CS520 Project 1: Voyage Into The Unknown

Daniel Ying (dty16) Sec 198:520:01
Zachary Tarman (zpt2) Sec 198:520:01
Pravin Kumaar (pr482) Sec 198:520:03

October 18, 2021

Submitter: Daniel Ying

Honor Code:

I abide to the rules laid in the Project 2: Partial Sensing description and I have not used anyone else's work for the project, and my work is only my own and my group's.

I acknowledge and accept the Honor Code and rules of Project 2.

Signed: Daniel Ying (dty16), Zachary Tarman (zpt2), Pravin Kumaar (pr482)

Workload:

Daniel Ying: Coded Agent 1 and Agent 2. Formatted the report in LaTeX. Recorded data and graph for Agent 1 and Agent 2, graphed Agent 3. Wrote Observation for Agent 1, Agent 2, and Agent 3, and agent 4. Wrote comparison of agents 1,2,3,4.

Zachary Tarman: Coded Agents 3 and 4. Recorded data and created graphs for Agent 3. Wrote about designing Agent 4. Wrote Information Representation and Workflow for Agents 3 and 4. Wrote Computational Issues section.

Pravin Kumaar: Coded Agent 2. Recorded data for Agent 1 and 4.

Together: Brainstormed the Algorithm of Agent 4. Discussed the inferences to the improved Agent 4 algorithm. Discussed problems in the assignment and code.

Problem 1

DIFFERENCE BETWEEN PROJ1's AGENTS AND Proj2's AGENTS:

For project 1 (Voyage Unknown), our tie-breaker for our blocks' f value, we switched focus onto the heuristic value then to g value. But for Project 2 (Partial Sensing), on top of project 1's tie-breaker, we added a calculation taken from Stanford's theory in Gaming Programming website (http://theory.stanford.edu/ amitp/GameProgramming/Heuristics.html) in tie-breaking, as seen in figure.

```
// tiebreaker equation citation:
// http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
public double h_function(int curr_row, int curr_col, int g_row, int g_col){
    double one = Math.abs(curr_row - g_row);
    double two = Math.abs(curr_col - g_col);

    double d_one = curr_row - g_row;
    double d_two = curr_col - g_col;

    double d_row = 0 - g_row;
    double d_col = 0 - g_col;

    //tie breaker
    double cross = Math.abs(d_one*d_col - d_two*d_row);

    double new_h = one+two;

    return new_h += cross*0.001;
}
```

Figure 1: Updated heuristic tie-breaking value for project 2 code, (formula calculation from Stanford's Gaming Programming website, code from Agent 1 and 2's java class file MazeGrid.java).

The project 2's agent 1 and 2 were able to produce better results as compared to those of project 1. Thus our data and graphs for this project collected reached different observations as compared to our previous project. Agent 2 (agent with field of sight) produced lower numbers in trajectory, processed grids, and block hits than Agent 1 (agent without field of sight), but the difference in the numbers were not so great as to change project 1's final conclusion. Thus, for project 1 and 2's conclusion, Agent 1 and Agent 2 are still similar with a slight difference in data between the two agents.

Problem 2

AGENT 1 (BLIND AGENT) WITHOUT SENSING

AGENT 1 OBSERVATION:

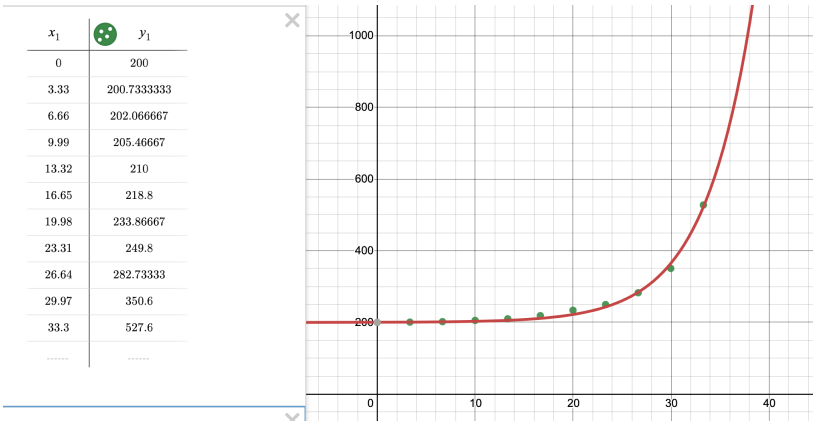
We did 11 densities (0, 3.33, 6.66, 9.99, 13.33, 16.66, 19.99, 23.33, 26.66, 29.99, 33.33) with data from 31 trials for each densities and took the average of these densities to plot the graphs below. The graphs below include Agent 1's trajectory path length, Agent 1's number of processed grids, the runtime of Agent 1, the number of block hits in Agent 1, and the length of the optimal path based on Agent 1's discovered information.

Agent 1 is implemented without the field of sight to detect neighbors on its four sides and without the ability of sensing the possible number of blocks around it (the 8 neighbors surround Agent 1). Because of Agent 1's limitations, it can only detect a block when it hits a block and then re-plan its path. Thus, in the case of updating the maze's block location, Agent 1 can only update one block at a time, much unlike Agent 2 which has the field of sight to record content of its neighbors.

Because at most only one block can be recorded in a path iteration, the number of block hits by Agent 1 would be rather high, for Agent 1 can only move around a block when it learns the blocks location. This would also mean, the number of blocks Agent 1 learns to avoid only increases by one for each iteration, explaining Agent 1's high number of block hits. Of course, this phenomenon can also be explained by the increasing density of block grids in the maze (as shown in figure 4).

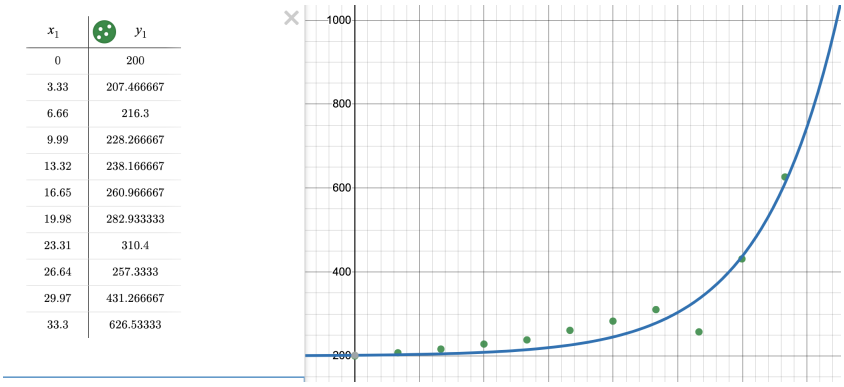
Despite Agent 1's high number of block hits, Agent 1 is not without merits: the runtime. Though Agent 1 has a long trajectory length, high number of grids processed, and high number of block hits, its runtime is quite fast. Thus, at the cost of producing a longer path length and processing more grids and block hits, Agent 1 produced a efficient runtime.

AGENT 1 GRAPH:



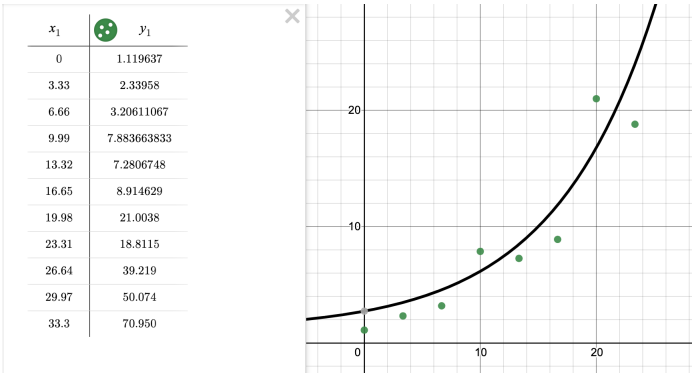
x_1	y_1
0	200
3.33	200.7333333
6.66	202.066667
9.99	205.46667
13.32	210
16.65	218.8
19.98	233.86667
23.31	249.8
26.64	282.73333
29.97	350.6
33.3	527.6

Figure 2: Density vs Agent 1 Trajectory Length (Red: Trajectory path length of Agent1).



x_1	y_1
0	200
3.33	207.466667
6.66	216.3
9.99	228.266667
13.32	238.166667
16.65	260.966667
19.98	282.933333
23.31	310.4
26.64	257.3333
29.97	431.266667
33.3	626.53333

Figure 3: Density vs Agent 1 Processed Grids (Blue: number of grids processed by Agent1).



x_1	y_1
0	1.119637
3.33	2.33958
6.66	3.20611067
9.99	7.883663833
13.32	7.2806748
16.65	8.914629
19.98	21.0038
23.31	18.8115
26.64	39.219
29.97	50.074
33.3	70.950

Figure 4: Density vs Agent 1 Runtime (ms) (Black: runtime for Agent1).

x_1	y_1
0	0
3.33	6.833
6.66	14.633
9.99	23.3
13.32	29.766666
16.65	44.733333
19.98	54.7
23.31	69.73333
26.64	93.3333
29.97	126.8
33.3	206.966667

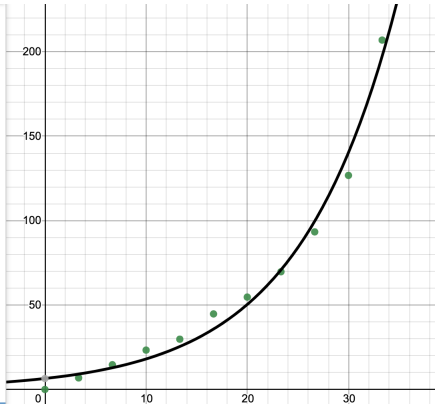


Figure 5: Density vs Agent 1 Blocks Hit (Black: number of blocks hit by Agent1).

x_1	y_1
0	200
3.33	200.066667
6.66	200.53333
9.99	201.53333
13.32	203.066667
16.65	207.6
19.98	214.2
23.31	219.66667
26.64	232.53333
29.97	244.46667
33.3	285.266667

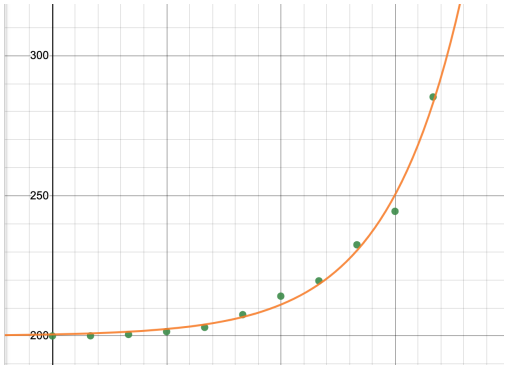


Figure 6: Density vs Agent 1 Optimal Discovered Path (Orange:Optimal Discovered Path for Agent1).

Problem 3

AGENT 2 (FIELD OF SIGHT) WITHOUT SENSING

AGENT 2 OBSERVATION:

We did 11 densities (0, 3.33, 6.66, 9.99, 13.33, 16.66, 19.99, 23.33, 26.66, 29.99, 33.33) with data from 31 trials for each densities and took the average of these densities to plot the graphs below. The graphs below include Agent 2's trajectory path length, Agent 2's number of processed grids, the runtime of Agent 2, the number of block hits in Agent 2, and the length of the optimal path based on Agent 2's discovered information.

Agent 2 is implemented with the field of sight that allows it to detect neighbors on its four sides. And because it does not have the ability of sensing the possible number of blocks around it (the 8 neighbors surround Agent 2), Agent 2 does not implement inference like that of Agent 3. Because of Agent 2's field of sight, it can detect its neighbors of the 4 sides, and update information of the maze. Once Agent 2 hits a block, it will then re-plan its path. Thus, in the case of updating the maze's block location, Agent 2 can update multiple blocks in one path iteration, which significantly more efficient in maze discovery update compared with Agent 1 (as shown in block graph below).

Because more than one block can be updated per path iteration, the number of block hits by Agent 2 would be lower than its counter part Agent 1. This would also mean, the number of

blocks Agent 2 learns to avoid increases by more than 1 for each iteration.

As a consequence of learning block locations at a faster rate, Agent 2 has a lower trajectory length and lower processed grids. But it is also important to note, though Agent 2 does have lower values of the mentioned data, the difference between Agent 1 and Agent 2 is not as large as expected according to the data collected. We believe the difference between the two Agents would clearer if we increase the density of the blocks and/or increase the dimensions of the maze. The same goes to the optimal path for discovered grids by the agent. Agent 2 had a shorter optimal path than Agent 1 due to the field of sight and updated information on the agent's neighbors.

While, in the matter of runtime, Agent 1 and 2 do not have a significant difference. This was expected, since the implementation structure of Agent 1 and Agent 2 is not so different from one another.

AGENT 2 GRAPH:

x_1	y_1
0	200
3.33	200
6.66	200.1290323
9.99	204.0645161
13.33	210.7741935
16.66	215.7419355
19.99	231.2903226
23.33	251.4193548
26.66	271.8387097
29.99	344.0645161
33.33	469.7419355

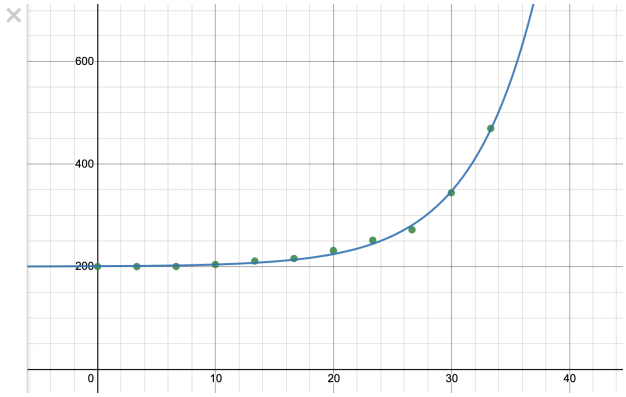


Figure 7: Density vs Agent 2 Trajectory Length (Black: Trajectory path length of Agent 2).

x_1	y_1
0	200
3.33	204.7419355
6.66	215.3225806
9.99	224.4193548
13.32	237.6129032
16.65	255.0645161
19.98	279.1612903
23.31	302.483871
26.64	322.6129032
29.97	403.4193548
33.3	510.3225806

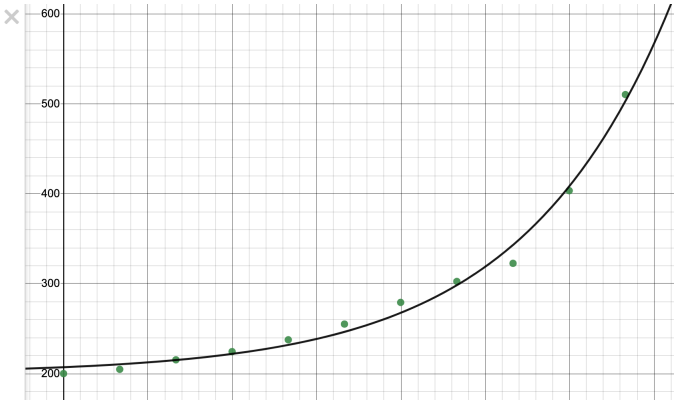


Figure 8: Density vs Agent 2 Processed Grids (Black: number of grids processed by Agent 2).

x_1	y_1
0	2.290322581
3.33	6.451612903
6.66	10.58064516
9.99	14.16129032
13.32	19.22580645
16.65	18.93548387
19.98	19.77419355
23.31	26.87096774
26.64	71.83870968
29.97	97.90322581
33.3	114.4193548

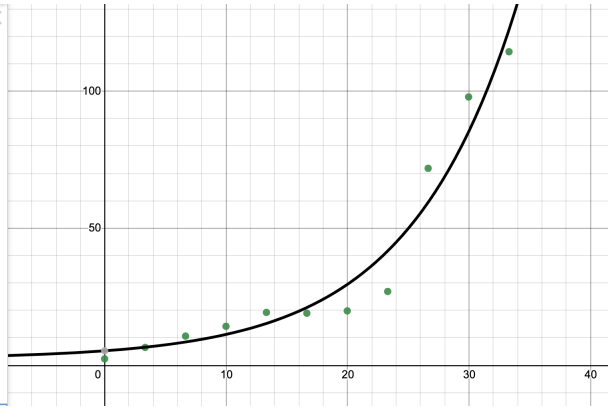


Figure 9: Density vs Agent 2 Runtime (ms) (Black: runtime for Agent 2).

x_1	y_1
0	0
3.33	6.161290323
6.66	14.25806452
9.99	23.70967742
13.32	32.12903226
16.65	41.74193548
19.98	52.74193548
23.31	67.25806452
26.64	81.19354839
29.97	115.2903226
33.3	131.0645161

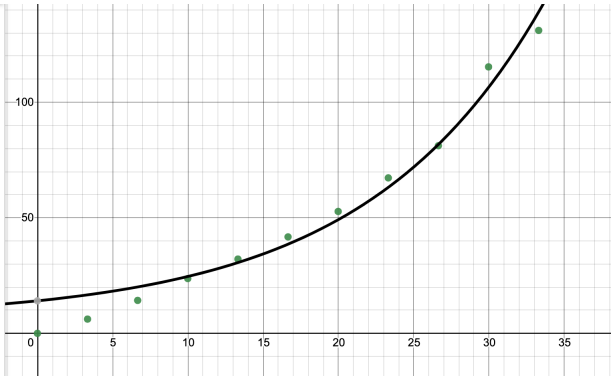


Figure 10: Density vs Agent 2 Blocks Hit (Black: number of blocks hit by Agent 2).

x_1	y_1
0	200
3.33	200
6.66	200.1290323
9.99	201.2903226
13.32	203.9354839
16.65	207.483871
19.98	213.6129032
23.31	220.7419355
26.64	228.3225806
29.97	252.8387097
33.3	268.5806452

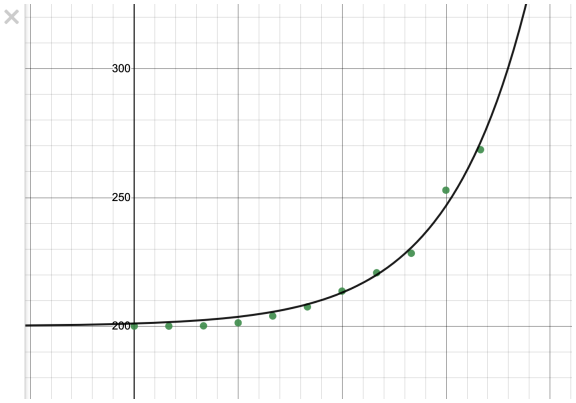


Figure 11: Density vs Agent 2 Optimal Discovered Path (Black:Optimal Discovered Path for Agent 2).

Problem 4

AGENT 3 WITH SENSING

AGENT 3 DESIGN:

When designing the inference agent, the first challenge was how to neatly organize and store information. If we didn't have a really solid system in place for that, we may waste precious time searching for the information as opposed to having direct access to it.

- For that reason, we created an object of type “CellInfo” that would then store all of the accompanying information for the cell in question. This includes whether the cell has been visited or not, whether the cell has been confirmed to be blocked, empty or is still unconfirmed in its status, the total number of neighbors the cell has, the number of cells around it that are sensed to be blocked, the number of neighboring cells confirmed to be blocked, the number of neighboring cells confirmed to be empty, and the number of neighboring cells still unconfirmed. This corresponds to all the information listed within the example inference agent write-up. - Then, we created a “Maze” data structure that was essentially a 2D arraylist of type CellInfo that would store these cells, and their coordinate could be directly accessed with something like maze.get(2).get(3) for the coordinate 2, 3 in the maze where 2 is the column number (corresponding to the X axis) and 3 is the row number (corresponding to the Y axis).

Each cell also stored its own g-value, h-value and subsequently f-value. This meant we wouldn’t have to search anywhere but the cell we’re interested in during the planning phase for determining where to go next. And lastly, we had each cell point to its “parent” or the cell that we used to get to this point in the maze. This was mostly in place so that backtracking, if needed, could be implemented a little easier during the planning phase as well.

AGENT 3 WORKFLOW DESIGN:

So, now we have pretty efficient access to all the information we need, but how do we go about processing it to our advantage? The workflow for this agent essentially followed the example’s suggestion from the project description. We made an initial plan under the freespace assumption, and then for every cell popped off this planned path, we sensed around us, attempted to infer any new information we could from that, and checked to see if there had been a block discovered in our path (if so we had to replan). If not, we would attempt to “move” to the next cell, and if it was blocked, we would infer anything we could given this update in the knowledge base and then replan. If the cell wasn’t blocked, we would actually physically move into the cell and continue the loop again, popping off another cell from the planned path.

To dig in a level of abstraction deeper, specifically with the actual inferring stage, we had to implement things a certain way to get them to work for us. So, say a cell has been sent to the inference method. Here’s the general logical flow for what would happen next:

- We would first “check surroundings”. This is essentially equivalent to making sure our knowledge base is up-to-date. We look around at all the neighboring cells and update our necessary counts. For example, if a cell had recently been confirmed to be open but to our previous knowledge it was unconfirmed, now we will have an increase in the “confirmed empty neighbors” count and a decrease in the “unconfirmed neighbors” count.

- So, now our knowledge base is up-to-date for the cell we are interested in. This is where we apply the inference rules seen in the project description. Specifically, we checked first to see if Hx wasn’t equal to zero because if it was, we know that there’s nothing left to infer for this cell. If Hx was equal to zero, then we checked if Cx = Bx (a.k.a. if the number of blocks sensed around this cell is equal to the number of neighbors confirmed to be blocked), and if we knew that, then we knew that the remaining unconfirmed neighbors were empty. If not that, then if Nx - Cx = Ex, we knew that the remaining unconfirmed neighbors were blocked.

- With this knowledge, we now “check surroundings” of the neighbors to the current cell. This makes sure any changes to the state of the cell we’re currently in (e.g. if we just confirmed it was empty by moving into it) is propagated to the surrounding cells. Also, if one of the

“restEmpty” or “restBlocked” conditions were true, we change the status of all unconfirmed neighbors with respect to the conditions met, and then we add them to a list to be sent into the inferring method recursively. The reason this is done is so that their change in being confirmed can be reflected in its neighbors as well, which might be out-of-reach to the current cell we’re at (at least some of them). - The last thing we do (this is still in the current stackframe, so after these checks but before sending neighboring cells back into the inference method) is if the neighboring cell was either confirmed or the “restEmpty” or “restBlocked” conditions weren’t meant, we actually do the same inference checks as the one done for the current cell on the neighboring cell as well, and if we find anything of interest, we add that to the list to be sent in recursively. This means any of the meat of the inference method (changing statuses of neighbors, etc.) will be done on the next stackframe up.

With this workflow, we make sure that everything that can be inferred about our immediate surroundings at a given point in the maze is propagated outward. Now, we don’t necessarily scan over the entire maze every time we move into a new cell. If there reaches a radius around the given cell that we can no longer infer anything new, we stop our inference process and move on. This saves computationally in potential wasteful ventures through the entire maze, and we describe more later in the report about how this was an intentional design choice to help us solve the maze and infer appropriate things but in a reasonable amount of time (refer to “Computational Issues”).

