a wall at {5, 5} from position {4, 5}, Agent Five would see that the ghost was last seen at {4, 5} and not at position {5, 5}. With this new way of tracking ghosts, Agent Five was not only able to keep survivability up but was the best Agent. I hypothesize that Agent Five outperforms all others because when a ghost enters a wall, a path opens up. Because Agent Five only remembers the last positions of ghosts outside of walls, paths that open up for other Agents remain closed for Agent Five. By having paths remain closed in Agent Five, it means that the Agent will not travel in dangerous areas, thus upping Agent Five’s survivability compared to all other Agents.

<u>Paths:</u>

Reading directions: Read each column from top to bottom

Context: Three paths visualized on a 5x5 maze with 3 ghosts

Legend: S: Start; A: Agent; #: wall; *: ghost; T: Goal;

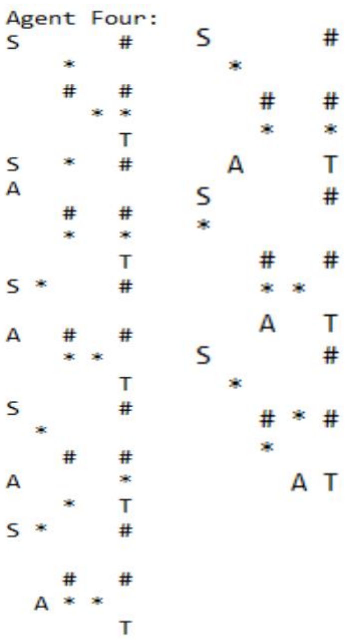


```
Agent Two:
S     # #
  *   #

* # #
*   # # T

S   A # #
  *   #
  *
  # #
*   # # T
S     # #
    A #
* *
  # #
*   # # T
```

```
S     # #
*     #
  * A
* # #
    # # T
*     # #
      #
*     A
  # #
    # # T
*     # #
*     #
        A
  # #
    # # T
*     # #
      #
*
  # #   A
    # # T
```

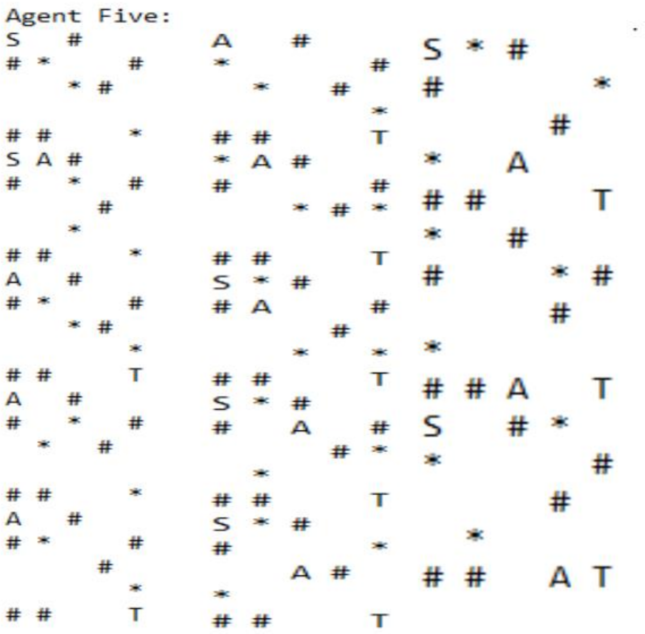


```
Agent Four:
S       #
    *
    #   #
      * *
        T
S   *   #
A
    #   #
    *   *
        T
S *     #

A   #   #
    * *
        T
S       #
  *
    #   #
A       *
    *   T
S *     #

    #   #
  A * *
        T
```

```
S       #
  *
    #   #
    *   *
  A     T
S       #
*
    #   #
    * *
    A   T
S       #
  *
    # * #
    *
      A T
```



```
Agent Five:
S   #
# *     #
    * #

# #     *
S A #
#   *   #
      #
    *
# #     *
A   #
# *     #
    * #
        *
# #     T
A   #
#   *   #
  *   #

# #     *
A   #
# *     #
      #
        *
# #     T
```

```
A   #
*       #
  *   #
        *
# #     T
* A #
#       #
    * # *

# #     T
S * #
# A     #
      #
    *   *
# #     T
S * #
#   A   #
      # *
  *
# #     T
S * #
#       *
    A #
*
# #     T
```

```
S * #
#       *
      #
*   A
# #     T
*   #
#     * #
      #
*
# # A   T
S   # *
*       #
      #
  *
# #   A T
```