AGENT 3 OBSERVATION:

We did 11 densities (0, 3.33, 6.66, 9.99, 13.33, 16.66, 19.99, 23.33, 26.66, 29.99, 33.33) with data from 31 trials for each densities and took the average of these densities to plot the graphs below. The graphs below include Agent 1’s trajectory path length, Agent 3’s number of processed grids, the runtime of Agent 3, the number of block hits in Agent 3, and the length of the optimal path based on Agent 3’s discovered information.

Looking at Agent 3’s graph, we can clearly see a significant improve in results when we compare Agent 3’s data to those of Agent 1 and 2.

As stated before in the Workflow section of Agent 3, Agent 3’s implementation is different than that of Agent 1 and 2. With the ability to sense and application of inference, Agent 3 could make better decisions to which path to take than Agent 1 and Agent 2, which make their decisions based on the heuristic value. From project 2, we know the limitation in efficiency of making path decisions mainly based on heuristic values.

With Agent 3’s inference and its sensing ability, we were able to significantly overcome such limitations. But this can at a price of runtime. Looking at the graph, we clearly see Agent 3 has a longer runtime in its run to reach the goal.

Further exploration and observation would be made later about Agent 3’s data.

AGENT 3 GRAPH:

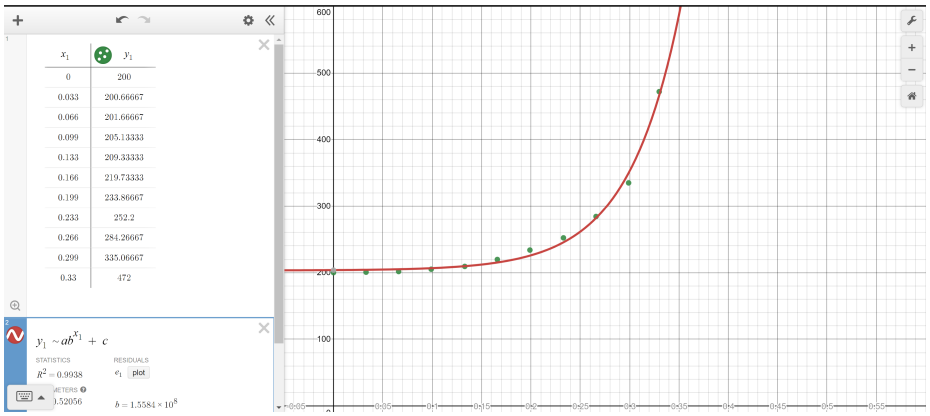


Figure 12: Density vs Agent 3 Trajectory Length(Red: Trajectory path length of Agent 3).

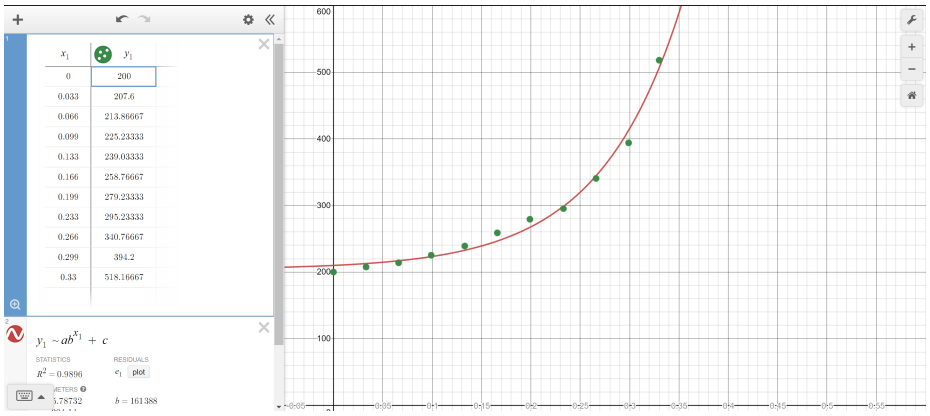


Figure 13: Density vs Agent 3 Processed Grids (Red: number of grids processed by Agent 3).

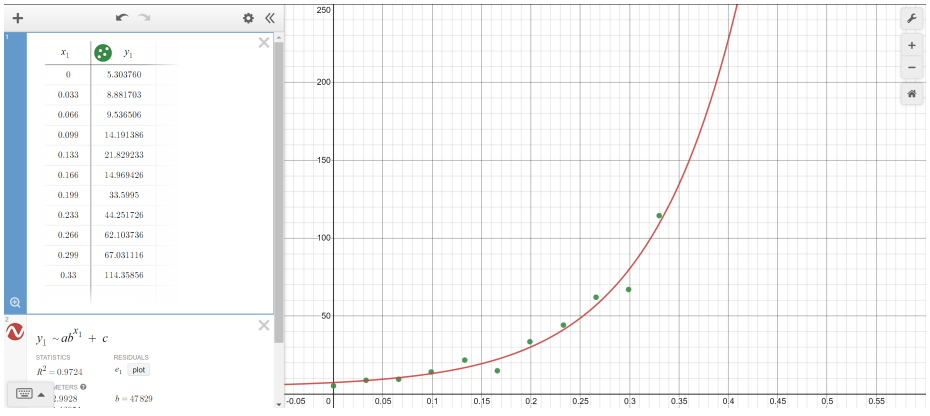


Figure 14: Density vs Agent 3 Runtime (ms) (Red: runtime for Agent 3).

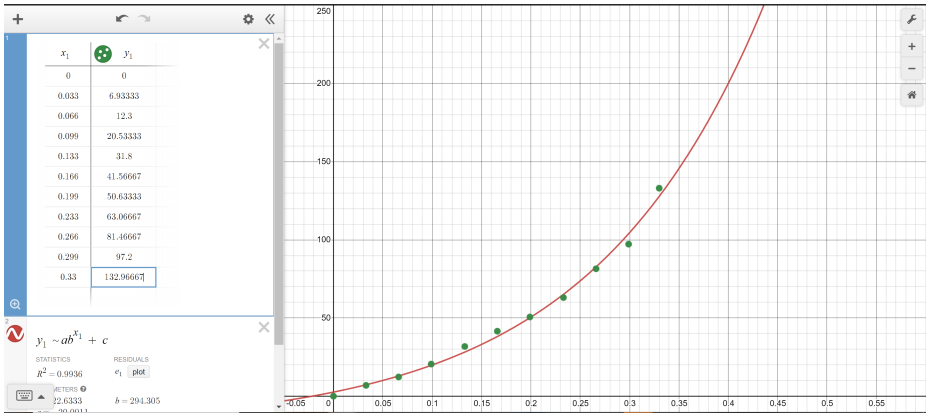


Figure 15: Density vs Agent 3 Blocks Hit(Red: number of blocks hit by Agent 3).

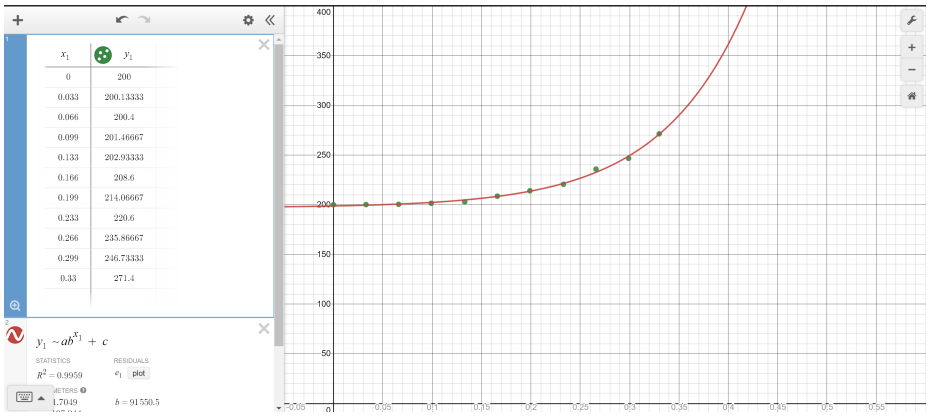


Figure 16: Density vs Agent 3 Optimal Discovered Path (Red:Optimal Discovered Path for Agent 3).

Problem 5

COMPARISON OF AGENTS 1, 2, 3

Q1) You need to compare the performance of Agent 3 to Agent 1 (blind agent) and Agent 2 (field of sight agent). Note that Agent 3 has immediate access to less information than Agent 1, but more information than Agent 2.

How can you accurately compare their performances?/ Things to consider might be: a) total trajectory length (or number of nodes visited), b) final path length through discovered gridworld, c) total planning time.

You should have graphs comparing the performance at different densities, and an explanation of what you see in those graphs.

A) TOTAL TRAJECTORY LENGTH

If we look at the first figure of each agent (1, 2, and 3) we can see that agent 2 has the shortest trajectory length, then agent 3 the second shortest, and agent 1 (blind agent) has the longest trajectory.

Agent 3's inference and sensing ability allowed it to make accurate and more efficient path plan making, updating the knowledge base with possible locations of the block. Ultimately, it obtained the second shortest trajectory path length.

Sight agent had the shortest length. It has the ability to see neighbors, thus updating information at a faster rate than agent 1. And due to its inability to make better decisions, its ability to record neighbors allowed it to get an overall trajectory length that is shorter than Agent 2 and 3 even though it has access to more immediate information than agent 3.

Blind agent had the longest length. This is due to the fact it is updating information at a less efficient rate than sight agent and making less accurate decisions than agent 3, leading to the worst length of the three agents.

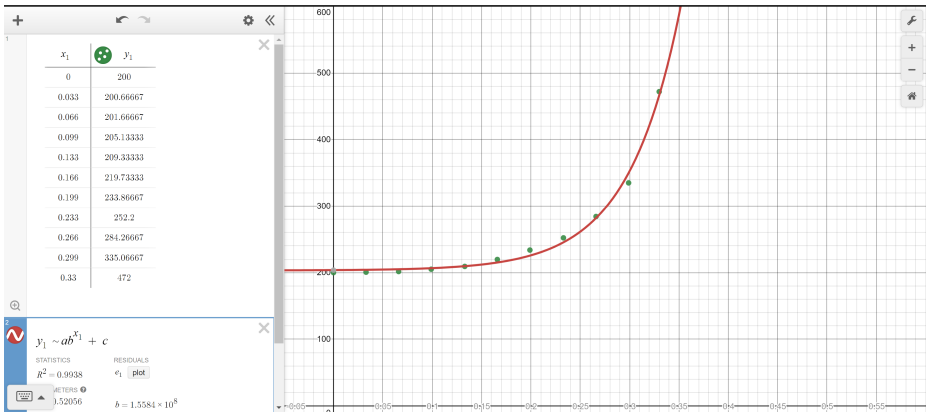


Figure 17: Density vs Agent 3 Trajectory Length(Red: Trajectory path length of Agent 3).

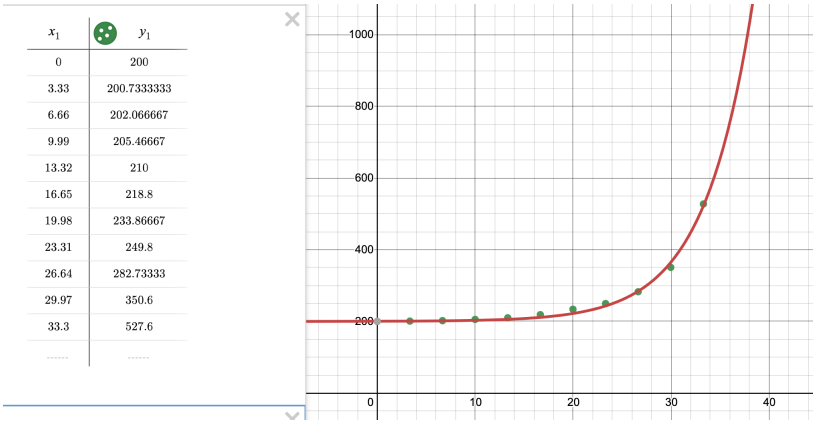


Figure 18: Density vs Agent 1 (Blind) Trajectory Length (Red: Trajectory path length of Agent1).

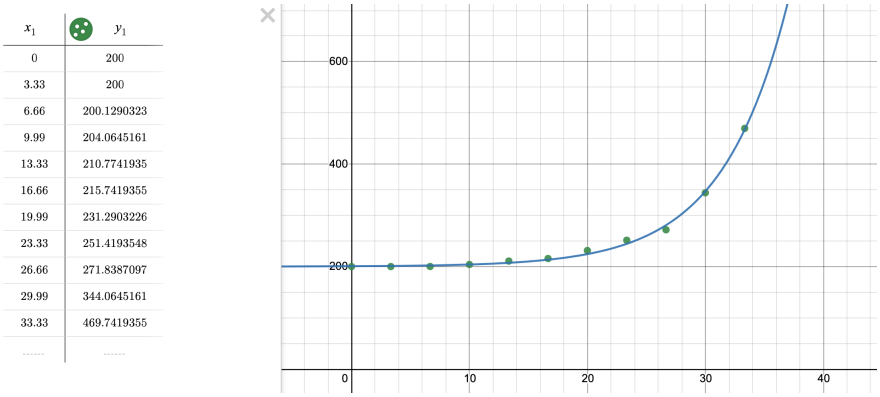


Figure 19: Density vs Agent 2 (Sight) Trajectory Length (Black: Trajectory path length of Agent 2).

B) OPTIMAL PATH LENGTH

If we look at the first figure of each agent (1, 2, and 3) we can see that agent 2 has the shortest discovered optimal path, then agent 3 the second shortest, and agent 1 (blind agent) has the longest path length.

Agent 3's inference and sensing ability allowed it to make accurate and more efficient path plan making, thus it obtained the second shortest optimal path length among the three agents

Sight agent had the shortest length. It has the ability to see neighbors, thus updating information at a faster rate than agent 1. And due its ability to record neighbors allowed it update the maze information faster and more than agent 3, thus allowing its optimal path length to be shorter than Agent 2 and 3.

Blind agent had the longest length. This is due to the fact it is updating information at a less efficient rate than sight agent and making less accurate decisions than agent 3 due to the lack of the ability to sense, leading to the worst length of the three agents.

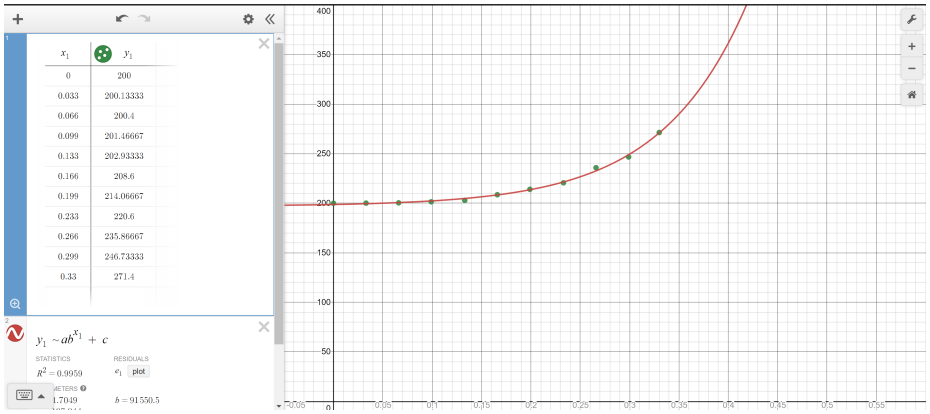


Figure 20: Density vs Agent 3 Optimal Discovered Path (Red:Optimal Discovered Path for Agent 3).

x_1	y_1
0	200
3.33	200.066667
6.66	200.53333
9.99	201.53333
13.32	203.066667
16.65	207.6
19.98	214.2
23.31	219.66667
26.64	232.53333
29.97	244.46667
33.3	285.266667

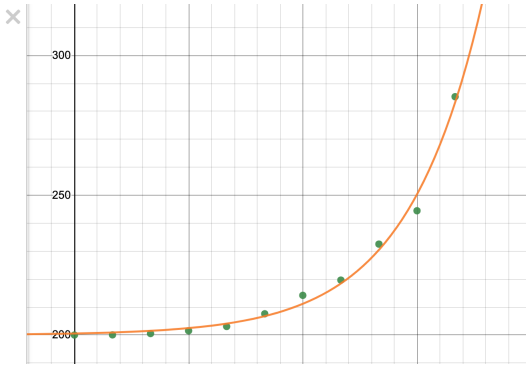


Figure 21: Density vs Agent 1 (Blind) Optimal Discovered Path (Orange:Optimal Discovered Path for Agent1).

x_1	y_1
0	200
3.33	200
6.66	200.1290323
9.99	201.2903226
13.32	203.9354839
16.65	207.483871
19.98	213.6129032
23.31	220.7419355
26.64	228.3225806
29.97	252.8387097
33.3	268.5806452

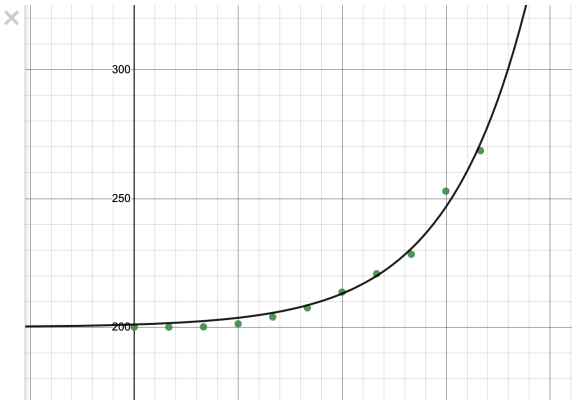


Figure 22: Density vs Agent 2 (Sight) Optimal Discovered Path (Black:Optimal Discovered Path for Agent 2).

C) RUNTIME

If we look at the first figure of each agent (1, 2, and 3) we can see that agent 1 has the shortest runtime, then agent 2 the second shortest, and agent 3 has the longest runtime.

Blind agent had the shortest time because it does not need to record neighbors nor does it need to make inferences based on sensing ability. Thus it only have to focus on collecting

information the block it hits, make a new path plan quickly, then execute. So this leads to the blind agent to get the faster time among the three agents.

The Agent 2 had the second fastest time. It was slower than blind agent because it needs to record and update the agent's 4 neighbors, but because it does not need to make inferences, it is still faster than agent 3.

Agent 3 was the slowest among the three. It does not collect data on the neighbors like agent 2, but agent 3 does use the sense ability and uses rules of inferences in planning its path. Thus this slowed agent 3's planning process down, leading to agent 3 obtaining the slowest time among the three agents.

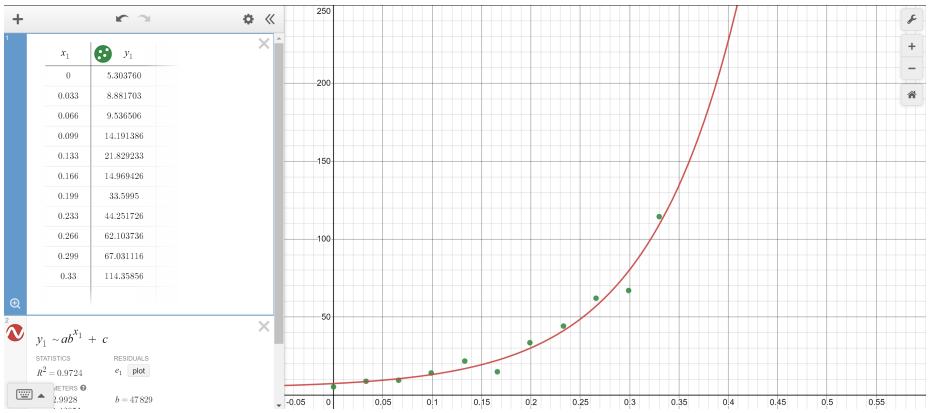


Figure 23: Density vs Agent 3 Runtime (ms) (Red: runtime for Agent 3).

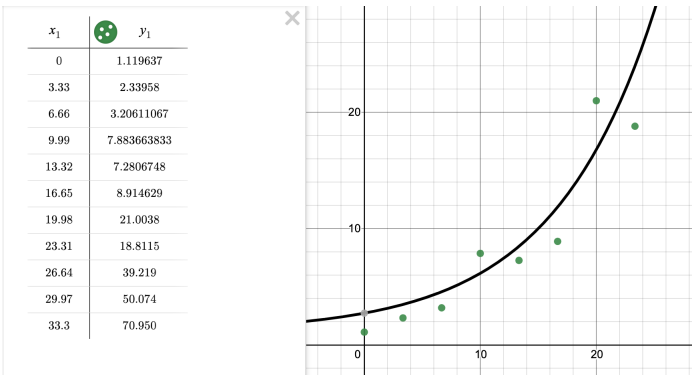


Figure 24: Density vs Agent 1 Runtime (ms) (Black: runtime for Agent1).

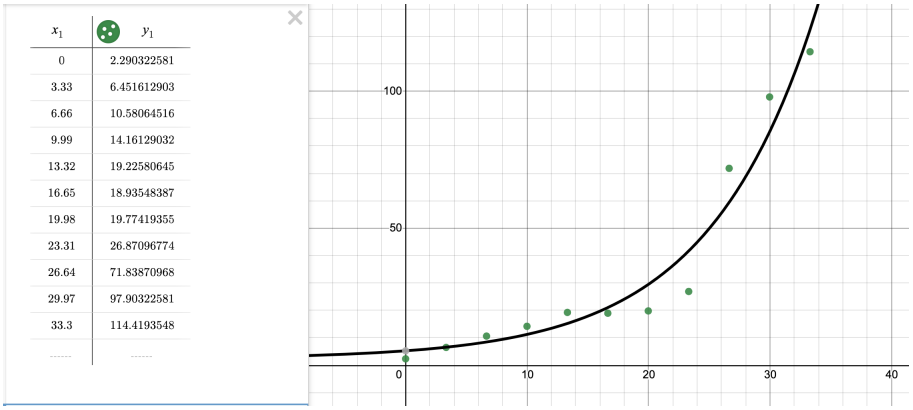


Figure 25: Density vs Agent 2 Runtime (ms) (Black: runtime for Agent 2).

D) BLOCK HIT

If we look at the first figure of each agent (1, 2, and 3) we can see that agent 2 has the least block collisions, then agent 3 the second least, and agent 1 (blind agent) has the most collisions.

Although it cannot immediately collect data about it, Agent 3 used inference and sensing ability to help it to make accurate and more efficient path planning, helping it to avoid block collisions, thus it obtained the second shortest least block collisions among the three agents.

Sight agent had the least collision. It has the ability to see neighbors, thus updating information at a faster rate than agent 1. And due its ability to record neighbors allowed it update the maze information faster and more than agent 3 and 2, thus allowing it to avoid walls more efficiently than Agent 2 and 3.

Blind agent had the most wall collisions. This is due to the fact it is updating information at a less efficient rate than sight agent (for agent 1 is unable to collect information about its neighbors) and making less accurate decisions than agent 3 due to the lack of the ability to sense and inference, leading blind agent to have the most block collisions among the three agents.

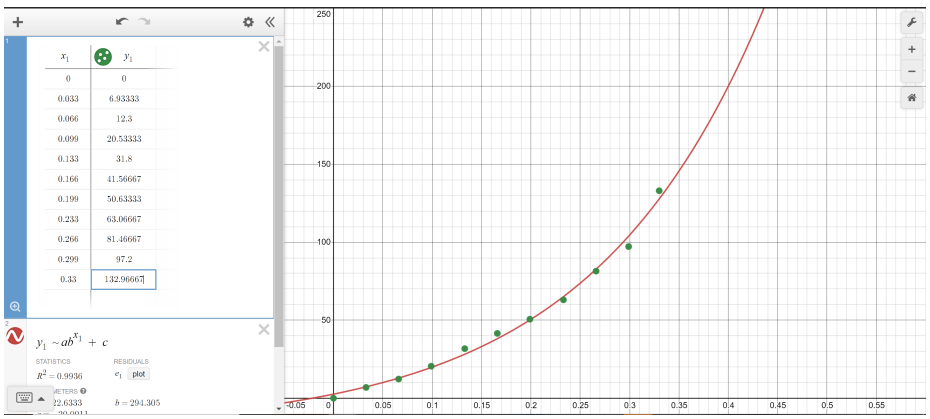


Figure 26: Density vs Agent 3 Blocks Hit(Red: number of blocks hit by Agent 3).

x_1	y_1
0	0
3.33	6.161290323
6.66	14.25806452
9.99	23.70967742
13.32	32.12903226
16.65	41.74193548
19.98	52.74193548
23.31	67.25806452
26.64	81.19354839
29.97	115.2903226
33.3	131.0645161

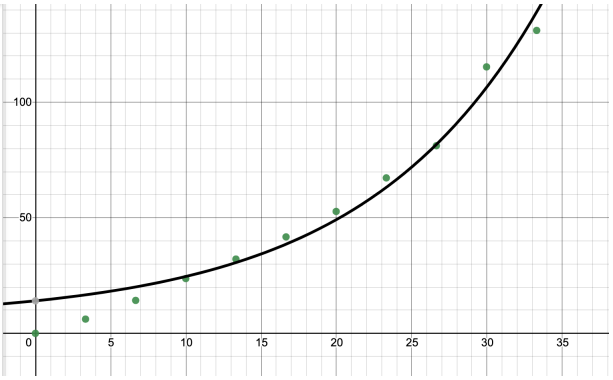


Figure 27: Density vs Agent 2 Blocks Hit (Black: number of blocks hit by Agent 2).

x_1	y_1
0	0
3.33	6.833
6.66	14.633
9.99	23.3
13.32	29.766666
16.65	44.733333
19.98	54.7
23.31	69.73333
26.64	93.3333
29.97	126.8
33.3	206.966667

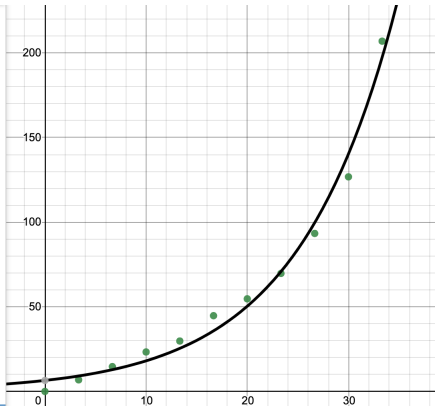


Figure 28: Density vs Agent 1 Blocks Hit (Black: number of blocks hit by Agent1).

Problem 6

AGENT 4 WITH SENSING (IMPROVED INFERENCE)

DESIGNING AGENT 4: What rules / inferences / other methods does Agent 4 perform? How does Agent 4 differ from Agent 3? Can you show that there are situations where Agent 4 can infer things that Agent 3 can't, even if they both have the same knowledge base? Does Agent 3 ever get anything that Agent 4 does not? Does Agent 4 infer "everything that is possible to infer" and if not, why not? Can you construct situations where inference is possible, but Agent 4 doesn't infer anything?

ANSWER:

The basic principle upon which the design of Agent 4 is based is the Minesweeper example that was done during lecture in which we treated unconfirmed cells like variables and the mines sensed to be neighboring that cell as the RHS of the equation. Only here, they're not mines, they're blocks. So, where Agent 3 can look at surrounding neighbors and infer (possibly propagate outward from there), Agent 4 attempts to synthesize information and observations across multiple cells and build new "equations" that may lead to something new that can be inferred.

There were five main inference rules that we decided were deterministic about the status of 1 or more cells with an equation.

First of all, if an equation has only one term, then the block status of the cell is easily found.

For example, if you had -2A = -2, you know for sure that A = 1, and that would mean A is blocked.

All remaining inference rules make use of counting up the sum of the factors (i.e. -2A + 2B + 1C would have a sum of factors equal to 1 because -2 + 2 + 1 = 1), the number of positive factors (in the prior example, that would be 2), and the sum of the positive factors (which would've been 3 in the prior example.

- The second inference rule we used was very niche but still quite effective. If the sum of the factors = 0 and there are 2 factors overall and the total number of blocks is equal to the absolute value of both terms' factors, then we know that one is blocked and one is empty. For example, if we had the equation 1A + -1B = 1, then we can see that 1A = 1 + 1B. Since A can only be zero or one, we can conclude that A must be 1 (blocked) and B must be 0 (empty).

- The third inference rule we used was if the sum of the positive factors was equal to the total number of blocks, then we can conclude that every term with a positive factor must be blocked and every term with a negative factor must be empty. This follows from the following proposition while considering the equation 1A + 1B + -1C = 2: say that any one of these terms with a positive factor were actually empty and evaluated to zero. The equation is no longer satisfiable. We cannot possibly reach the total number of blocks without this term being equal to the factor * 1. On the other hand, consider if a cell with a negative factor was actually blocked. Again, this is no longer satisfiable because we can no longer reach the magnitude of the RHS of the equation.

- The fourth inference rule is essentially the same idea but switching roles. Take -1A + -1B + 1C = -2. It follows that every cell with a negative factor must be blocked and every cell with a positive factor must be empty (refer to above).

- The fifth and final inference rule we decided to employ was this idea: if all terms were either positive or negative and the RHS (total number of blocks between them) was equal to 0, then we know that every cell in this equation must be empty. Imagine 1A + 2B + 3C + 1D = 0. If any one of these terms evaluated to anything but zero, there would be no way to reduce the RHS back down to zero. Thus, every term must be zero and every cell must be empty.

Let's see this in practice. Looking at Agent 4 Example in Practice...

The check marks indicate the cells that have been confirmed by the agent, and the numbers in the middle of the cells explored so far indicate the number of blocks sensed around that cell. We've assigned variables to unconfirmed cells for simplicity's sake, and let's say we try to derive a new equation from cells 1,1, and 1,0. On their own, they each know they have a block somewhere around them, but they don't necessarily know where it is. However, if we compile their information together, then we find that the agent can determine at least some information about the neighboring cells. Mainly, we determine that A, B and C are all empty thanks to the fifth rule described above.

So, you can see that Agent 4 just builds on Agent 3, taking in the same information and also having a similar inference phase. Agent 4 just extends that idea of inference into a database of extra facts as well. You can actually think of Agent 4 as just an extension of Agent 3.

The biggest challenge in designing this particular approach is having a computer decide which equations were best to pair together and match up to reduce the amount of terms we are looking at. The simpler the equation, the less variables there are, the better. But often as humans, we're

making judgements that are very complex and not necessarily intuitive algorithmically. With that in mind, if you don't synthesize / reduce your equations in just the right way, the agent may not be inferring every possible thing that it could possibly infer for the maze at a particular state. But how much time would it take for the agent to work out exactly what equations to match up? And would it be worth it? That's a fair question. So, when implementing this was something we had to take into account.

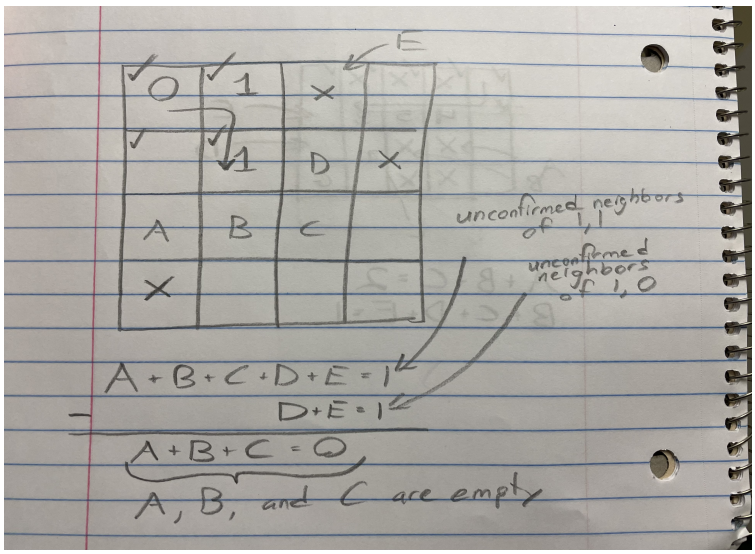


Figure 29: Agent 4 Example in Practice.

IMPLEMENTING AGENT 4:

Be clear about a) how you are representing your information, and b) how you are manipulating that information to perform your inferences.

ANSWER:

INFORMATION REPRESENTATION

For this implementation, this is essentially using Agent 3 as a framework and building on from that. The biggest addition (refer to the appendix) is the Association class. This is how we devised to store these so-called "equations" in our program. Each Association instance had an integer called totalBlocks which acted as the RHS of an equation (also known as the total blocks sensed between all terms) and an array of type Association.Unit that stored the terms of the LHS. The Unit inner class defined an object type that stored a cell (CellInfo type) and an integer that simulates the factor of a given term (i.e. with 2A, 2 is the factor).

As opposed to Agent 3, we've now added an Association field to each cell that stores all of its unconfirmed neighbors. This is set up when the maze is built and then each subsequent "discovery" allows us to update the necessary neighbors of this change in status. More specifically, if a cell is stored in an Association instance, its status has not yet been confirmed to us, either by inference or observation. Whenever a cell's status is confirmed, we update our knowledge base immediately so that we're not trying to infer things that have already been determined (and also possibly shielding us from more information).

Our Agent 4 has a field that is essentially a database of extra facts that we've been able to determine from the cells in the maze. This is an array of Association objects, and it is also updated upon each subsequent discovery. The agent looks to this database at certain times throughout the traversal of the maze (described in Workflow below).

WORKFLOW

The workflow is almost identical to Agent 3 except for a few key distinctions that come about during the inference phase that we've improved upon / extended. Once we're done inferring everything we can about the cell that the agent is currently in and its neighbors (Agent 3 implementation), we look to the next cell in our path. Treating this almost like the expert systems idea from class, we try to narrow our focus in this extension of the inference phase so we don't get stuck looking at the entire maze, potentially wasting time (refer to "Computational Issues" below). It directs our attention and computational power to something that truly matters for us, the next step in our path.

The first thing we do is we try to synthesize a new Association based on the information provided by the current cell and its parent cell (the next cell back in the path). The reason we are doing this is because Association objects require a "number of neighbors sensed to be blocked" value to be a valid equation, so we can't look to any neighbors we haven't already visited, and the safest bet is the cell we just came from. The other added bonus of synthesizing the information from these two cells in particular is they will have shared neighbors, so they're bound to have some terms cancel out. The hope is that they provide concrete information as a result of this, but that's an issue for the processing part (paragraph below). The last note I'll make is that if the equation actually has all terms cancel, then it's not very helpful to us. This situation would occur if the only unconfirmed neighbors were the ones that the two cells shared.

After this synthesis is done and there has been a new Association (potentially) added to the knowledge base, we feed this next cell into the database of extra facts we've compiled over time, we check for any and all equations that may contain this cell, and we attempt to process these facts. This is essentially inference part 2. This is where we take all of those inference rules from the design plan above and we check for them here. If one of them gets a hit and we can confirm some of the cells in the equation, we send those back to the Agent and that tells it to infer (part 1 / Agent 3 stuff) on those cells again, just in case we learn even more. And of course, our knowledge base is updated as soon as possible. A small change from a method that was in Agent 3 was that checkSurroundings() also updates the Association object attached to the cell and gets rid of any cells that have since been confirmed.

AGENT 4 GRAPH: We did 11 densities (0, 3.33, 6.66, 9.99, 13.33, 16.66, 19.99, 23.33, 26.66, 29.99, 33.33) with data from 31 trials for each densities and took the average of these densities to plot the graphs below. The graphs below include Agent 1's trajectory path length, Agent 4's number of processed grids, the runtime of Agent 4, the number of block hits in Agent 4, and the length of the optimal path based on Agent 4's discovered information.

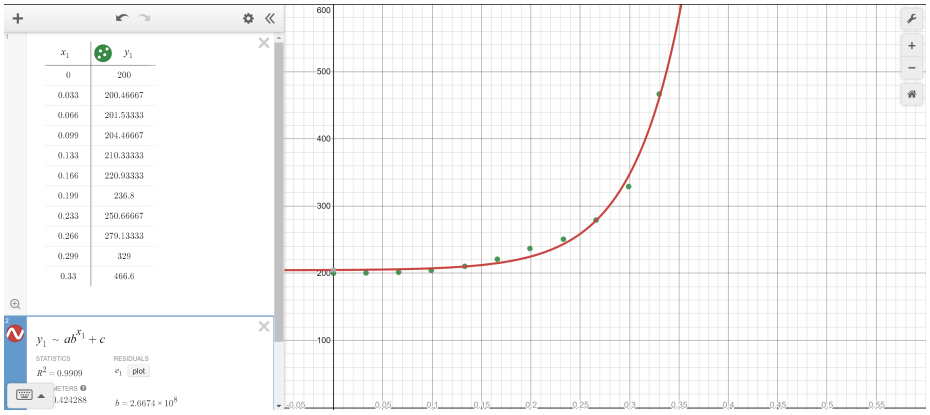


Figure 30: Density vs Agent 4 Trajectory Length(Red: Trajectory path length of Agent 4).

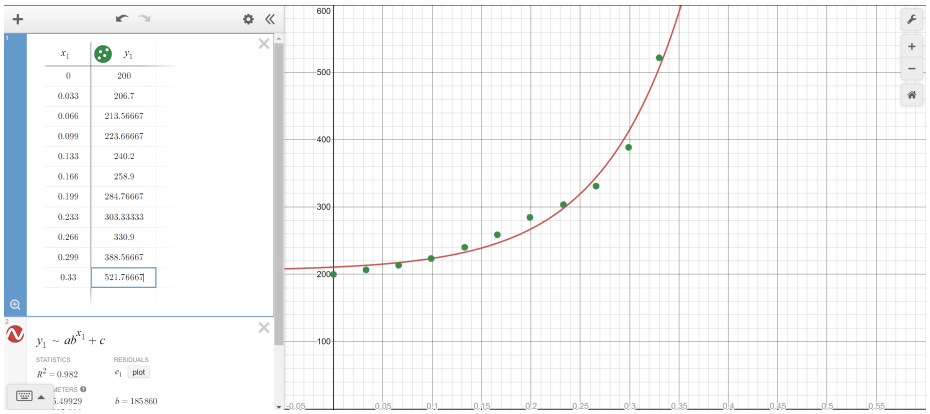


Figure 31: Density vs Agent 4 Processed Grids (Red: number of grids processed by Agent 4).

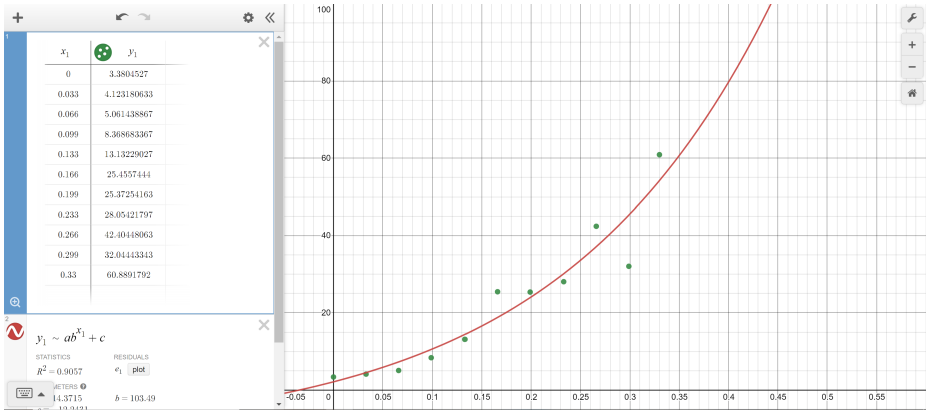


Figure 32: Density vs Agent 4 Runtime (ms) (Red: runtime for Agent 4).

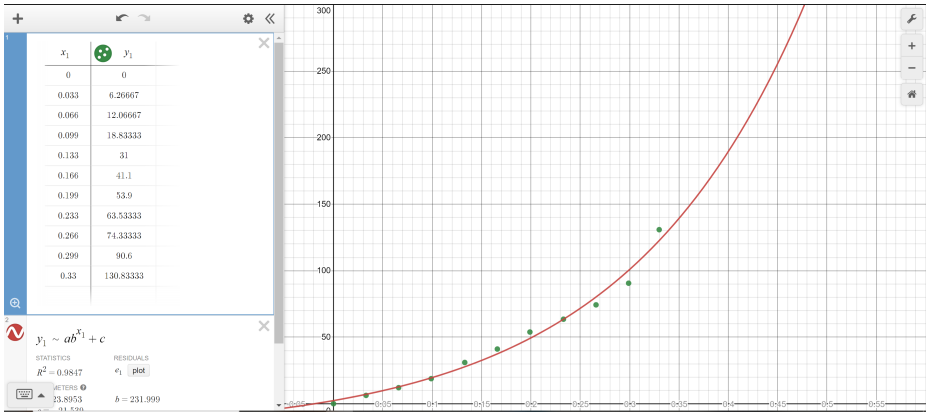


Figure 33: Density vs Agent 4 Blocks Hit(Red: number of blocks hit by Agent 4).

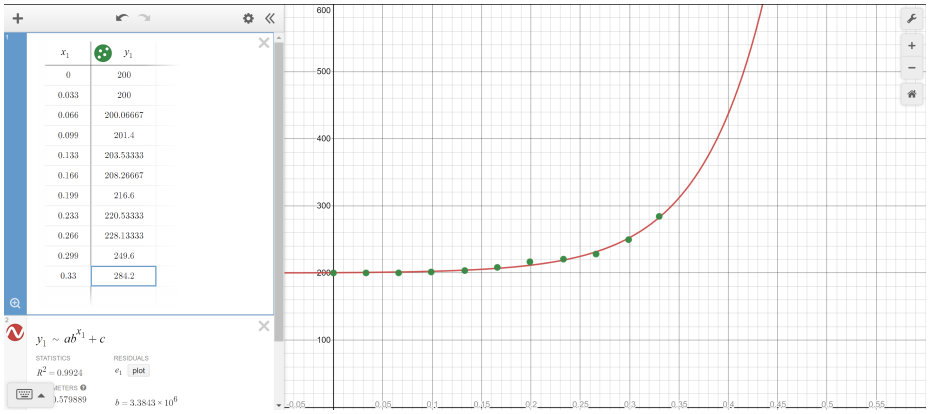


Figure 34: Density vs Agent 4 Optimal Discovered Path (Red:Optimal Discovered Path for Agent 4).

Problem 7

COMPARISON AGENT 3 and 4:

Does Agent 4 actually perform better than Agent 3? For instance, even though Agent 4 may be able to infer more things - do those situations ever actually arise in practice?

ANSWER:

Surprisingly, the data we collected from Agent 4 are not that different from those of Agent 3. If we look at the figures below, despite adding new rules of inferences for agent 4, the two agents' performance were very similar. In fact, the data were so close that it is difficult to make a sound answer of which agent is better performance-wise than the other.

We have observed that sometimes the situations set in agent 4 do come up in practice, and these inference do work, but it is rather limited effect in the grand scope of things. Thus, even though agent 4 was given extra inferences to infer more situations, it does not actually have a tangible effect on the metrics like trajectory, collisions, or discovered optimal path.

Even though we had added inferences to Agent 4, the blocks collision was still similar to that of Agent 3. Thus showing that the extra inferences do not have a observable effect on improving data for our inference agent.

Although there was another unexpected observation was made from the data, Agent 4 is actually faster than Agent 3. We believe that a possible cause of this phenomenon is that agent 4 can process information faster than agent 3. Although it is also possible different machines that run the program and the agents would yield results at a broad range of runtime.

This is shown in the graphs and data below.

PROBLEM 7 - AGENT 3:

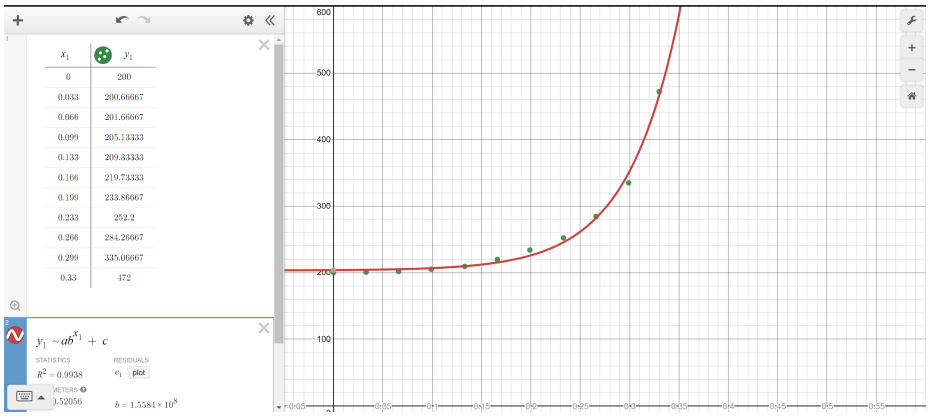


Figure 35: Density vs Agent 3 Trajectory Length(Red: Trajectory path length of Agent 3).

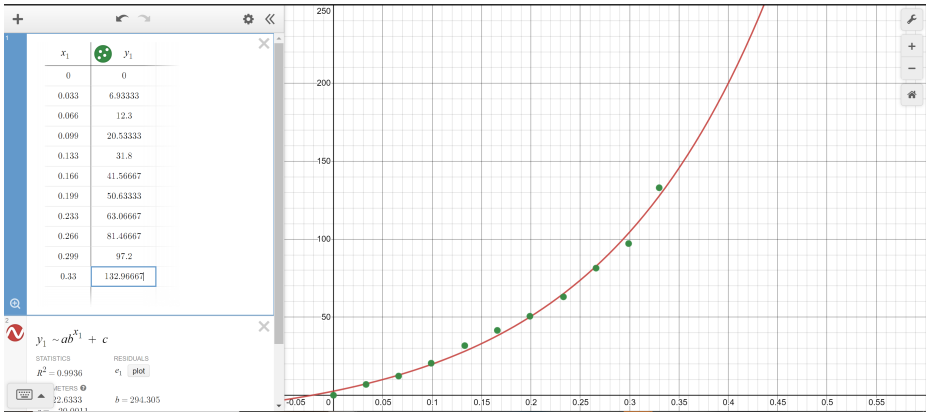


Figure 36: Density vs Agent 3 Blocks Hit(Red: number of blocks hit by Agent 3).

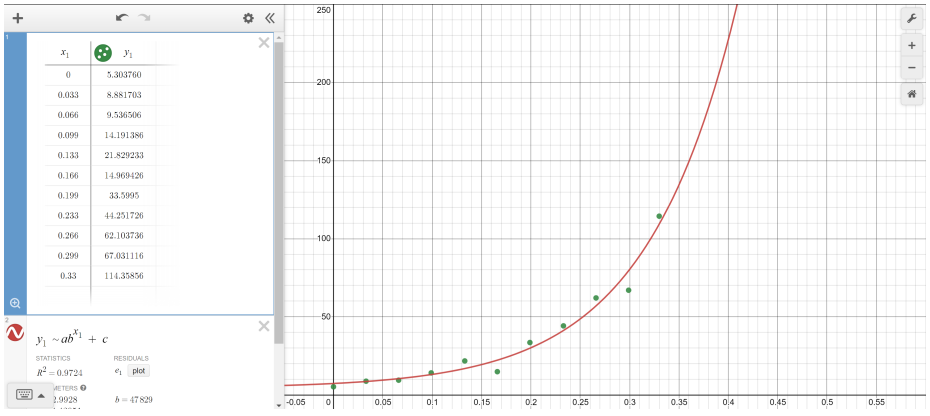


Figure 37: Density vs Agent 3 Runtime (ms) (Red: runtime for Agent 3).

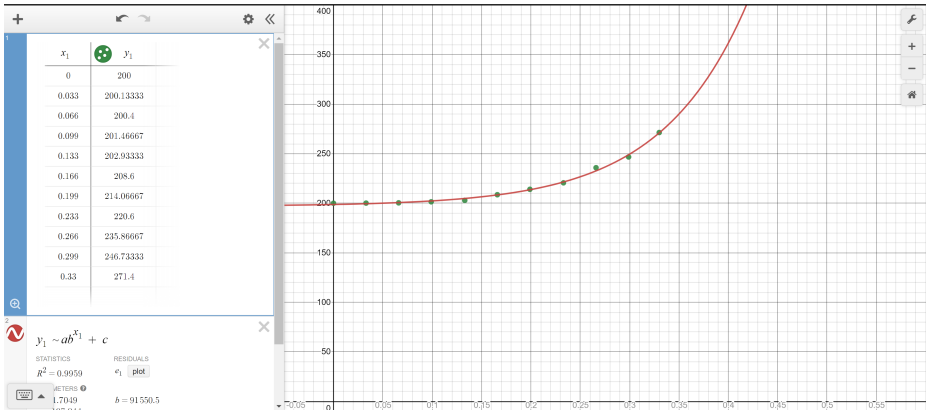


Figure 38: Density vs Agent 3 Optimal Discovered Path (Red:Optimal Discovered Path for Agent 3).

PROBLEM 7 - AGENT 4

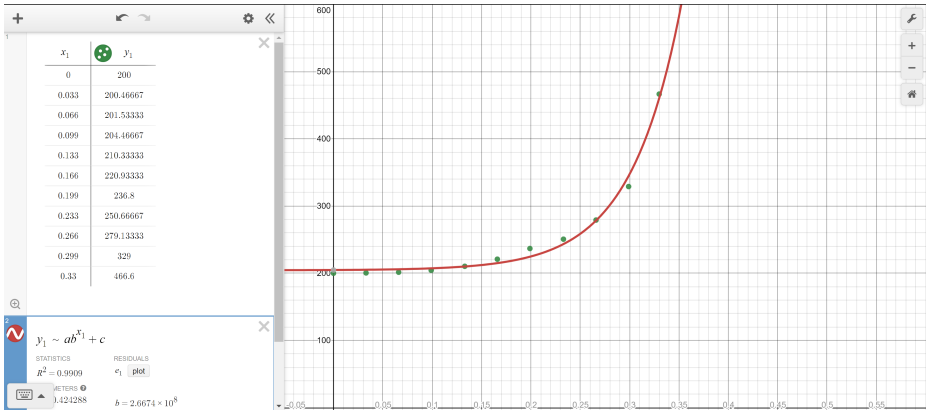


Figure 39: Density vs Agent 4 Trajectory Length(Red: Trajectory path length of Agent 4).

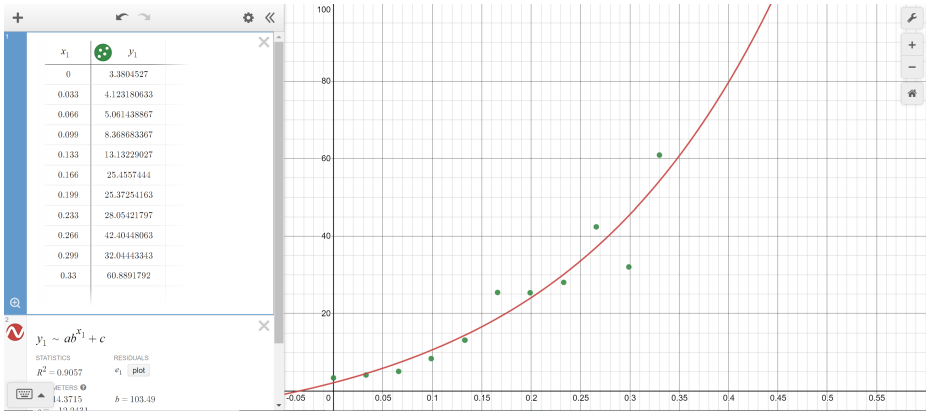


Figure 40: Density vs Agent 4 Runtime (ms) (Red: runtime for Agent 4).

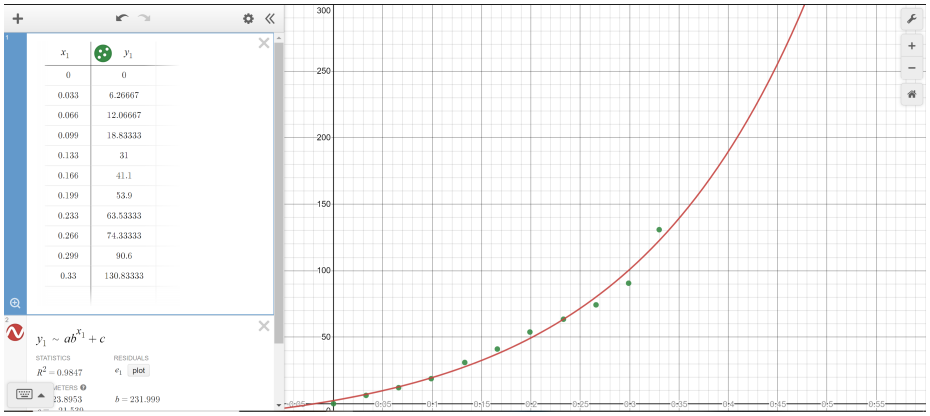


Figure 41: Density vs Agent 4 Blocks Hit(Red: number of blocks hit by Agent 4).

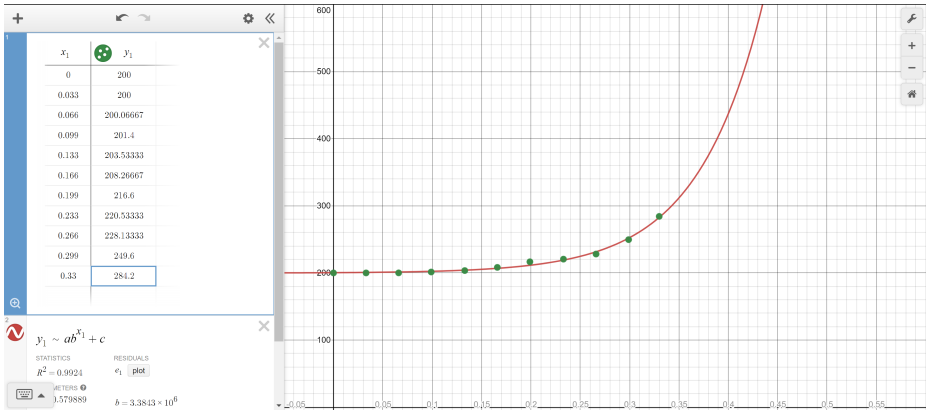


Figure 42: Density vs Agent 4 Optimal Discovered Path (Red:Optimal Discovered Path for Agent 4).

Problem 8

Q: How does Agent 4 compare to Agent 1 and 2? Again, graphs and analysis.

ANSWER:

TRAJECTORY LENGTH COMPARISON:

If we look at the first figure of each agent (1, 2, and 4) we can see that agent 2 has the shortest trajectory length, then agent 4 the second shortest, and agent 1 (blind agent) has the longest trajectory.

Agent 4's inference and sensing ability allowed it to make accurate and more efficient path plan making, thus it was about to obtain the second shortest trajectory path length, which is very similar to agent 3.

Sight agent had the shortest length. It has the ability to see neighbors, thus updating information at a faster rate than agent 1. Despite due to its inability to make better path plans, its ability to record neighbors allowed it to get an overall trajectory length that is shorter than Agent 2 and 4, this is because it has access to more immediate information than agent 4.

Blind agent had the longest length. This is due to the fact it is updating information at a less efficient rate than sight agent and making less accurate plans than agent 4, leading to the worst

length of the three agents.

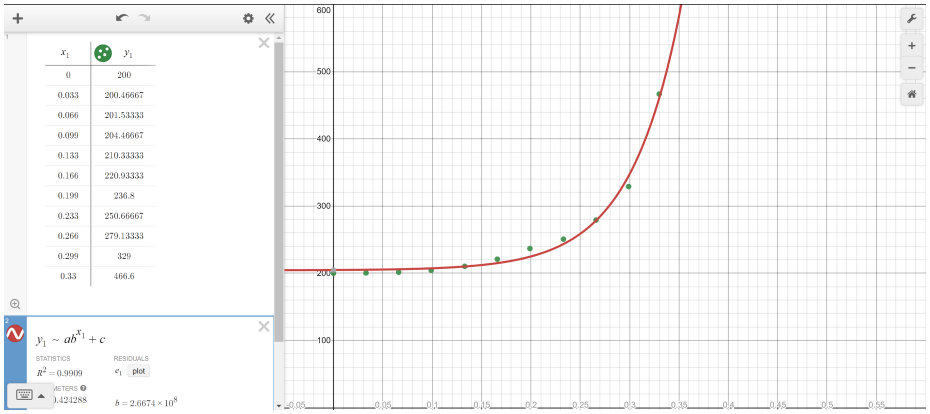


Figure 43: Density vs Agent 4 Trajectory Length(Red: Trajectory path length of Agent 4).

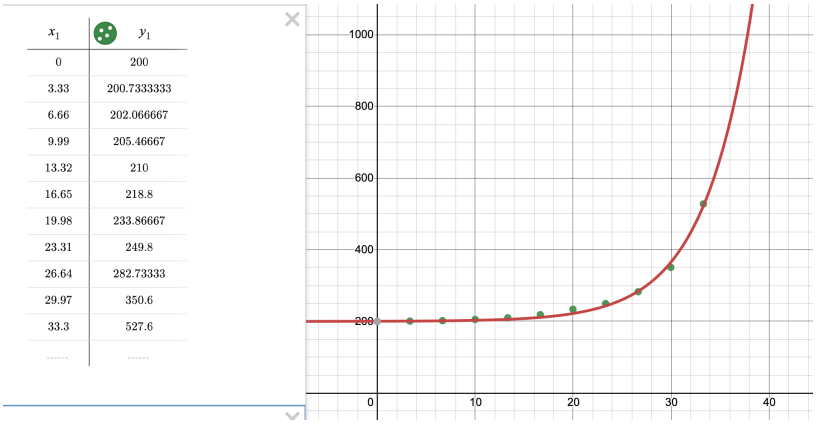


Figure 44: Density vs Agent 1 (Blind) Trajectory Length (Red: Trajectory path length of Agent1).

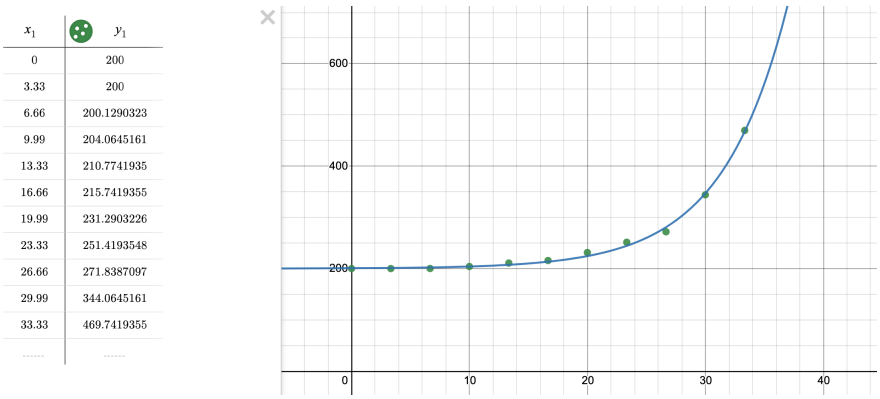


Figure 45: Density vs Agent 2 (Sight) Trajectory Length (Black: Trajectory path length of Agent 2).

OPTIMAL LENGTH COMPARISON:

If we look at the first figure of each agent (1, 2, and 4) we can see that agent 2 has the shortest discovered optimal path, then agent 4 the second shortest, and agent 1 (blind agent) has the longest path length.

Agent 4's inference and sensing ability allowed it to make accurate and more efficient path decision making better than agent 1, thus it obtained the second shortest optimal path length among the three agents.

Sight agent had the shortest length. It has the ability to see neighbors, thus updating information at a faster rate than agent 1. And due its ability to record neighbors allowed it update the maze information faster and more than agent 4, thus allowing its optimal path length to be shorter than Agent 2 and 4.

Blind agent had the longest length. This is due to the fact it is updating information at a less efficient rate than sight agent and making less accurate plans than agent 4 due to the lack of the ability to sense and inference, leading to the worst length of the three agents.

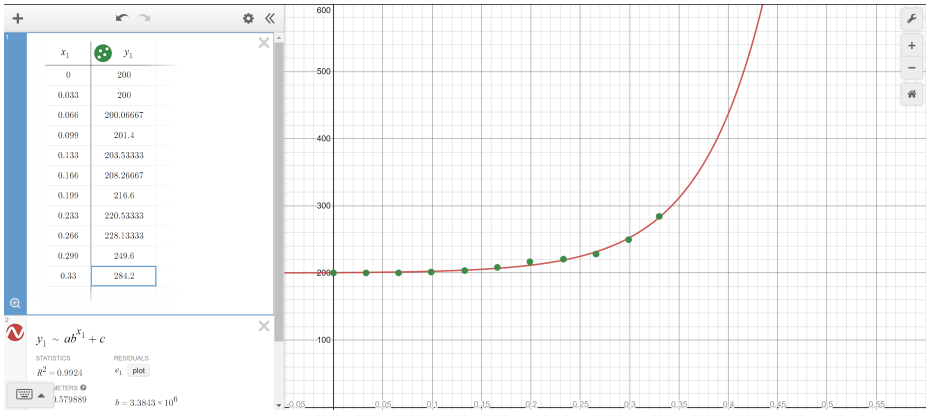


Figure 46: Density vs Agent 4 Optimal Discovered Path (Red:Optimal Discovered Path for Agent 4).

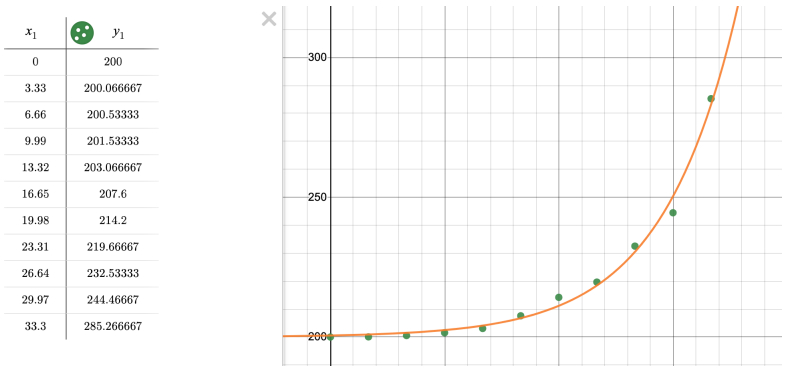


Figure 47: Density vs Agent 1 (Blind) Optimal Discovered Path (Orange:Optimal Discovered Path for Agent1).

x_1	y_1
0	200
3.33	200
6.66	200.1290323
9.99	201.2903226
13.32	203.9354839
16.65	207.483871
19.98	213.6129032
23.31	220.7419355
26.64	228.3225806
29.97	252.8387097
33.3	268.5806452

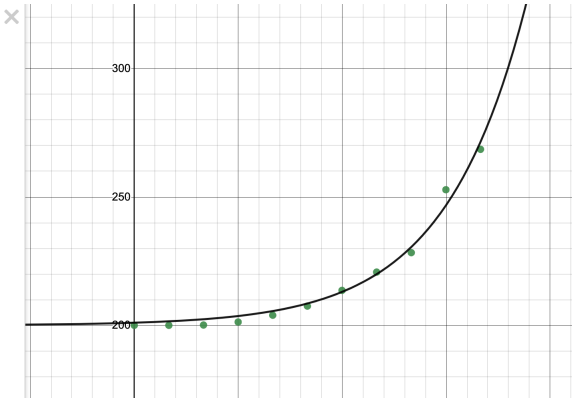


Figure 48: Density vs Agent 2 (Sight) Optimal Discovered Path (Black:Optimal Discovered Path for Agent 2).

RUNTIME COMPARISON:

If we look at the first figure of each agent (1, 2, and 4) we can see that agent 1 has the second shortest runtime, then agent 2 the longest, and agent 4 has the shortest runtime.

Blind agent had the second shortest time because it does not need to record neighbors nor does it need to make inferences based on sensing ability. Thus it only have to focus on collecting information the block it hits, make a new path plan quickly, then execute. So this leads to the blind agent to get the faster time among the three agents.

The Agent 2 had the slowest time. It was slower than blind agent because it needs to record and update the agent's 4 neighbors, and despite it does not need to make inferences, it was actually slower than agent 4.

Agent 4 was the fastest among the three. It does not collect data on the neighbors like agent 2, but agent 4 does use the sense ability and uses rules of inferences in planning its path. This was suppose to slow down agent 4's planning process down and lead to agent 4 obtaining the slowest time among the three agents. But the data we obtained showed that Agent 4 was actually the fastest.

We were able to explain why Agent 4 was faster than Agent 3 as a cause of faster processing of inference than that of Agent 3. But in the case of being faster than agent 1 and 2, the reasoning would most likely be caused by the difference in the performance of the machine.

One of the machine that ran the program for agent 4 produced faster results than the other machines, thus explaining why Agent 4 is faster than the other 3 agents, despite the fact more inferences were suppose to slow down the agent not make it faster.

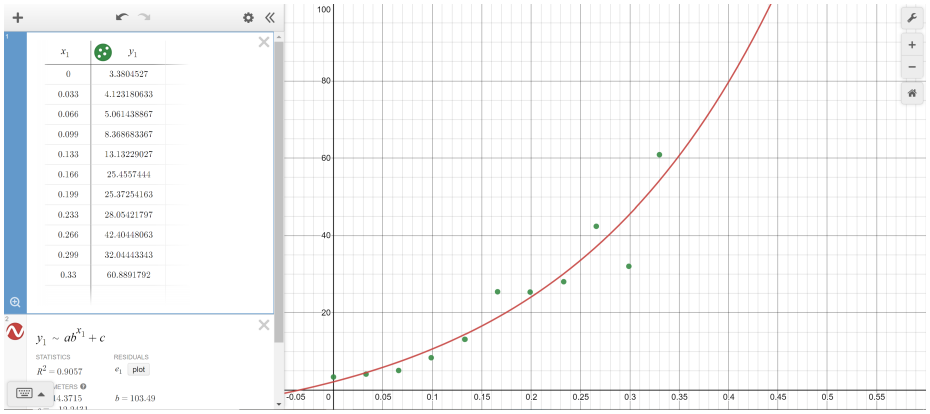


Figure 49: Density vs Agent 4 Runtime (ms) (Red: runtime for Agent 4).

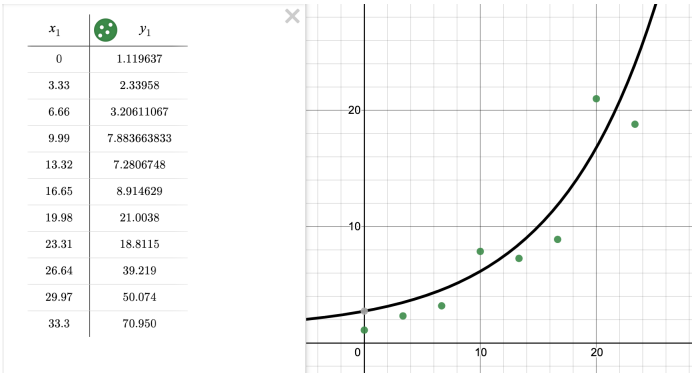


Figure 50: Density vs Agent 1 Runtime (ms) (Black: runtime for Agent1).

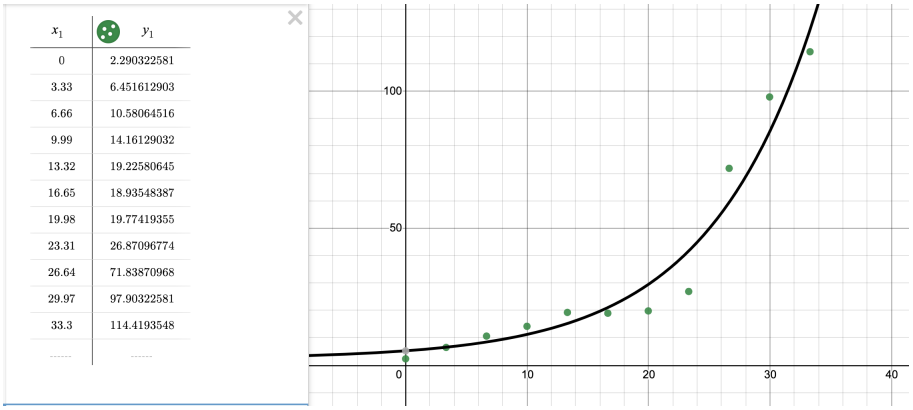


Figure 51: Density vs Agent 2 Runtime (ms) (Black: runtime for Agent 2).

BLOCK COLLISION:

If we look at the first figure of each agent (1, 2, and 4) we can see that agent 2 has the least block collisions, then agent 4 the second least, and agent 1 (blind agent) has the most collisions.

Although it cannot immediately collect data about it, Agent 4 used inference and sensing ability to help it to make accurate and more efficient path planning, helping it to avoid block collisions, thus it obtained the second shortest least block collisions among the three agents.

Sight agent had the least collision. It has the ability to sense neighbors, thus updating informa-

tion at a faster rate than agent 1. And due its ability to record neighbors allowed it update the maze information faster and more than agent 4 and 2, thus allowing it to avoid walls more efficiently than Agent 2 and 4.

Blind agent had the most wall collisions. This is due to the fact it is updating information at a less efficient rate than sight agent (for agent 1 is unable to collect information about its neighbors) and making less accurate decisions than agent 4 due to the lack of the ability to sense and inference, leading blind agent to have the most block collisions among the three agents.

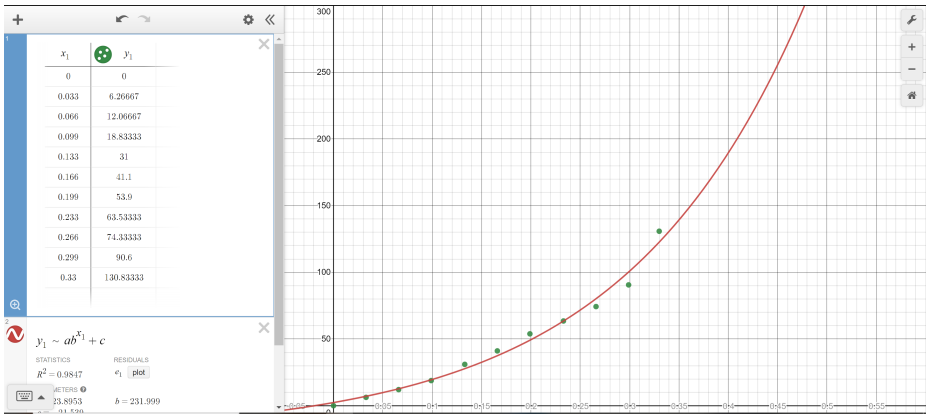


Figure 52: Density vs Agent 4 Blocks Hit(Red: number of blocks hit by Agent 4).

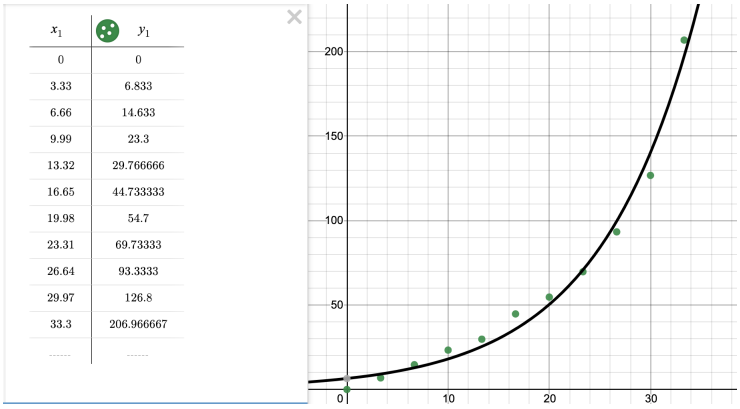


Figure 53: Density vs Agent 1 Blocks Hit (Black: number of blocks hit by Agent1).

x_1	y_1
0	0
3.33	6.161290323
6.66	14.25806452
9.99	23.70967742
13.32	32.12903226
16.65	41.74193548
19.98	52.74193548
23.31	67.25806452
26.64	81.19354839
29.97	115.2903226
33.3	131.0645161

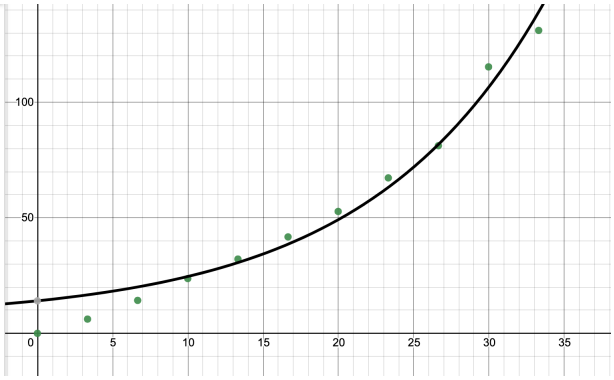


Figure 54: Density vs Agent 2 Blocks Hit (Black: number of blocks hit by Agent 2).

Problem 9

COMPUTATIONAL ISSUE:

One thing that you'll run into building these inference engines is a computational overhead. There are a number of ways to try to cut down on this - for instance observing that new data won't be relevant to inferences all over the board (at least not directly) and there are plenty of nodes that you aren't able to infer anything about at a given time. What steps do you take to try to minimize your computational burden?

ANSWER:

As mentioned above, there are certain practices that are adopted so that we limit the burden of computational overhead when trying to infer things about the maze. Like the professor said in the announcement, new data / observations are not necessarily important for all over the board. In fact, most times, those new pieces of information are only important to the immediate neighborhood (possibly the neighbors of neighbors, but that's specifically in Agent 4's territory).

With this in mind, we purposefully designed the inference agent to operate on a "ripple" principle. To clarify, when the agent gets a new piece of information, we check around us to see if anything is affected (certainly observations will be), and if another cell is able to be confirmed either blocked or empty, we can begin a ripple effect out from there as well. With this approach, we don't have to look at every cell on the board, which can be quite a large number, and it gives us some flexibility as to how far we infer, based on the momentum of the information that's being uncovered (it can propagate out really wide if that was a key piece of information almost like a large rock or it can stop at a really short radius like a small gnat landing on the water).

This also led to a key choice in the design of Agent 4. As mentioned in the Agent 4 section, we didn't devote the time and resources to trying to find the potentially perfect synthesis of equations that may lead us to infer a random spot on the board. This would be very cumbersome with a very limited payoff on average. For example, another principle we observed when testing and refining the project is that the information we could determine in the maze spread out behind us on our previous path outward like a small wake off the stern of a ship moving through water (apologies for all of my amphibious analogies). It's held in by the lack of sensing that we're able to do outside of our path. With this in mind, we can see that trying to determine information for what's to come (the cells unexplored) is an uphill battle.

Much like in the game Minesweeper, it is often difficult to find a breach point in the nebulous cloud of unconfirmed cells, and it takes a lot of brain / computing power to figure out (and even then, sometimes you have to guess). This is why we chose to simply try to focus on seeing if we could determine if our next step in the path is safe or not. This directed us towards a goal, much like only inferring for the immediate neighborhood, and we get that added benefit that the other neighbors may be inferred because of their proximity to the next cell in the path.

The drawback is that we aren't inferring everything there is to be inferred, and this could potentially lead us to have information that's at our fingertips slip through the cracks. However, we made a judgment call that it would be better to not compromise speed for a potentially slight improvement in performance from what we're already experiencing.

Problem 10

APPENDIX

```
// AUTHOR: Zachary Tarman (zpt2)

import java.util.ArrayList;
import java.util.LinkedList;
import java.awt.Point;

public class Agent3 {

        public Maze maze;
        public int rows; // ROW DIMENSION OF MAZE
        public int cols; // COLUMN DIMENSION OF MAZE

        public int collisions = 0; // THE NUMBER OF BLOCKS THE AGENT PHYSICALLY HITS
    public int cellsProcessed = 0; // THE NUMBER OF CELLS THAT WE "EXPLORE" OR POP OFF THE PLANNED PATH
        public int trajectoryLength = 0; // THE TRAJECTORY LENGTH OF THE AGENT
    public int shortestPathFound = 0; // THE LENGTH OF THE SHORTEST PATH FOUND BY THE AGENT WHILE TRAVERSING THE MAZE
    public long runtime = 0; // THE RUNTIME OF THE PROGRAM TO FIND A PATH TO THE GOAL



    // USED TO PRINT THE ABOVE STATS FOR THE PROJECT
    public void printStats() {
        System.out.println("Statistics for Maze Solution");
        System.out.println("Trajectory Length: " + trajectoryLength);
        System.out.println("Cells Popped: " + cellsProcessed);
        System.out.println("Runtime: " + runtime);
        System.out.println("Collisions: " + collisions);
        System.out.println("Shortest Path Found: " + shortestPathFound);
        System.out.println();
        return;
    }



        // PLANNING METHOD (AKA A SINGLE ITERATION OF A* WITHOUT PHYSICALLY MOVING THE AGENT THROUGH THE MAZE)
        public LinkedList<CellInfo> plan(CellInfo start) {

                LinkedList<CellInfo> plannedPath = new LinkedList<CellInfo>(); // TO STORE THE NEW PLANNED PATH
                ArrayList<CellInfo> toExplore = new ArrayList<CellInfo>(); // TO STORE THE CELLS TO BE EXPLORED
                ArrayList<CellInfo> doneWith = new ArrayList<CellInfo>();

                CellInfo curr = start; // PTR TO THE CURRENT CELL WE'RE EVALUATING TO MOVE ON FROM IN OUR PLAN
                Point curr_position; // THE COORDINATE OF THE CURRENT CELL THAT WE'RE LOOKING AT
                int x, y; // THE X AND Y VALUES OF THE COORDINATE FOR THE CURRENT CELL THAT WE'RE LOOKING AT (FOR FINDING NEIGHBORING COORDINATES)
                boolean addUp, addDown, addLeft, addRight; // TO INDICATE IF WE CAN PLAN TO GO IN THAT DIRECTION FROM CURRENT CELL

                Point up = new Point(); // COORDINATE OF NORTH NEIGHBOR
                Point down = new Point(); // COORDINATE OF SOUTH NEIGHBOR
                Point left = new Point(); // COORDINATE OF WEST NEIGHBOR
                Point right = new Point(); // COORDINATE OF EAST NEIGHBOR

                // DEBUGGING STATEMENT
                // System.out.println("We're in a new planning phase.");

                CellInfo first = curr; // THIS IS TO MARK WHERE THE REST OF PLANNING WILL CONTINUE FROM
                toExplore.add(first);

                // BEGIN LOOP UNTIL PLAN REACHES GOAL
                while (toExplore.size() > 0) {

                        curr = toExplore.remove(0); // CURRENT CELL THAT WE'RE LOOKING AT

                        if (contains(curr, doneWith)) { // WE DON'T WANT TO EXPAND THE SAME CELL AGAIN (THIS IS HERE JUST IN CASE)
                                // System.out.println("We've already seen this cell and its directions: " + curr.getPos().toString());
                                continue;
                        }

                        // DEBUGGING STATEMENT
                        // System.out.println("We're currently figuring out where to plan to go to next from " + curr.getPos().toString());

                        curr_position = curr.getPos(); // COORDINATE OF THE CELL WE'RE CURRENTLY EXPLORING
                        x = (int) curr_position.getX(); // X COORDINATE
                        y = (int) curr_position.getY(); // Y COORDINATE
                        doneWith.add(curr); // WE DON'T WANT TO EXPAND / LOOK AT THIS CELL AGAIN IN THIS PLANNING PHASE

                        // IS THIS CELL THE GOAL??
                        // IF SO, LET'S TRACE BACK TO OUR STARTING POSITION
                        if (x == cols - 1 && y == rows - 1) {
                                CellInfo goal = maze.getCell(cols - 1, rows - 1);
                                CellInfo ptr = goal;
                                while (ptr.getPos().getX() != first.getPos().getX() || ptr.getPos().getY() != first.getPos().getY()) {
                                        // DEBUGGING STATEMENT
                                        // System.out.print("(" + ptr.getPos().getX() + "," + ptr.getPos().getY() + "), ");
                                        plannedPath.addFirst(ptr);
                                        ptr = ptr.getParent();
                                }
                                plannedPath.addFirst(ptr); // ADDING START CELL TO THE PATH
                                return plannedPath;
                        }
```

```
// DETERMINE POSSIBLE PLACES TO MOVE FROM CURRENT POSITION
up.setLocation(x, y - 1); // NORTH
down.setLocation(x, y + 1); // SOUTH
left.setLocation(x - 1, y); // WEST
right.setLocation(x + 1, y); // EAST
addUp = true;
addDown = true;
addLeft = true;
addRight = true;

// CHECK FOR CELLS WE CAN'T / SHOULDN'T EXPLORE OR MOVE INTO ON OUR WAY TO THE GOAL
// CHECKS FOR NORTHBOUND NEIGHBOR
if (!inBounds(up)) {
        addUp = false;
} else if (maze.getCell((int) up.getX(), (int) up.getY()).isBlocked()) {
        addUp = false;
        // System.out.println("North is blocked.");
} else if (maze.getCell((int) up.getX(), (int) up.getY()).isVisited()) {
        addUp = false;
        // System.out.println("North is visited.");
} else if (contains(maze.getCell((int) up.getX(), (int) up.getY()), doneWith)) {
        addUp = false;
}

// CHECKS FOR SOUTHBOUND NEIGHBOR
if (!inBounds(down)) {
        addDown = false;
} else if (maze.getCell((int) down.getX(), (int) down.getY()).isBlocked()) {
        addDown = false;
        // System.out.println("South is blocked.");
} else if (maze.getCell((int) down.getX(), (int) down.getY()).isVisited()) {
        addDown = false;
        // System.out.println("South is visited.");
} else if (contains(maze.getCell((int) down.getX(), (int) down.getY()), doneWith)) {
        addDown = false;
}

// CHECKS FOR WESTBOUND NEIGHBOR
if (!inBounds(left)) {
        addLeft = false;
} else if (maze.getCell((int) left.getX(), (int) left.getY()).isBlocked()) {
        addLeft = false;
        // System.out.println("West is blocked.");
} else if (maze.getCell((int) left.getX(), (int) left.getY()).isVisited()) {
        addLeft = false;
        // System.out.println("West is visited.");
} else if (contains(maze.getCell((int) left.getX(), (int) left.getY()), doneWith)) {
        addLeft = false;
}

// CHECKS FOR EASTBOUND NEIGHBOR
if (!inBounds(right)) {
        addRight = false;
} else if (maze.getCell((int) right.getX(), (int) right.getY()).isBlocked()) {
        addRight = false;
        // System.out.println("East is blocked.");
} else if (maze.getCell((int) right.getX(), (int) right.getY()).isVisited()) {
        addRight = false;
        // System.out.println("East is visited.");
} else if (contains(maze.getCell((int) right.getX(), (int) right.getY()), doneWith)) {
        addRight = false;
}

// ADD ALL UNVISITED, UNBLOCKED AND NOT-LOOKED-AT-ALREADY CELLS TO PRIORITY QUEUE + SET PARENTS AND G-VALUES
CellInfo temp;
double curr_g = curr.getG(); // THE G_VALUE OF THE CURRENT CELL IN THE PLANNING PROCESS
if (addUp) { // THE CELL TO OUR NORTH IS A CELL WE CAN EXPLORE
        temp = maze.getCell((int) up.getX(), (int) up.getY());
        temp.setG(curr_g + 1);
        temp.setParent(curr);
        toExplore = insertCell(temp, toExplore);
        // DEBUGGING STATEMENT
        /* System.out.println("Inserting the north cell " + temp.getPos().toString() +
                        " into the priority queue. Its parent is " + temp.getParent().getPos().toString() +
                        " and its f / g values are: " + temp.getF() + ", " + temp.getG()); */
}
if (addLeft) { // THE CELL TO OUR WEST IS A CELL WE CAN EXPLORE
        temp = maze.getCell((int) left.getX(), (int) left.getY());
        temp.setG(curr_g + 1);
        temp.setParent(curr);
        toExplore = insertCell(temp, toExplore);
        // DEBUGGING STATEMENT
        /* System.out.println("Inserting the west cell " + temp.getPos().toString() +
                        " into the priority queue. Its parent is " + temp.getParent().getPos().toString() +
                        " and its f / g values are: " + temp.getF() + ", " + temp.getG()); */
}
if (addDown) { // THE CELL TO OUR SOUTH IS A CELL WE CAN EXPLORE
        temp = maze.getCell((int) down.getX(), (int) down.getY());
        temp.setG(curr_g + 1);
        temp.setParent(curr);
        toExplore = insertCell(temp, toExplore);
        // DEBUGGING STATEMENT
        /* System.out.println("Inserting the south cell " + temp.getPos().toString() +
                        " into the priority queue. Its parent is " + temp.getParent().getPos().toString() +
                        " and its f / g values are: " + temp.getF() + ", " + temp.getG()); */

}
if (addRight) { // THE CELL TO OUR EAST IS A CELL WE CAN EXPLORE
        temp = maze.getCell((int) right.getX(), (int) right.getY());
```

```
                        temp.setG(curr_g + 1);
                        temp.setParent(curr);
                        toExplore = insertCell(temp, toExplore);
                        // DEBUGGING STATEMENT
                        /* System.out.println("Inserting the east cell " + temp.getPos().toString() +
                                        " into the priority queue. Its parent is " + temp.getParent().getPos().toString() +
                                        " and its f / g values are: " + temp.getF() + ", " + temp.getG()); */
                }


                // IF WE HAVE NO CELLS LEFT TO EXPLORE, WE NEED TRY AGAIN FROM A DIFFERENT POINT IN OUR PARENTAGE
                // IF THERE ARE NO OTHER VIABLE OPTIONS, WE'LL TERMINATE
                if (toExplore.size() == 0) {
                        // System.out.println("We have to backtrack.");
                        CellInfo lastChance = backtrack(first, doneWith);
                        if (lastChance != null) {
                                // System.out.println("The new beginning point for this planning phase is " + lastChance.getPos().toString
                                first = lastChance;
                                toExplore.add(lastChance);
                        }
                }

        }

        // System.out.println("We've reached here for some reason.");
        return null; // MAZE IS UNSOLVABLE ):
}




// SENSING METHOD
public void sense(CellInfo pos) {

        // DEBUGGING STATEMENT
        //System.out.println("We're currently in the sensing phase for " + pos.getPos().toString());

        int blocked_neighbors = 0; // COUNTER TO KEEP TRACK OF HOW MANY NEIGHBORS ARE SENSED TO BE BLOCKED
        Point coor = pos.getPos();
        int x = (int) coor.getX();
        int y = (int) coor.getY();

        // ALL THE POSSIBLE NEIGHBORS OF THE CURRENT CELL
                // NOT ALL OF THESE NEIGHBORS ARE LEGITIMATE CELLS (I.E. IN THE BOUNDS OF THE MAZE)
                // THIS IS TAKEN CARE OF BY THE inbounds() HELPER METHOD
        Point n = new Point(x, y - 1);
        Point nw = new Point(x - 1, y - 1);
        Point w = new Point(x - 1, y);
        Point sw = new Point(x - 1, y + 1);
        Point s = new Point(x, y + 1);
        Point se = new Point(x + 1, y + 1);
        Point e = new Point(x + 1, y);
        Point ne = new Point(x + 1, y - 1);

        ArrayList<Point> neighbors = new ArrayList<Point>();
        neighbors.add(n);
        neighbors.add(nw);
        neighbors.add(w);
        neighbors.add(sw);
        neighbors.add(s);
        neighbors.add(se);
        neighbors.add(e);
        neighbors.add(ne);

        for (int i = 0; i < neighbors.size(); i++) {
                if (inBounds(neighbors.get(i))) { // ONLY CHECK NEIGHBORING CELLS THAT ARE ACTUALLY IN BOUNDS (I.E. A LEGITIMATE CELL)
                        if (maze.getCell((int) neighbors.get(i).getX(), (int) neighbors.get(i).getY()).isActuallyBlocked()) { // CHECK ACT
                                blocked_neighbors++;
                        }
                }
        }

        // BUT WE DON'T REPORT THE LOCATIONS OF BLOCKS, JUST THE TOTAL NUMBER SURROUNDING THE CURRENT CELL
        maze.getCell(x, y).setBlocksSensed(blocked_neighbors);

        // DEBUGGING STATEMENT
        //System.out.println("We sensed " + blocked_neighbors + " blocks around us in this cell.");

        return;
}




// INFERENCE METHOD (ACTS RECURSIVELY IF REQUIRED BY GIVEN UPDATES TO THE KNOWLEDGE BASE)
public void infer(CellInfo pos) {

        // DEBUGGING STATEMENT
        // System.out.println("We're currently in the inference phase for cell " + pos.getPos().getX() + ", " + pos.getPos().getY());

        LinkedList<CellInfo> propagate = new LinkedList<CellInfo>(); // TO BE USED IN PROPAGATING INFERENCES
        Point coor = pos.getPos();
        int x = (int) coor.getX();
        int y = (int) coor.getY();
```

```
        checkSurroundings(pos); // UPDATING KNOWLEDGE BASE, REFER TO HELPER METHOD BELOW

        // THESE TWO WILL INDICATE IF WE'VE INFERRED SOMETHING ABOUT THE REMAINING
                // UNCONFIRMED NEIGHBORS (EITHER THE REST ARE BLOCKED, OR THE REST ARE EMPTY)
        boolean restBlocked = false;
        boolean restEmpty = false;

        if (pos.getNeighborsUnconfirmed() != 0) { // IF IT DID EQUAL 0, THERE WOULD BE NOTHING MORE TO INFER (WE JUST NEED TO UPDATE NEIGH
                if (pos.getBlocksSensed() >= 0) { // IF WE HAVEN'T SCANNED YET, WE DON'T WANT TO PREMATURELY MAKE INFERENCES
                        if (pos.getNeighborsBlocked() == pos.getBlocksSensed()) { // EQUIVALENT TO THE Cx = Nx INFERENCE
                                restEmpty = true; // WE NOW KNOW THAT THE REMAINING UNCONFIRMED NEIGHBORS MUST BE EMPTY
                        } else if (pos.getNeighborsEmpty() == pos.getNeighbors() - pos.getBlocksSensed()) { // EQUIVALENT TO THE Ex = Nx -
                                restBlocked = true; // WE NOW KNOW THAT THE REMAINING UNCONFIRMED NEIGHBORS MUST BE BLOCKED
                        }
                }

        }

        // ALL THE POSSIBLE NEIGHBORS, POSSIBLY REDUCED BY THE INBOUNDS CHECKER
        Point n = new Point(x, y - 1);
        Point nw = new Point(x - 1, y - 1);
        Point w = new Point(x - 1, y);
        Point sw = new Point(x - 1, y + 1);
        Point s = new Point(x, y + 1);
        Point se = new Point(x + 1, y + 1);
        Point e = new Point(x + 1, y);
        Point ne = new Point(x + 1, y - 1);

        ArrayList<Point> neighbors = new ArrayList<Point>();
        neighbors.add(n);
        neighbors.add(nw);
        neighbors.add(w);
        neighbors.add(sw);
        neighbors.add(s);
        neighbors.add(se);
        neighbors.add(e);
        neighbors.add(ne);

        // NOW WE UPDATE THE NEIGHBORS KNOWLEDGE BASES
        for (int i = 0; i < neighbors.size(); i++) {

                if (inBounds(neighbors.get(i))) { // ONLY CHECK NEIGHBORING CELLS THAT ARE ACTUALLY IN BOUNDS (I.E. A LEGITIMATE CELL)

                        int neighbor_x = (int) neighbors.get(i).getX();
                        int neighbor_y = (int) neighbors.get(i).getY();
                        // WE CAN INFER / CONFIRM SOMETHING ABOUT THE NEIGHBOR'S STATUS
                        if (maze.getCell(neighbor_x, neighbor_y).isUnconfirmed() && (restBlocked || restEmpty)) {
                                if (restBlocked) {
                                        maze.getCell(neighbor_x, neighbor_y).setBlocked();
                                        // System.out.println("We've inferred " + neighbor_x + ", " + neighbor_y + " to be blocked.");
                                } else if (restEmpty) {
                                        maze.getCell(neighbor_x, neighbor_y).setEmpty();
                                        // System.out.println("We've inferred " + neighbor_x + ", " + neighbor_y + " to be empty.");
                                }
                                // NOW THAT THIS CELL'S STATUS HAS BEEN UPDATED, THEIR NEIGHBORS MIGHT BE INTERESTED IN UPDATING THEIR KNO
                                // WE'RE NOT SENSING (DIFFERENT PHASE) AROUND THIS NEIGHBOR CELL BUT WE CAN POSSIBLY INFER SOMETHING AS WE
                                propagate.add(maze.getCell(neighbor_x, neighbor_y)); // WILL ALSO UPDATE THIS NEIGHBOR BELOW THIS LOOP
                        } else { // THE SURROUNDINGS WILL BE CHECKED ON THE NEXT LEVEL DOWN IF THE FIRST CONDITION IS MET
                                // OTHERWISE, WE SHOULD UPDATE THEM HERE
                                checkSurroundings(maze.getCell(neighbor_x, neighbor_y));

                                if (maze.getCell(neighbor_x, neighbor_y).getNeighborsUnconfirmed() != 0) { // IF IT DID EQUAL 0, THERE WOU
                                        if (maze.getCell(neighbor_x, neighbor_y).getBlocksSensed() >= 0) { // IF WE HAVEN'T SCANNED YET, W
                                                if ((maze.getCell(neighbor_x, neighbor_y).getNeighborsBlocked() == maze.getCell(neighbor_x
                                                                (maze.getCell(neighbor_x, neighbor_y).getNeighborsEmpty() ==
                                                                maze.getCell(neighbor_x, neighbor_y).getNeighbors() - maze.getCell(neighbo
                                                        propagate.add(maze.getCell(neighbor_x, neighbor_y)); // WE CAN INFER THINGS PAST T
                                                }
                                        }

                                }
                        }

                }
        }

        while (propagate.peek() != null) { // RECURSIVE FUNCTION TO MAKE SURE WE UPDATE ALL CELLS ACCORDINGLY
                infer(propagate.poll());
        }

        // NOW WE'RE DONE INFERRING
        return;
}




// HELPER METHODS FOR THE DIFFERENT STEPS OF THE AGENT'S ALGORITHM
public CellInfo backtrack(CellInfo pos, ArrayList<CellInfo> doneWith) { // USED TO BACKTRACK IF AGENT IS STUCK

        Point curr_position; // THE COORDINATE OF THE CURRENT CELL THAT WE'RE LOOKING AT
        int x, y; // THE X AND Y VALUES OF THE COORDINATE FOR THE CURRENT CELL THAT WE'RE LOOKING AT (FOR FINDING NEIGHBORING COORDINATES)

        Point up = new Point(); // COORDINATE OF NORTH NEIGHBOR
        Point down = new Point(); // COORDINATE OF SOUTH NEIGHBOR
        Point left = new Point(); // COORDINATE OF WEST NEIGHBOR
```

```
        Point right = new Point(); // COORDINATE OF EAST NEIGHBOR

        // LOOP TO TAKE CARE OF BACKTRACKING IF NO VIABLE MOVES "FORWARD" (I.E. AT END OF LONG HALLWAY)
        do {
                curr_position = pos.getPos();
                x = (int) curr_position.getX();
                y = (int) curr_position.getY();

                // DEBUGGING STATEMENT
                // System.out.println("We're evaluating " + pos.getPos().toString() + " for viable neighbors.");

                up.setLocation(x, y - 1); // NORTH
                down.setLocation(x, y + 1); // SOUTH
                left.setLocation(x - 1, y); // WEST
                right.setLocation(x + 1, y); // EAST

                // FOR EACH POSSIBLE NEIGHBOR, WE CHECK IF IT'S IN BOUNDS,
                        // IF IT'S INFERRED / OBSERVED TO BE BLOCKED,
                        // AND IF IT HAS ALREADY BEEN VISITED
                                // IF IT'S ALREADY BEEN VISITED, THEN WE KNOW THAT
                                // IT'S NOT WORTH EXPLORING THAT AREA AGAIN
                                // THE ONLY EXCEPTION TO THIS WILL BE IF WE NEED TO BACKTRACK
                                // WHICH IS EXPLAINED BELOW
                if (inBounds(up)) {
                        if (!(maze.getCell((int) up.getX(), (int) up.getY()).isBlocked())) {
                                if (!(maze.getCell((int) up.getX(), (int) up.getY()).isVisited())) {
                                        if (!(contains(maze.getCell((int)up.getX(), (int)up.getY()), doneWith))) {
                                                // DEBUGGING STATEMENT
                                                // System.out.println("We've hit a break north condition in the backtracking.");
                                                break;
                                        }
                                }
                        }
                }

                if (inBounds(down)) {
                        if (!(maze.getCell((int) down.getX(), (int) down.getY()).isBlocked())) {
                                if (!(maze.getCell((int) down.getX(), (int) down.getY()).isVisited())) {
                                        if (!(contains(maze.getCell((int)down.getX(), (int)down.getY()), doneWith))) {
                                                // DEBUGGING STATEMENT
                                                // System.out.println("We've hit a break south condition in the backtracking.");
                                                break;
                                        }
                                }
                        }
                }

                if (inBounds(left)) {
                        if (!(maze.getCell((int) left.getX(), (int) left.getY()).isBlocked())) {
                                if (!(maze.getCell((int) left.getX(), (int) left.getY()).isVisited())) {
                                        if (!(contains(maze.getCell((int)left.getX(), (int)left.getY()), doneWith))) {
                                                // DEBUGGING STATEMENT
                                                // System.out.println("We've hit a break west condition in the backtracking.");
                                                break;
                                        }
                                }
                        }
                }

                if (inBounds(right)) {
                        if (!(maze.getCell((int) right.getX(), (int) right.getY()).isBlocked())) {
                                if (!(maze.getCell((int) right.getX(), (int) right.getY()).isVisited())) {
                                        if (!(contains(maze.getCell((int)right.getX(), (int)right.getY()), doneWith))) {
                                                // DEBUGGING STATEMENT
                                                // System.out.println("We've hit a break east condition in the backtracking.");
                                                break;
                                        }
                                }
                        }
                }

                // IF WE'RE STILL HERE, THEN WE KNOW THAT THERE ARE NO VIABLE NEIGHBORS THAT WE HAVEN'T ALREADY VISITED
                // ARE WE AT THE START NODE RIGHT NOW?
                if (x == 0 && y == 0) { // WE'RE STILL AT THE START NODE HERE, AND THERE'S NOWHERE TO GO
                        return null; // MAZE ISN'T SOLVABLE :(
                }

                // THE ONLY HOPE FOR FINDING A VIABLE PATH AT THIS POINT WOULD BE TO BACKTRACK
                pos = pos.getParent(); // WE'VE BACKTRACKED AND WE WILL START THIS PROCESS AGAIN
                trajectoryLength++; // WE INCLUDE THIS BACKTRACKING IN THE TRAJECTORY LENGTH

        } while (true);


        return pos;
}



public boolean canMove(CellInfo pos, LinkedList<CellInfo> path) { // CHECK IF A CELL CAN ACTUALLY MOVE TO THE NEXT CELL OR NOT

        if (path.peekFirst().isActuallyBlocked()) {
                // System.out.println("Cell " + path.peekFirst().getPos().getX() + ", " + path.peekFirst().getPos().getY() + " is blocked.
                return false;
        }
        return true;
}
```

```
public boolean inBounds(Point coor) { // CHECK IF A NEIGHBORING CELL'S COORDINATES ARE IN BOUNDS (DON'T CHECK NON-EXISTENT CELLS)
        if (coor.getX() < 0 || coor.getX() >= cols || coor.getY() < 0 || coor.getY() >= rows) {
                return false;
        }
        return true;
}



public boolean contains(CellInfo newCell, ArrayList<CellInfo> doneWith) { // TO USE FOR THE "CLOSED LIST" IN THE PLANNING PHASE
        for (int i = 0; i < doneWith.size(); i++) {
                if (newCell.getPos().getX() == doneWith.get(i).getPos().getX()
                                && newCell.getPos().getY() == doneWith.get(i).getPos().getY()) {
                        return true;
                }
        }
        return false;
}



public ArrayList<CellInfo> insertCell(CellInfo newCell, ArrayList<CellInfo> toExplore) { // TO USE FOR IMPLEMENTING A PRIORITY QUEUE IN TH

        // FIRST CELL TO BE ADDED TO AN EMPTY LIST
        if (toExplore.isEmpty()) {
                toExplore.add(newCell);
                return toExplore;
        }

        double cell_f = newCell.getF();
        for (int i = 0; i < toExplore.size(); i++) { // WE WANT TO CHECK IF IT'S ALREADY IN THE LIST
                if (newCell.getPos().getX() == toExplore.get(i).getPos().getX() && newCell.getPos().getY() == toExplore.get(i).getPos().ge
                        if (cell_f <= toExplore.get(i).getF()) { // IF THE EXISTING F-VALUE IS HIGHER THAN WE JUST FOUND, WE WANT TO UPDAT
                                toExplore.remove(i);
                        } else { // OTHERWISE, WE JUST RETURN THE LIST AS IS
                                return toExplore;
                        }
                }
        }

        if (toExplore.isEmpty()) {
                toExplore.add(newCell);
                return toExplore;
        }

        // IF OUR CELL HAS A BETTER F-VALUE OR THE CELL ISN'T IN THE LIST ALREADY, THEN WE ADD IT HERE
        for (int i = 0; i < toExplore.size(); i++) {
                if (cell_f < toExplore.get(i).getF()) {
                        toExplore.add(i, newCell);
                        return toExplore;
                } else if (cell_f == toExplore.get(i).getF()) {
                        double cell_g = newCell.getG();
                        if (cell_g <= toExplore.get(i).getG()) {
                                toExplore.add(i, newCell);
                                return toExplore;
                        }
                }
        }

        // THE CELL WE FOUND HAS THE HIGHEST F-VALUE OF ANY WE FOUND SO FAR
        toExplore.add(toExplore.size() - 1, newCell); // ADDING IT TO THE END OF THE LIST
        return toExplore;

}



public void checkSurroundings(CellInfo pos) { // THIS IS USED TO UPDATE THE KNOWLEDGE BASE
        // ESSENTIALLY, WE JUST OBSERVE A NEIGHBOR'S SURROUNDINGS

        Point coor = pos.getPos();
        int x = (int) coor.getX();
        int y = (int) coor.getY();

        int blocked = 0; // HOW MANY NEIGHBORS ARE INFERRED / OBSERVED TO BE BLOCKED?
        int empty = 0; // HOW MANY NEIGHBORS ARE INFERRED / OBSERVED TO BE EMPTY?
        int unconfirmed = 0; // HOW MANY NEIGHBORS HAVE AN UNCONFIRMED STATUS?

        // ALL THE POSSIBLE NEIGHBORS, POSSIBLY REDUCED BY THE INBOUNDS CHECKER
        Point n = new Point(x, y - 1);
        Point nw = new Point(x - 1, y - 1);
        Point w = new Point(x - 1, y);
        Point sw = new Point(x - 1, y + 1);
        Point s = new Point(x, y + 1);
        Point se = new Point(x + 1, y + 1);
        Point e = new Point(x + 1, y);
        Point ne = new Point(x + 1, y - 1);

        ArrayList<Point> neighbors = new ArrayList<Point>();
        neighbors.add(n);
        neighbors.add(nw);
        neighbors.add(w);
        neighbors.add(sw);
        neighbors.add(s);
        neighbors.add(se);
        neighbors.add(e);
        neighbors.add(ne);
```

```
        // UPDATE OUR KNOWLEDGE BASE FOR JUST THE CELL WE ARE IN
        // THIS IS SEPARATE FROM THE SENSING PHASE
        for (int i = 0; i < neighbors.size(); i++) {
                if (inBounds(neighbors.get(i))) {

                        int neighbor_x = (int) neighbors.get(i).getX();
                        int neighbor_y = (int) neighbors.get(i).getY();

                        if (maze.getCell(neighbor_x, neighbor_y).isBlocked()) {
                                blocked++;
                        } else if (!maze.getCell(neighbor_x, neighbor_y).isUnconfirmed()) {
                                empty++;
                        } else {
                                unconfirmed++;
                        }
                }
        }
        pos.setNeighborsBlocked(blocked);
        pos.setNeighborsEmpty(empty);
        pos.setNeighborsUnconfirmed(unconfirmed);
        // DEBUGGING STATEMENT
        /*System.out.println("Neighbors blocked: " + pos.getNeighborsBlocked()
                + ". Neighbors empty: " + pos.getNeighborsEmpty()
                + ". Neighbors unconfirmed: " + pos.getNeighborsUnconfirmed()); */
        return;

}



// THIS IS THE METHOD THAT POWERS THE ENTIRE ALGORITHM, RUNNING THE AGENT UNTIL GOAL CELL OR IT DETERMINES THERE'S NO PATH TO GOAL
public static char run(int rowNum, int colNum, double prob) {

        Agent3 mazeRunner = new Agent3(); // KEEPS TRACK OF ALL OF OUR DATA AND STRUCTURES

        // READING FROM INPUT
        mazeRunner.rows = rowNum; // THE NUMBER OF ROWS THAT WE WANT IN THE CONSTRUCTED MAZE
        mazeRunner.cols = colNum; // THE NUMBER OF COLUMNS THAT WE WANT IN THE CONSTRUCTED MAZE
        double p = prob; // VALUE BETWEEN 0.0 AND 1.0

        // SET UP MAZE
        mazeRunner.maze = new Maze(mazeRunner.rows, mazeRunner.cols, p);
        boolean badPath = false; // USED FOR DETERMINING WITH WE'VE FOUND A BAD PATH BY INFERENCE

        long begin = System.nanoTime();
        CellInfo start = mazeRunner.maze.getCell(0, 0);
        LinkedList<CellInfo> plannedPath = mazeRunner.plan(start); // STORES OUR BEST PATH THROUGH THE MAZE

        // MAIN LOOP FOR AGENT TO FOLLOW AFTER FIRST PLANNING PHASE
        while (true) {

                // EXTRACT THE NEXT CELL IN THE PLANNED PATH
                CellInfo currCell = plannedPath.poll();

                // DEBUGGING STATEMENT
                // System.out.println("Agent is currently in " + currCell.getPos().getX() + ", " + currCell.getPos().getY());

                // HAVE WE HIT THE GOAL CELL YET?
                if (currCell.getPos().getX() == mazeRunner.cols - 1 && currCell.getPos().getY() == mazeRunner.rows - 1) { // WE'VE HIT THE
                        break;
                }

                // WE HAVEN'T HIT THE GOAL CELL, SO WE CONTINUE ONWARD
                currCell.setEmpty(); // THE CELL WE ARE CURRENTLY IS EMPTY / UNBLOCKED
                if (!currCell.isVisited()) { // IF THE CELL HAS ALREADY BEEN VISITED THEN WE DON'T NEED TO SENSE (SAVING SOME COMPUTATIONA
                        mazeRunner.sense(currCell); // SENSE HOW MANY NEIGHBORS ARE BLOCKED (BUT NOT WHERE THE BLOCKS ARE)
                }
                currCell.setVisited(); // WE HAVE NOW OFFICIALLY VISITED THIS CELL
                mazeRunner.infer(currCell); // INFER WHAT WE CAN ABOUT OUR SURROUNDINGS AND UPDATE KNOWLEDGE BASE

                // HAVE WE HAD AN UPDATE THAT REQUIRES US TO REPLAN?
                // CHECK IF WE'VE HAD AN UPDATE (BLOCK) IN THE PATH THAT REQUIRES US TO REPLAN
                for (int i = 0; i < plannedPath.size(); i++) {
                        CellInfo temp = plannedPath.poll();
                        if (temp.isBlocked()) { // THIS IS CHECKING THE INFERRED / OBSERVED BLOCKS, NOT SNEAKING A PEEK AT THE LEGITIMATE
                                badPath = true;
                                break;
                        }
                        plannedPath.addLast(temp); // CYCLE THIS CELL TO THE BACK
                        // IF ALL CELLS ARE STILL THOUGHT TO BE UNBLOCKED, THEY'LL END UP BACK IN THEIR CORRECT POSITIONS
                }

                // WE'VE FOUND A BLOCK IN OUR PATH WITHOUT ACTUALLY RUNNING INTO IT VIA INFERENCE
                if (badPath) {

                        // DEBUGGING STATEMENT
                        // System.out.println("WE'VE DISCOVERED A BAD PATH WITHOUT COLLISION.");

                        plannedPath = mazeRunner.plan(currCell);
                        if (plannedPath == null) {
                                System.out.println("Maze is unsolvable.\n");
                                // System.out.println(mazeRunner.maze.toString());
                                return 'F';
                        }
                        badPath = false;
                        continue;
                }
```

```
                // OUR PLANNED PATH IS STILL OKAY AS FAR AS WE KNOW IF WE'RE HERE
                // ATTEMPT TO EXECUTE EXACTLY ONE CELL MOVEMENT
                mazeRunner.cellsProcessed++;
                if (!mazeRunner.canMove(currCell, plannedPath)) {
                        // WE'VE FOUND / HIT A BLOCK
                        CellInfo obstruction = plannedPath.peekFirst();
                        obstruction.setBlocked();
                        obstruction.setVisited();
                        mazeRunner.collisions++;

                        // DEBUGGING STATEMENT
                        // System.out.println("We've hit a block at coordinate " + obstruction.getPos().toString());

                        mazeRunner.infer(obstruction); // WE CAN UPDATE OUR KNOWLEDGE BASE
                        // WE CAN'T SENSE THOUGH SINCE WE CAN'T OCCUPY THAT CELL

                        // AND WE NEED TO REPLAN AS WELL
                        plannedPath = mazeRunner.plan(currCell);
                        if (plannedPath == null) {
                                System.out.println("Maze is unsolvable.\n");
                                // System.out.println(mazeRunner.maze.toString());
                                return 'F';
                        }
                        continue;
                }

                // IF WE HAVE SUCCESSFULLY MOVED TO ANOTHER CELL, WE UPDATE THE TRAJECTORY LENGTH
                mazeRunner.trajectoryLength++;


        }

        // IF WE BREAK FROM THE LOOP (AKA WE'RE HERE AND HAVEN'T RETURNED YET), WE KNOW WE FOUND THE GOAL.
        long end = System.nanoTime();
        mazeRunner.runtime = end - begin;

        CellInfo ptr = mazeRunner.maze.getCell(mazeRunner.cols - 1, mazeRunner.rows - 1);
        while (ptr.getPos().getX() != 0 || ptr.getPos().getY() != 0) {

                // DEBUGGING STATEMENT
                // System.out.println("Currently backtracking and at " + ptr.getPos().toString());

                // THE WHOLE PURPOSE OF THE FOLLOWING SEVERAL LINES IS TO MAKE SURE WE'RE COMPUTING THE SHORTEST PATH CORRECTLY
                        // WE MAY HAVE SITUATIONS WHERE WE RUN INTO A BLOCK AND SWERVE AROUND IT WITHOUT REALIZING THAT
                        // IN THE COMPUTATION OF THE SHORTEST PATH, WE COULD'VE AVOIDED THAT DETOUR
                // THIS JUST LOOKS FOR IMMEDIATE NEIGHBORS FARTHER BACK IN THE PARENT CHAIN
                // AND IF WE HAVE A HIT, THEN WE KNOW WE HAVE AN EVEN SHORTER PATH THAN WE THOUGHT AND WE CHANGE THE POINTER
                // OF THE CURRENT CELL WE'RE AT DURING THE BACKTRACKING
                CellInfo temp = ptr.getParent();
                if (temp.getPos().getX() == 0 && temp.getPos().getY() == 0) {
                        ptr.setOnShortestPath(); // FOR PRINTING PURPOSES DURING DEBUGGING
                        mazeRunner.shortestPathFound++;
                        break;
                }
                temp = temp.getParent(); // THIS IS TO INSURE WE ARE PAST THE MOST IMMEDIATE NEIGHBOR OF THE CURRENT CELL
                // OTHERWISE THIS WOULD BE A FRUITLESS VENTURE

                int x = (int)ptr.getPos().getX();
                int y = (int)ptr.getPos().getY();
                while (temp.getPos().getX() != 0 || temp.getPos().getY() != 0) {
                        int x2 = (int)temp.getPos().getX();
                        int y2 = (int)temp.getPos().getY();

                        if ((Math.abs(x - x2) == 1 && y - y2 == 0) || (Math.abs(y - y2) == 1) && x - x2 == 0) {
                                // DEBUGGING STATEMENT
                                // System.out.println("We've determined that " + temp.getPos().toString() + " is a closer neighbor.");
                                ptr.setParent(temp);
                                break;
                        } else {
                                temp = temp.getParent();
                        }
                }

                if (temp.getPos().getX() == 0 && temp.getPos().getY() == 0) {
                        if ((x == 1 && y == 0) || (y == 1) && x == 0) {
                                ptr.setParent(temp);
                        }
                }

                ptr.setOnShortestPath(); // FOR PRINTING PURPOSES DURING DEBUGGING
                mazeRunner.shortestPathFound++;
                ptr = ptr.getParent(); // FOLLOW THE PARENT CHAIN BACK UP UNTIL THE START CELL
        }

        System.out.println("Path Found!");
        // System.out.println(mazeRunner.maze.toString());
        mazeRunner.printStats();

        return 'S';
}

// DRIVER METHOD
public static void main(String args[]) {

        // ROWS, COLUMNS, DENSITY OF BLOCKED CELLS AND THE NUMBER OF SUCCESSFUL PATHS FOUND
                // ALL READ IN AS COMMAND LINE ARGUMENTS
        int rowNum = Integer.parseInt(args[0]);
        int colNum = Integer.parseInt(args[1]);
        double prob = Double.parseDouble(args[2]);
```

```
                int successfulTrials = Integer.parseInt(args[3]);

                while (successfulTrials > 0) {
                        char result = run(rowNum, colNum, prob);
                        if (result == 'S') {
                                successfulTrials--;
                        }
                }

                return;

        }

}



import java.util.ArrayList;
import java.util.LinkedList;
import java.awt.Point;


public class Agent4 {


        public Maze maze;
        public int rows; // ROW DIMENSION OF MAZE
        public int cols; // COLUMN DIMENSION OF MAZE

        public int collisions = 0; // THE NUMBER OF BLOCKS THE AGENT PHYSICALLY HITS
    public int cellsProcessed = 0; // THE NUMBER OF CELLS THAT WE "EXPLORE" OR POP OFF THE PLANNED PATH
        public int trajectoryLength = 0; // THE TRAJECTORY LENGTH OF THE AGENT
    public int shortestPathFound = 0; // THE LENGTH OF THE SHORTEST PATH FOUND BY THE AGENT WHILE TRAVERSING THE MAZE
    public long runtime = 0; // THE RUNTIME OF THE PROGRAM TO FIND A PATH TO THE GOAL

    public ArrayList<Association> facts = new ArrayList<Association>(); // THIS WILL STORE UPDATES TO THE KNOWLEDGE BASE
        // THIS IS ESSENTIALLY CROSS-SECTIONS OF UNCONFIRMED NEIGHBORS FOR DIFFERENT CELLS
        // COMPUTED BY TREATING UNCONFIRMED NEIGHBORS LIKE VARIABLES AND TREATING THE WHOLE MAZE
                // AS A SYSTEM OF EQUATIONS, REDUCING EQUATIONS DOWN TO MINIMAL VARIABLES WHEN POSSIBLE


    // USED TO PRINT THE ABOVE STATS FOR THE PROJECT
    public void printStats() {
        System.out.println("Statistics for Maze Solution");
        System.out.println("Trajectory Length: " + trajectoryLength);
        System.out.println("Cells Popped: " + cellsProcessed);
        System.out.println("Runtime: " + runtime);
        System.out.println("Collisions: " + collisions);
        System.out.println("Shortest Path Found: " + shortestPathFound);
        System.out.println();
        return;
    }



        // PLANNING METHOD (AKA A SINGLE ITERATION OF A* WITHOUT PHYSICALLY MOVING THE AGENT THROUGH THE MAZE)
        public LinkedList<CellInfo> plan(CellInfo start) {

                LinkedList<CellInfo> plannedPath = new LinkedList<CellInfo>(); // TO STORE THE NEW PLANNED PATH
                ArrayList<CellInfo> toExplore = new ArrayList<CellInfo>(); // TO STORE THE CELLS TO BE EXPLORED
                ArrayList<CellInfo> doneWith = new ArrayList<CellInfo>();

                CellInfo curr = start; // PTR TO THE CURRENT CELL WE'RE EVALUATING TO MOVE ON FROM IN OUR PLAN
                Point curr_position; // THE COORDINATE OF THE CURRENT CELL THAT WE'RE LOOKING AT
                int x, y; // THE X AND Y VALUES OF THE COORDINATE FOR THE CURRENT CELL THAT WE'RE LOOKING AT (FOR FINDING NEIGHBORING COORDINATES)
                boolean addUp, addDown, addLeft, addRight; // TO INDICATE IF WE CAN PLAN TO GO IN THAT DIRECTION FROM CURRENT CELL

                Point up = new Point(); // COORDINATE OF NORTH NEIGHBOR
                Point down = new Point(); // COORDINATE OF SOUTH NEIGHBOR
                Point left = new Point(); // COORDINATE OF WEST NEIGHBOR
                Point right = new Point(); // COORDINATE OF EAST NEIGHBOR

                // DEBUGGING STATEMENT
                // System.out.println("We're in a new planning phase.");

                CellInfo first = curr; // THIS IS TO MARK WHERE THE REST OF PLANNING WILL CONTINUE FROM
                toExplore.add(first);

                // BEGIN LOOP UNTIL PLAN REACHES GOAL
                while (toExplore.size() > 0) {

                        curr = toExplore.remove(0); // CURRENT CELL THAT WE'RE LOOKING AT

                        if (contains(curr, doneWith)) { // WE DON'T WANT TO EXPAND THE SAME CELL AGAIN (THIS IS HERE JUST IN CASE)
                                // System.out.println("We've already seen this cell and its directions: " + curr.getPos().toString());
                                continue;
                        }

                        // DEBUGGING STATEMENT
                        // System.out.println("We're currently figuring out where to plan to go to next from " + curr.getPos().toString());

                        curr_position = curr.getPos(); // COORDINATE OF THE CELL WE'RE CURRENTLY EXPLORING
                        x = (int) curr_position.getX(); // X COORDINATE
                        y = (int) curr_position.getY(); // Y COORDINATE
                        doneWith.add(curr); // WE DON'T WANT TO EXPAND / LOOK AT THIS CELL AGAIN IN THIS PLANNING PHASE

                        // IS THIS CELL THE GOAL??
                        // IF SO, LET'S TRACE BACK TO OUR STARTING POSITION
```

```
if (x == cols - 1 && y == rows - 1) {
        CellInfo goal = maze.getCell(cols - 1, rows - 1);
        CellInfo ptr = goal;
        while (ptr.getPos().getX() != first.getPos().getX() || ptr.getPos().getY() != first.getPos().getY()) {
                // DEBUGGING STATEMENT
                // System.out.print("(" + ptr.getPos().getX() + "," + ptr.getPos().getY() + "), ");
                plannedPath.addFirst(ptr);
                ptr = ptr.getParent();
        }
        plannedPath.addFirst(ptr); // ADDING START CELL TO THE PATH
        return plannedPath;
}

// DETERMINE POSSIBLE PLACES TO MOVE FROM CURRENT POSITION
up.setLocation(x, y - 1); // NORTH
down.setLocation(x, y + 1); // SOUTH
left.setLocation(x - 1, y); // WEST
right.setLocation(x + 1, y); // EAST
addUp = true;
addDown = true;
addLeft = true;
addRight = true;

// CHECK FOR CELLS WE CAN'T / SHOULDN'T EXPLORE OR MOVE INTO ON OUR WAY TO THE GOAL
// CHECKS FOR NORTHBOUND NEIGHBOR
if (!inBounds(up)) {
        addUp = false;
} else if (maze.getCell((int) up.getX(), (int) up.getY()).isBlocked()) {
        addUp = false;
        // System.out.println("North is blocked.");
} else if (maze.getCell((int) up.getX(), (int) up.getY()).isVisited()) {
        addUp = false;
        // System.out.println("North is visited.");
} else if (contains(maze.getCell((int) up.getX(), (int) up.getY()), doneWith)) {
        addUp = false;
}

// CHECKS FOR SOUTHBOUND NEIGHBOR
if (!inBounds(down)) {
        addDown = false;
} else if (maze.getCell((int) down.getX(), (int) down.getY()).isBlocked()) {
        addDown = false;
        // System.out.println("South is blocked.");
} else if (maze.getCell((int) down.getX(), (int) down.getY()).isVisited()) {
        addDown = false;
        // System.out.println("South is visited.");
} else if (contains(maze.getCell((int) down.getX(), (int) down.getY()), doneWith)) {
        addDown = false;
}

// CHECKS FOR WESTBOUND NEIGHBOR
if (!inBounds(left)) {
        addLeft = false;
} else if (maze.getCell((int) left.getX(), (int) left.getY()).isBlocked()) {
        addLeft = false;
        // System.out.println("West is blocked.");
} else if (maze.getCell((int) left.getX(), (int) left.getY()).isVisited()) {
        addLeft = false;
        // System.out.println("West is visited.");
} else if (contains(maze.getCell((int) left.getX(), (int) left.getY()), doneWith)) {
        addLeft = false;
}

// CHECKS FOR EASTBOUND NEIGHBOR
if (!inBounds(right)) {
        addRight = false;
} else if (maze.getCell((int) right.getX(), (int) right.getY()).isBlocked()) {
        addRight = false;
        // System.out.println("East is blocked.");
} else if (maze.getCell((int) right.getX(), (int) right.getY()).isVisited()) {
        addRight = false;
        // System.out.println("East is visited.");
} else if (contains(maze.getCell((int) right.getX(), (int) right.getY()), doneWith)) {
        addRight = false;
}

// ADD ALL UNVISITED, UNBLOCKED AND NOT-LOOKED-AT-ALREADY CELLS TO PRIORITY QUEUE + SET PARENTS AND G-VALUES
CellInfo temp;
double curr_g = curr.getG(); // THE G_VALUE OF THE CURRENT CELL IN THE PLANNING PROCESS
if (addUp) { // THE CELL TO OUR NORTH IS A CELL WE CAN EXPLORE
        temp = maze.getCell((int) up.getX(), (int) up.getY());
        temp.setG(curr_g + 1);
        temp.setParent(curr);
        toExplore = insertCell(temp, toExplore);
        // DEBUGGING STATEMENT
        /* System.out.println("Inserting the north cell " + temp.getPos().toString() +
                        " into the priority queue. Its parent is " + temp.getParent().getPos().toString() +
                        " and its f / g values are: " + temp.getF() + ", " + temp.getG()); */
}
if (addLeft) { // THE CELL TO OUR WEST IS A CELL WE CAN EXPLORE
        temp = maze.getCell((int) left.getX(), (int) left.getY());
        temp.setG(curr_g + 1);
        temp.setParent(curr);
        toExplore = insertCell(temp, toExplore);
        // DEBUGGING STATEMENT
        /* System.out.println("Inserting the west cell " + temp.getPos().toString() +
                        " into the priority queue. Its parent is " + temp.getParent().getPos().toString() +
                        " and its f / g values are: " + temp.getF() + ", " + temp.getG()); */
}
```

```
                if (addDown) { // THE CELL TO OUR SOUTH IS A CELL WE CAN EXPLORE
                        temp = maze.getCell((int) down.getX(), (int) down.getY());
                        temp.setG(curr_g + 1);
                        temp.setParent(curr);
                        toExplore = insertCell(temp, toExplore);
                        // DEBUGGING STATEMENT
                        /* System.out.println("Inserting the south cell " + temp.getPos().toString() +
                                        " into the priority queue. Its parent is " + temp.getParent().getPos().toString() +
                                        " and its f / g values are: " + temp.getF() + ", " + temp.getG()); */

                }
                if (addRight) { // THE CELL TO OUR EAST IS A CELL WE CAN EXPLORE
                        temp = maze.getCell((int) right.getX(), (int) right.getY());
                        temp.setG(curr_g + 1);
                        temp.setParent(curr);
                        toExplore = insertCell(temp, toExplore);
                        // DEBUGGING STATEMENT
                        /* System.out.println("Inserting the east cell " + temp.getPos().toString() +
                                        " into the priority queue. Its parent is " + temp.getParent().getPos().toString() +
                                        " and its f / g values are: " + temp.getF() + ", " + temp.getG()); */
                }


                // IF WE HAVE NO CELLS LEFT TO EXPLORE, WE NEED TRY AGAIN FROM A DIFFERENT POINT IN OUR PARENTAGE
                // IF THERE ARE NO OTHER VIABLE OPTIONS, WE'LL TERMINATE
                if (toExplore.size() == 0) {
                        // System.out.println("We have to backtrack.");
                        CellInfo lastChance = backtrack(first, doneWith);
                        if (lastChance != null) {
                                // System.out.println("The new beginning point for this planning phase is " + lastChance.getPos().toString
                                first = lastChance;
                                toExplore.add(lastChance);
                        }
                }

        }

        // System.out.println("We've reached here for some reason.");
        return null; // MAZE IS UNSOLVABLE ):
}

// SENSING METHOD
public void sense(CellInfo pos) {

        // DEBUGGING STATEMENT
        //System.out.println("We're currently in the sensing phase for " + pos.getPos().toString());

        int blocked_neighbors = 0; // COUNTER TO KEEP TRACK OF HOW MANY NEIGHBORS ARE SENSED TO BE BLOCKED
        Point coor = pos.getPos();
        int x = (int) coor.getX();
        int y = (int) coor.getY();

        // ALL THE POSSIBLE NEIGHBORS OF THE CURRENT CELL
                // NOT ALL OF THESE NEIGHBORS ARE LEGITIMATE CELLS (I.E. IN THE BOUNDS OF THE MAZE)
                // THIS IS TAKEN CARE OF BY THE inbounds() HELPER METHOD
        Point n = new Point(x, y - 1);
        Point nw = new Point(x - 1, y - 1);
        Point w = new Point(x - 1, y);
        Point sw = new Point(x - 1, y + 1);
        Point s = new Point(x, y + 1);
        Point se = new Point(x + 1, y + 1);
        Point e = new Point(x + 1, y);
        Point ne = new Point(x + 1, y - 1);

        ArrayList<Point> neighbors = new ArrayList<Point>();
        neighbors.add(n);
        neighbors.add(nw);
        neighbors.add(w);
        neighbors.add(sw);
        neighbors.add(s);
        neighbors.add(se);
        neighbors.add(e);
        neighbors.add(ne);

        for (int i = 0; i < neighbors.size(); i++) {
                if (inBounds(neighbors.get(i))) { // ONLY CHECK NEIGHBORING CELLS THAT ARE ACTUALLY IN BOUNDS (I.E. A LEGITIMATE CELL)
                        if (maze.getCell((int) neighbors.get(i).getX(), (int) neighbors.get(i).getY()).isActuallyBlocked()) { // CHECK ACT
                                blocked_neighbors++;
                        }
                }
        }

        // BUT WE DON'T REPORT THE LOCATIONS OF BLOCKS, JUST THE TOTAL NUMBER SURROUNDING THE CURRENT CELL
        pos.setBlocksSensed(blocked_neighbors);

        // DEBUGGING STATEMENT
        //System.out.println("We sensed " + blocked_neighbors + " blocks around us in this cell.");

        return;
}




// INFERENCE METHOD (ACTS RECURSIVELY IF REQUIRED BY GIVEN UPDATES TO THE KNOWLEDGE BASE)
public void infer(CellInfo pos) {
```

```
// DEBUGGING STATEMENT
// System.out.println("We're currently in the inference phase for cell " + pos.getPos().getX() + ", " + pos.getPos().getY());

LinkedList<CellInfo> propagate = new LinkedList<CellInfo>(); // TO BE USED IN PROPAGATING INFERENCES
Point coor = pos.getPos();
int x = (int) coor.getX();
int y = (int) coor.getY();

checkSurroundings(pos); // UPDATING KNOWLEDGE BASE, REFER TO HELPER METHOD BELOW

// THESE TWO WILL INDICATE IF WE'VE INFERRED SOMETHING ABOUT THE REMAINING
        // UNCONFIRMED NEIGHBORS (EITHER THE REST ARE BLOCKED, OR THE REST ARE EMPTY)
boolean restBlocked = false;
boolean restEmpty = false;

if (pos.getNeighborsUnconfirmed() != 0) { // IF IT DID EQUAL 0, THERE WOULD BE NOTHING MORE TO INFER (WE JUST NEED TO UPDATE NEIGH
        if (pos.getBlocksSensed() >= 0) { // IF WE HAVEN'T SCANNED YET, WE DON'T WANT TO PREMATURELY MAKE INFERENCES
                if (pos.getNeighborsBlocked() == pos.getBlocksSensed()) { // EQUIVALENT TO THE Cx = Nx INFERENCE
                        restEmpty = true; // WE NOW KNOW THAT THE REMAINING UNCONFIRMED NEIGHBORS MUST BE EMPTY
                } else if (pos.getNeighborsEmpty() == pos.getNeighbors() - pos.getBlocksSensed()) { // EQUIVALENT TO THE Ex = Nx -
                        restBlocked = true; // WE NOW KNOW THAT THE REMAINING UNCONFIRMED NEIGHBORS MUST BE BLOCKED
                }
        }

}

// ALL THE POSSIBLE NEIGHBORS, POSSIBLY REDUCED BY THE INBOUNDS CHECKER
Point n = new Point(x, y - 1);
Point nw = new Point(x - 1, y - 1);
Point w = new Point(x - 1, y);
Point sw = new Point(x - 1, y + 1);
Point s = new Point(x, y + 1);
Point se = new Point(x + 1, y + 1);
Point e = new Point(x + 1, y);
Point ne = new Point(x + 1, y - 1);

ArrayList<Point> neighbors = new ArrayList<Point>();
neighbors.add(n);
neighbors.add(nw);
neighbors.add(w);
neighbors.add(sw);
neighbors.add(s);
neighbors.add(se);
neighbors.add(e);
neighbors.add(ne);

// NOW WE UPDATE THE NEIGHBORS KNOWLEDGE BASES
for (int i = 0; i < neighbors.size(); i++) {

        if (inBounds(neighbors.get(i))) { // ONLY CHECK NEIGHBORING CELLS THAT ARE ACTUALLY IN BOUNDS (I.E. A LEGITIMATE CELL)

                int neighbor_x = (int) neighbors.get(i).getX();
                int neighbor_y = (int) neighbors.get(i).getY();

                // WE CAN INFER / CONFIRM SOMETHING ABOUT THE NEIGHBOR'S STATUS
                if (maze.getCell(neighbor_x, neighbor_y).isUnconfirmed() && (restBlocked || restEmpty)) {
                        if (restBlocked) {
                                maze.getCell(neighbor_x, neighbor_y).setBlocked();
                                for (int j = 0; j < facts.size(); j++) { // WE NEED TO UPDATE OUR KNOWLEDGE BASE
                                        removeFromAssociation(maze.getCell(neighbor_x, neighbor_y), j);
                                }
                                // System.out.println("We've inferred " + neighbor_x + ", " + neighbor_y + " to be blocked.");
                        } else if (restEmpty) {
                                maze.getCell(neighbor_x, neighbor_y).setEmpty();
                                for (int j = 0; j < facts.size(); j++) { // WE NEED TO UPDATE OUR KNOWLEDGE BASE
                                        removeFromAssociation(maze.getCell(neighbor_x, neighbor_y), j);
                                }
                                // System.out.println("We've inferred " + neighbor_x + ", " + neighbor_y + " to be empty.");
                        }
                        // NOW THAT THIS CELL'S STATUS HAS BEEN UPDATED, THEIR NEIGHBORS MIGHT BE INTERESTED IN UPDATING THEIR KNO
                        // WE'RE NOT SENSING (DIFFERENT PHASE) AROUND THIS NEIGHBOR CELL BUT WE CAN POSSIBLY INFER SOMETHING AS WE
                        propagate.add(maze.getCell(neighbor_x, neighbor_y)); // WILL ALSO UPDATE THIS NEIGHBOR BELOW THIS LOOP

                } else { // THE SURROUNDINGS WILL BE CHECKED ON THE NEXT LEVEL DOWN IF THE FIRST CONDITION IS MET
                        // OTHERWISE, WE SHOULD UPDATE THEM HERE
                        checkSurroundings(maze.getCell(neighbor_x, neighbor_y));

                        // WE CAN ALSO DO INFERENCE CHECKS HERE, JUST TO SEE IF WE'VE FOUND OUT ANYTHING DURING THIS TIME
                        if (maze.getCell(neighbor_x, neighbor_y).getNeighborsUnconfirmed() != 0) { // IF IT DID EQUAL 0, THERE WOU
                                if (maze.getCell(neighbor_x, neighbor_y).getBlocksSensed() >= 0) { // IF WE HAVEN'T SCANNED YET, W
                                        if ((maze.getCell(neighbor_x, neighbor_y).getNeighborsBlocked() == maze.getCell(neighbor_x
                                                        (maze.getCell(neighbor_x, neighbor_y).getNeighborsEmpty() ==
                                                        maze.getCell(neighbor_x, neighbor_y).getNeighbors() - maze.getCell(neighbo
                                                propagate.add(maze.getCell(neighbor_x, neighbor_y)); // WE CAN INFER THINGS PAST T
                                        }
                                }

                        }
                }

        }
}

while (propagate.peek() != null) { // RECURSIVE FUNCTION TO MAKE SURE WE UPDATE ALL CELLS ACCORDINGLY
        infer(propagate.poll());
}

// NOW WE'RE DONE INFERRING
```

```
        return;
}


// THE POINT OF THIS METHOD WOULD BE TO TRY TO EVALUATE IF OUR NEXT STEP IS GOING TO BE BLOCKED
        // WE'RE TRYING TO LIMIT OUR SCOPE SO THAT WE'RE NOT INFERRING FOREVER, BUT IF WE FIND OUT
        // OTHER STUFF ALONG WITH WAY, THAT'S AN ADDED BONUS
public void inferExtended(CellInfo next) {

        for (int i = 0; i < facts.size(); i++) {
                if (facts.get(i).contains(next) >= 0) {
                        ArrayList<CellInfo> confirmed = facts.get(i).process();
                        if (confirmed != null) {
                                for (int j = 0; j < confirmed.size(); j++) {
                                        for (int k = 0; k < facts.size(); k++) {
                                                removeFromAssociation(confirmed.get(j), k);
                                        }
                                        infer(confirmed.get(j));
                                }
                        }
                }
        }

        return;

}



// HELPER METHODS FOR THE DIFFERENT STEPS OF THE AGENT'S ALGORITHM
public CellInfo backtrack(CellInfo pos, ArrayList<CellInfo> doneWith) { // USED TO BACKTRACK IF AGENT IS STUCK

        Point curr_position; // THE COORDINATE OF THE CURRENT CELL THAT WE'RE LOOKING AT
        int x, y; // THE X AND Y VALUES OF THE COORDINATE FOR THE CURRENT CELL THAT WE'RE LOOKING AT (FOR FINDING NEIGHBORING COORDINATES)

        Point up = new Point(); // COORDINATE OF NORTH NEIGHBOR
        Point down = new Point(); // COORDINATE OF SOUTH NEIGHBOR
        Point left = new Point(); // COORDINATE OF WEST NEIGHBOR
        Point right = new Point(); // COORDINATE OF EAST NEIGHBOR

        // LOOP TO TAKE CARE OF BACKTRACKING IF NO VIABLE MOVES "FORWARD" (I.E. AT END OF LONG HALLWAY)
        do {
                curr_position = pos.getPos();
                x = (int) curr_position.getX();
                y = (int) curr_position.getY();

                // DEBUGGING STATEMENT
                // System.out.println("We're evaluating " + pos.getPos().toString() + " for viable neighbors.");

                up.setLocation(x, y - 1); // NORTH
                down.setLocation(x, y + 1); // SOUTH
                left.setLocation(x - 1, y); // WEST
                right.setLocation(x + 1, y); // EAST

                // FOR EACH POSSIBLE NEIGHBOR, WE CHECK IF IT'S IN BOUNDS,
                        // IF IT'S INFERRED / OBSERVED TO BE BLOCKED,
                        // AND IF IT HAS ALREADY BEEN VISITED
                                // IF IT'S ALREADY BEEN VISITED, THEN WE KNOW THAT
                                // IT'S NOT WORTH EXPLORING THAT AREA AGAIN
                                // THE ONLY EXCEPTION TO THIS WILL BE IF WE NEED TO BACKTRACK
                                // WHICH IS EXPLAINED BELOW
                if (inBounds(up)) {
                        if (!(maze.getCell((int) up.getX(), (int) up.getY()).isBlocked())) {
                                if (!(maze.getCell((int) up.getX(), (int) up.getY()).isVisited())) {
                                        if (!(contains(maze.getCell((int)up.getX(), (int)up.getY()), doneWith))) {
                                                // DEBUGGING STATEMENT
                                                // System.out.println("We've hit a break north condition in the backtracking.");
                                                break;
                                        }
                                }
                        }
                }

                if (inBounds(down)) {
                        if (!(maze.getCell((int) down.getX(), (int) down.getY()).isBlocked())) {
                                if (!(maze.getCell((int) down.getX(), (int) down.getY()).isVisited())) {
                                        if (!(contains(maze.getCell((int)down.getX(), (int)down.getY()), doneWith))) {
                                                // DEBUGGING STATEMENT
                                                // System.out.println("We've hit a break south condition in the backtracking.");
                                                break;
                                        }
                                }
                        }
                }

                if (inBounds(left)) {
                        if (!(maze.getCell((int) left.getX(), (int) left.getY()).isBlocked())) {
                                if (!(maze.getCell((int) left.getX(), (int) left.getY()).isVisited())) {
                                        if (!(contains(maze.getCell((int)left.getX(), (int)left.getY()), doneWith))) {
                                                // DEBUGGING STATEMENT
                                                // System.out.println("We've hit a break west condition in the backtracking.");
                                                break;
                                        }
                                }
                        }
                }

                if (inBounds(right)) {
```

```
                        if (!(maze.getCell((int) right.getX(), (int) right.getY()).isBlocked())) {
                                if (!(maze.getCell((int) right.getX(), (int) right.getY()).isVisited())) {
                                        if (!(contains(maze.getCell((int)right.getX(), (int)right.getY()), doneWith))) {
                                                // DEBUGGING STATEMENT
                                                // System.out.println("We've hit a break east condition in the backtracking.");
                                                break;
                                        }
                                }
                        }
                }

                // IF WE'RE STILL HERE, THEN WE KNOW THAT THERE ARE NO VIABLE NEIGHBORS THAT WE HAVEN'T ALREADY VISITED
                // ARE WE AT THE START NODE RIGHT NOW?
                if (x == 0 && y == 0) { // WE'RE STILL AT THE START NODE HERE, AND THERE'S NOWHERE TO GO
                        return null; // MAZE ISN'T SOLVABLE :(
                }

                // THE ONLY HOPE FOR FINDING A VIABLE PATH AT THIS POINT WOULD BE TO BACKTRACK
                pos = pos.getParent(); // WE'VE BACKTRACKED AND WE WILL START THIS PROCESS AGAIN
                trajectoryLength++; // WE INCLUDE THIS BACKTRACKING IN THE TRAJECTORY LENGTH

        } while (true);


        return pos;
}



public boolean canMove(CellInfo pos, LinkedList<CellInfo> path) { // CHECK IF A CELL CAN ACTUALLY MOVE TO THE NEXT CELL OR NOT

        if (path.peekFirst().isActuallyBlocked()) {
                // System.out.println("Cell " + path.peekFirst().getPos().getX() + ", " + path.peekFirst().getPos().getY() + " is blocked.
                return false;
        }
        return true;
}



public boolean inBounds(Point coor) { // CHECK IF A NEIGHBORING CELL'S COORDINATES ARE IN BOUNDS (DON'T CHECK NON-EXISTENT CELLS)
        if (coor.getX() < 0 || coor.getX() >= cols || coor.getY() < 0 || coor.getY() >= rows) {
                return false;
        }
        return true;
}



public boolean contains(CellInfo newCell, ArrayList<CellInfo> doneWith) { // TO USE FOR THE "CLOSED LIST" IN THE PLANNING PHASE
        for (int i = 0; i < doneWith.size(); i++) {
                if (newCell.getPos().getX() == doneWith.get(i).getPos().getX()
                                && newCell.getPos().getY() == doneWith.get(i).getPos().getY()) {
                        return true;
                }
        }
        return false;
}



public ArrayList<CellInfo> insertCell(CellInfo newCell, ArrayList<CellInfo> toExplore) { // TO USE FOR IMPLEMENTING A PRIORITY QUEUE IN TH

        // FIRST CELL TO BE ADDED TO AN EMPTY LIST
        if (toExplore.isEmpty()) {
                toExplore.add(newCell);
                return toExplore;
        }

        double cell_f = newCell.getF();
        for (int i = 0; i < toExplore.size(); i++) { // WE WANT TO CHECK IF IT'S ALREADY IN THE LIST
                if (newCell.getPos().getX() == toExplore.get(i).getPos().getX() && newCell.getPos().getY() == toExplore.get(i).getPos().ge
                        if (cell_f <= toExplore.get(i).getF()) { // IF THE EXISTING F-VALUE IS HIGHER THAN WE JUST FOUND, WE WANT TO UPDAT
                                toExplore.remove(i);
                        } else { // OTHERWISE, WE JUST RETURN THE LIST AS IS
                                return toExplore;
                        }
                }
        }

        if (toExplore.isEmpty()) {
                toExplore.add(newCell);
                return toExplore;
        }

        // IF OUR CELL HAS A BETTER F-VALUE OR THE CELL ISN'T IN THE LIST ALREADY, THEN WE ADD IT HERE
        for (int i = 0; i < toExplore.size(); i++) {
                if (cell_f < toExplore.get(i).getF()) {
                        toExplore.add(i, newCell);
                        return toExplore;
                } else if (cell_f == toExplore.get(i).getF()) {
                        double cell_g = newCell.getG();
                        if (cell_g <= toExplore.get(i).getG()) {
                                toExplore.add(i, newCell);
                                return toExplore;
                        }
                }
        }
```

```
        // THE CELL WE FOUND HAS THE HIGHEST F-VALUE OF ANY WE FOUND SO FAR
        toExplore.add(toExplore.size() - 1, newCell); // ADDING IT TO THE END OF THE LIST
        return toExplore;

}



public void checkSurroundings(CellInfo pos) { // THIS IS USED TO UPDATE THE KNOWLEDGE BASE
        // ESSENTIALLY, WE JUST OBSERVE A NEIGHBOR'S SURROUNDINGS

        if (pos.getNeighborsUnconfirmed() == 0) { // NOTHING MORE WE CAN OBSERVE HERE
                return;
        }

        Point coor = pos.getPos();
        int x = (int) coor.getX();
        int y = (int) coor.getY();

        int blocked = 0; // HOW MANY MORE NEIGHBORS ARE INFERRED / OBSERVED TO BE BLOCKED?
        int empty = 0; // HOW MANY MORE NEIGHBORS ARE INFERRED / OBSERVED TO BE EMPTY?

        // ALL THE POSSIBLE NEIGHBORS, POSSIBLY REDUCED BY THE INBOUNDS CHECKER
        Point n = new Point(x, y - 1);
        Point nw = new Point(x - 1, y - 1);
        Point w = new Point(x - 1, y);
        Point sw = new Point(x - 1, y + 1);
        Point s = new Point(x, y + 1);
        Point se = new Point(x + 1, y + 1);
        Point e = new Point(x + 1, y);
        Point ne = new Point(x + 1, y - 1);

        ArrayList<Point> neighbors = new ArrayList<Point>();
        neighbors.add(n);
        neighbors.add(nw);
        neighbors.add(w);
        neighbors.add(sw);
        neighbors.add(s);
        neighbors.add(se);
        neighbors.add(e);
        neighbors.add(ne);
        ArrayList<Association.Unit> unconfirmedNeighbors = pos.retrieveUnconfirmedNeighbors();

        // UPDATE OUR KNOWLEDGE BASE FOR JUST THE CELL WE ARE IN
        // THIS IS SEPARATE FROM THE SENSING PHASE
        for (int i = 0; i < neighbors.size(); i++) {
                if (inBounds(neighbors.get(i))) {

                        int neighbor_x = (int) neighbors.get(i).getX();
                        int neighbor_y = (int) neighbors.get(i).getY();

                        if (maze.getCell(neighbor_x, neighbor_y).isBlocked()) {
                                for (int j = 0; j < unconfirmedNeighbors.size(); j++) {
                                        if (neighbor_x == unconfirmedNeighbors.get(j).getCell().getPos().getX() &&
                                                        neighbor_y == unconfirmedNeighbors.get(j).getCell().getPos().getY()) {
                                                blocked++;
                                                pos.removeFromUnconfirmed(maze.getCell(neighbor_x, neighbor_y));
                                        }
                                }
                        } else if (!maze.getCell(neighbor_x, neighbor_y).isUnconfirmed()) {
                                for (int j = 0; j < unconfirmedNeighbors.size(); j++) {
                                        if (neighbor_x == unconfirmedNeighbors.get(j).getCell().getPos().getX() &&
                                                        neighbor_y == unconfirmedNeighbors.get(j).getCell().getPos().getY()) {
                                                empty++;
                                                pos.removeFromUnconfirmed(maze.getCell(neighbor_x, neighbor_y));
                                        }
                                }
                        }
                }
        }
        pos.setNeighborsBlocked(pos.getNeighborsBlocked() + blocked);
        pos.setNeighborsEmpty(pos.getNeighborsEmpty() + empty);
        pos.setNeighborsUnconfirmed(pos.getNeighborsUnconfirmed() - blocked - empty);
        // DEBUGGING STATEMENT
        /*System.out.println("Neighbors blocked: " + pos.getNeighborsBlocked()
                + ". Neighbors empty: " + pos.getNeighborsEmpty()
                + ". Neighbors unconfirmed: " + pos.getNeighborsUnconfirmed()); */
        return;

}

// THIS IS USED TO REMOVE A RECENTLY CONFIRMED CELL FROM OUR FACTS IN THE KNOWLEDGE BASE
public void removeFromAssociation(CellInfo currCell, int i) {

        // CHECK WILL EITHER EQUAL THE POSITION OF THE CELL IN THE ASSOCIATION OR -1 (MEANING NOT FOUND)
        int check = facts.get(i).contains(currCell);

        if (check >= 0) { // NOW WE KNOW WE CAN PHYSICALLY REMOVE IT
                int factor = facts.get(i).allUnknowns.get(check).getFactor();
                facts.get(i).allUnknowns.remove(check);
                if (currCell.isBlocked()) { // IF IT WAS BLOCKED, WE HAVE TO CHANGE THE RHS OF THE ASSOCIATION APPROPRIATELY
                        facts.get(i).totalBlocks -= factor;
                }
        }

        // IF WE'VE REDUCED THIS ASSOCIATION TO NOTHING, THEN WE CAN PITCH IT
        if (facts.get(i).allUnknowns.size() == 0) {
                facts.remove(i);
        }
```

```
        return;
}



// THIS IS THE METHOD THAT POWERS THE ENTIRE ALGORITHM, RUNNING THE AGENT UNTIL GOAL CELL OR IT DETERMINES THERE'S NO PATH TO GOAL
public static char run(int rowNum, int colNum, double prob) {

        Agent4 mazeRunner = new Agent4(); // KEEPS TRACK OF ALL OF OUR DATA AND STRUCTURES

        // READING FROM INPUT
        mazeRunner.rows = rowNum; // THE NUMBER OF ROWS THAT WE WANT IN THE CONSTRUCTED MAZE
        mazeRunner.cols = colNum; // THE NUMBER OF COLUMNS THAT WE WANT IN THE CONSTRUCTED MAZE
        double p = prob; // VALUE BETWEEN 0.0 AND 1.0

        // SET UP MAZE
        mazeRunner.maze = new Maze(mazeRunner.rows, mazeRunner.cols, p);
        boolean badPath = false; // USED FOR DETERMINING WITH WE'VE FOUND A BAD PATH BY INFERENCE

        long begin = System.nanoTime();
        CellInfo start = mazeRunner.maze.getCell(0, 0);
        LinkedList<CellInfo> plannedPath = mazeRunner.plan(start); // STORES OUR BEST PATH THROUGH THE MAZE

        // MAIN LOOP FOR AGENT TO FOLLOW AFTER FIRST PLANNING PHASE
        while (true) {

                // EXTRACT THE NEXT CELL IN THE PLANNED PATH
                CellInfo currCell = plannedPath.poll();

                // DEBUGGING STATEMENT
                // System.out.println("Agent is currently in " + currCell.getPos().getX() + ", " + currCell.getPos().getY());

                // HAVE WE HIT THE GOAL CELL YET?
                if (currCell.getPos().getX() == mazeRunner.cols - 1 && currCell.getPos().getY() == mazeRunner.rows - 1) { // WE'VE HIT THE
                        break;
                }
                // WE HAVEN'T HIT THE GOAL CELL, SO WE CONTINUE ONWARD

                // IF THIS CELL WAS UNCONFIRMED BEFORE, WE NEED TO UPDATE ALL OF OUR FACTS IN THE KNOWLEDGE BASE
                if (currCell.isUnconfirmed()) {
                        for (int i = 0; i < mazeRunner.facts.size(); i++) {
                                mazeRunner.removeFromAssociation(currCell, i);
                        }
                }
                currCell.setEmpty(); // THE CELL WE ARE CURRENTLY IS EMPTY / UNBLOCKED
                if (!currCell.isVisited()) { // IF THE CELL HAS ALREADY BEEN VISITED THEN WE DON'T NEED TO SENSE (SAVING SOME COMPUTATIONA
                        mazeRunner.sense(currCell); // SENSE HOW MANY NEIGHBORS ARE BLOCKED (BUT NOT WHERE THE BLOCKS ARE)
                }
                currCell.setVisited(); // WE HAVE NOW OFFICIALLY VISITED THIS CELL
                mazeRunner.infer(currCell); // INFER WHAT WE CAN ABOUT OUR SURROUNDINGS AND UPDATE KNOWLEDGE BASE


                // NOW WHAT MORE CAN WE INFER? WHAT CAN WE FIGURE OUT ABOUT THE NEXT CELL IN OUR PATH?
                        // WE'RE GONNA LOOK AT THE CURRENT AND PREVIOUS CELL TO CREATE A NEW EQUATION TO PUT INTO OUR KNOWLEDGE BASE
                                // THEY'RE GOING TO HAVE AT LEAST SOME COMMON NEIGHBORS, BUT THE QUESTION IS WHETHER WE CAN FIND
                                // ANYTHING HELPFUL FROM THIS SYNTHESIS OF THE TWO NEIGHBORS' KNOWLEDGE
                        // OBVIOUSLY, IF WE'RE STILL IN THE START CELL, THERE'S NOWHERE TO LOOK BACK TO
                if (currCell.getPos().getX() != 0 || currCell.getPos().getY() != 0) {
                        Association possiblyHelpful = currCell.getAssociationInfo().synthesize(currCell.getParent().getAssociationInfo());

                        if (possiblyHelpful != null) {
                                ArrayList<CellInfo> confirmed = possiblyHelpful.process();
                                if (confirmed != null) {
                                        for (int j = 0; j < confirmed.size(); j++) {
                                                for (int k = 0; k < mazeRunner.facts.size(); k++) {
                                                        mazeRunner.removeFromAssociation(confirmed.get(j), k);
                                                }
                                                mazeRunner.infer(confirmed.get(j));
                                        }
                                } else {
                                        mazeRunner.facts.add(possiblyHelpful);
                                }
                        }
                }

                // WE'RE GOING TO TRY TO LOOK AT THE NEXT CELL IN THE PATH AND SEE IF OUR PATH IS STILL ALRIGHT
                mazeRunner.inferExtended(plannedPath.peekFirst());
                // NOW WE'VE DONE ALL THE INFERRING WE POSSIBLY CAN FOR THE NEXT CELL AHEAD
                        // THE REASON WE DON'T TRY TO MAKE A NEW ASSOCIATION WITH THE NEXT CELL
                        // IS THAT THERE HAS BEEN NOTHING SENSED FROM THAT CELL YET
                        // AND THIS IS A KEY PIECE OF INFORMATION FOR CREATING A NEW ASSOCIATION
                        // SO WE ONLY SYNTHESIZE FROM VISITED, NON-BLOCKED CELLS WHICH HAVE A "SENSED" VALUE


                // HAVE WE HAD AN UPDATE THAT REQUIRES US TO REPLAN?
                // CHECK IF WE'VE HAD AN UPDATE (BLOCK) IN THE PATH THAT REQUIRES US TO REPLAN
                for (int i = 0; i < plannedPath.size(); i++) {
                        CellInfo temp = plannedPath.poll();
                        if (temp.isBlocked()) { // THIS IS CHECKING THE INFERRED / OBSERVED BLOCKS, NOT SNEAKING A PEEK AT THE LEGITIMATE
                                badPath = true;
                                break;
                        }
                        plannedPath.addLast(temp); // CYCLE THIS CELL TO THE BACK
                        // IF ALL CELLS ARE STILL THOUGHT TO BE UNBLOCKED, THEY'LL END UP BACK IN THEIR CORRECT POSITIONS
                }

                // WE'VE FOUND A BLOCK IN OUR PATH WITHOUT ACTUALLY RUNNING INTO IT VIA INFERENCE
```

```
        if (badPath) {

                // DEBUGGING STATEMENT
                // System.out.println("WE'VE DISCOVERED A BAD PATH WITHOUT COLLISION.");

                plannedPath = mazeRunner.plan(currCell);
                if (plannedPath == null) {
                        System.out.println("Maze is unsolvable.\n");
                        // System.out.println(mazeRunner.maze.toString());
                        return 'F';
                }
                badPath = false;
                continue;
        }

        // OUR PLANNED PATH IS STILL OKAY AS FAR AS WE KNOW IF WE'RE HERE
        // ATTEMPT TO EXECUTE EXACTLY ONE CELL MOVEMENT
        mazeRunner.cellsProcessed++;
        if (!mazeRunner.canMove(currCell, plannedPath)) {
                // WE'VE FOUND / HIT A BLOCK
                CellInfo obstruction = plannedPath.peekFirst();
                obstruction.setBlocked();
                obstruction.setVisited();
                mazeRunner.collisions++;

                // DEBUGGING STATEMENT
                // System.out.println("We've hit a block at coordinate " + obstruction.getPos().toString());

                // WE NEED TO UPDATE OUR KNOWLEDGE BASE
                Point n = new Point((int) obstruction.getPos().getX(), (int) obstruction.getPos().getY() - 1);
                Point nw = new Point((int) obstruction.getPos().getX() - 1, (int) obstruction.getPos().getY() - 1);
                Point w = new Point((int) obstruction.getPos().getX() - 1, (int) obstruction.getPos().getY());
                Point sw = new Point((int) obstruction.getPos().getX() - 1, (int) obstruction.getPos().getY() + 1);
                Point s = new Point((int) obstruction.getPos().getX(), (int) obstruction.getPos().getY() + 1);
                Point se = new Point((int) obstruction.getPos().getX() + 1, (int) obstruction.getPos().getY() + 1);
                Point e = new Point((int) obstruction.getPos().getX() + 1, (int) obstruction.getPos().getY());
                Point ne = new Point((int) obstruction.getPos().getX() + 1, (int) obstruction.getPos().getY() - 1);

                ArrayList<Point> neighbors = new ArrayList<Point>();
                neighbors.add(n);
                neighbors.add(nw);
                neighbors.add(w);
                neighbors.add(sw);
                neighbors.add(s);
                neighbors.add(se);
                neighbors.add(e);
                neighbors.add(ne);

                for (int i = 0; i < neighbors.size(); i++) {
                        if (mazeRunner.inBounds(neighbors.get(i))) {
                                mazeRunner.checkSurroundings(mazeRunner.maze.getCell((int) neighbors.get(i).getX(), (int) neighbor
                                mazeRunner.maze.getCell((int) neighbors.get(i).getX(), (int) neighbors.get(i).getY()).removeFromU
                        }
                }
                for (int i = 0; i < mazeRunner.facts.size(); i++) {
                        mazeRunner.removeFromAssociation(obstruction, i);
                }

                mazeRunner.infer(obstruction);
                // WE CAN'T SENSE THOUGH SINCE WE CAN'T OCCUPY THAT CELL

                // AND WE NEED TO REPLAN AS WELL
                plannedPath = mazeRunner.plan(currCell);
                if (plannedPath == null) {
                        System.out.println("Maze is unsolvable.\n");
                        // System.out.println(mazeRunner.maze.toString());
                        return 'F';
                }
                continue;
        }

        // IF WE HAVE SUCCESSFULLY MOVED TO ANOTHER CELL, WE UPDATE THE TRAJECTORY LENGTH
        mazeRunner.trajectoryLength++;


}

// IF WE BREAK FROM THE LOOP (AKA WE'RE HERE AND HAVEN'T RETURNED YET), WE KNOW WE FOUND THE GOAL.
long end = System.nanoTime();
mazeRunner.runtime = end - begin;

CellInfo ptr = mazeRunner.maze.getCell(mazeRunner.cols - 1, mazeRunner.rows - 1);
while (ptr.getPos().getX() != 0 || ptr.getPos().getY() != 0) {

        // DEBUGGING STATEMENT
        // System.out.println("Currently backtracking and at " + ptr.getPos().toString());

        // THE WHOLE PURPOSE OF THE FOLLOWING SEVERAL LINES IS TO MAKE SURE WE'RE COMPUTING THE SHORTEST PATH CORRECTLY
                // WE MAY HAVE SITUATIONS WHERE WE RUN INTO A BLOCK AND SWERVE AROUND IT WITHOUT REALIZING THAT
                // IN THE COMPUTATION OF THE SHORTEST PATH, WE COULD'VE AVOIDED THAT DETOUR
        // THIS JUST LOOKS FOR IMMEDIATE NEIGHBORS FARTHER BACK IN THE PARENT CHAIN
        // AND IF WE HAVE A HIT, THEN WE KNOW WE HAVE AN EVEN SHORTER PATH THAN WE THOUGHT AND WE CHANGE THE POINTER
        // OF THE CURRENT CELL WE'RE AT DURING THE BACKTRACKING
        CellInfo temp = ptr.getParent();
        if (temp.getPos().getX() == 0 && temp.getPos().getY() == 0) {
                ptr.setOnShortestPath(); // FOR PRINTING PURPOSES DURING DEBUGGING
                mazeRunner.shortestPathFound++;
                break;
        }
```

```
                        temp = temp.getParent(); // THIS IS TO INSURE WE ARE PAST THE MOST IMMEDIATE NEIGHBOR OF THE CURRENT CELL
                        // OTHERWISE THIS WOULD BE A FRUITLESS VENTURE

                        int x = (int)ptr.getPos().getX();
                        int y = (int)ptr.getPos().getY();
                        while (temp.getPos().getX() != 0 || temp.getPos().getY() != 0) {
                                int x2 = (int)temp.getPos().getX();
                                int y2 = (int)temp.getPos().getY();

                                if ((Math.abs(x - x2) == 1 && y - y2 == 0) || (Math.abs(y - y2) == 1) && x - x2 == 0) {
                                        // DEBUGGING STATEMENT
                                        // System.out.println("We've determined that " + temp.getPos().toString() + " is a closer neighbor.");
                                        ptr.setParent(temp);
                                        break;
                                } else {
                                        temp = temp.getParent();
                                }
                        }

                        if (temp.getPos().getX() == 0 && temp.getPos().getY() == 0) {
                                if ((x == 1 && y == 0) || (y == 1) && x == 0) {
                                        ptr.setParent(temp);
                                }
                        }

                        ptr.setOnShortestPath(); // FOR PRINTING PURPOSES DURING DEBUGGING
                        mazeRunner.shortestPathFound++;
                        ptr = ptr.getParent(); // FOLLOW THE PARENT CHAIN BACK UP UNTIL THE START CELL
                }

                System.out.println("Path Found!");
                // System.out.println(mazeRunner.maze.toString());
                mazeRunner.printStats();

                return 'S';
        }

        // DRIVER METHOD
        public static void main(String args[]) {

                // ROWS, COLUMNS, DENSITY OF BLOCKED CELLS AND THE NUMBER OF SUCCESSFUL PATHS FOUND
                        // ALL READ IN AS COMMAND LINE ARGUMENTS
                int rowNum = Integer.parseInt(args[0]);
                int colNum = Integer.parseInt(args[1]);
                double prob = Double.parseDouble(args[2]);
                int successfulTrials = Integer.parseInt(args[3]);

                while (successfulTrials > 0) {
                        char result = run(rowNum, colNum, prob);
                        if (result == 'S') {
                                successfulTrials--;
                        }
                }

                return;

        }
}
```