CS520 Project 3: Probability in GridWorld

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

November 17, 2021

Submitter: Daniel Ying

Honor Code:

I abide to the rules laid in the Project 3: Probability in the Gridworld 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 3.

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

Workload:

Daniel Ying: Worked on compiling the Latex report. Collected data and graphing agents 7 and 8. Contributed in devising strategies for Agent 8.

Zachary Tarman: Worked on implementing Agents 6, 7, 8, and 9. Contributed in authoring the report and devising strategies for Agents 8 and 9.

Pravin Kumaar: Contributed in devising strategies for agent 9.

Together: Brainstormed the Algorithm of Agent 8. Discussed calculation of probability in each situation. Discussed problems in the assignment and code.

Problem 1

Question 1: (2 points) Prior to any interaction with the environment, what is the probability of the target being in a given cell?

ANSWER:

Based on the free-space assumption that has preceded exploration of the mazes in project 1-3, all cells are assumed to be unblocked in the initial state (or at the very least, the agent has no reason to think any cell is more likely to be blocked over another). So, the agent starts with the number of cells assumed to be unblocked equal to the total number of cells in the maze.

Given this, the agent assumes that each cell is equally likely to contain the target. We can calculate this with the following:

P(ij containing target) = 1 / (of unblocked cells)

= 1 / (of total cells) **for all cells i,j**

Problem 2

Question 2: (10 points) Let Pi,j (t) be the probability that cell (i, j) contains the target, given the observations collected up to time t. At time t + 1, suppose you learn new information about cell (x, y). Depending on what information you learn, the probability for each cell needs to be updated. What should the new Pi,j (t + 1) be for each cell (i, j) under the following circumstances:

– At time t + 1 you attempt to enter (x, y) and find it is blocked?

– At time t + 1 you attempt to enter (x, y), find it unblocked, and also learn its terrain type?

– At time t + 1 you examine cell (x, y) of terrain type flat, and fail to find the target?

– At time t + 1 you examine cell (x, y) of terrain type hilly, and fail to find the target?

– At time t + 1 you examine cell (x, y) of terrain type forest, and fail to find the target?

– At time t + 1 you examine cell (x, y) and find the target?

ANSWER:

For all of the following equations, the LHS signifies what will become the probability that the target is contained within the associated cell at time t+1, and everything on the RHS is the current belief state of probabilities that cells contain the target at time t.

Each scenario will also show separate update equations for what we'll call the "event" cell xy (i.e. the cell where the probability changes due to a discovery associated with that cell), and every other cell ij in the maze that is not equal to xy.

a) Update when attempting to enter a cell and finding it blocked

xy update:

P(target is in xy | xy is blocked) = 0

ij update:

P(target is in ij | xy is blocked)

= P(target is in ij | target is not in xy)

// xy being blocked signifies that the cell doesn't contain the target

= P(target in ij) * P(target is not in xy | target in ij) / P(target is not in xy)

// P(target is not in xy | target in ij) is equal to 1 for fairly self-explanatory reasons

= P(target in ij) * 1 / P(target is not in xy)

= P(target in ij) / (1 – P(target is in xy))

b) Update when entering a previously unvisited cell and discovering its terrain type

As discussed in the announcement posts by the professor, in real world situations, this would change the belief state, but for the purposes of this project, we are assuming that there is no change to the belief state in this situation.

c) Update when examining a cell with flat terrain and failing to find the target xy update:

P(target in xy | failed examination on xy of terrain type flat)

= P(target in xy) * P(failed exam of flat xy | target in xy) / P(failed exam of flat xy)

// The false negative rate for flat terrain is 0.2

= P(target in xy) * 0.2 / (1 – P(successful exam of flat xy))

// P(successful exam of flat xy) = P(target in , successful exam of flat xy) via marginalization

// P(target in , successful exam of flat xy)

// = P(target in) * P(successful exam of flat xy | target in)

// = (some probability)*0 + (some probability*0) + … + P(Target in xy) * 0.8 + …

// = P(Target in xy) * 0.8

// The 0.8 multiplier comes from the probability of a true positive on flat terrain

= 0.2*P(target in xy) / (1 – (0.8*P(target in xy)))

ij update:

P(target in ij | failed examination of xy of terrain type flat)

= P(target in ij) * P(failed exam of flat xy | target in ij) / P(failed exam of flat xy)

// Denominator is computed in the exact same way as in the xy update

// Also, P(failed exam of flat xy | target in ij) is equal to 1 because a failure is bound to happen

= P(target in ij) * 1 / (1 – (0.8*P(target in xy)))

= P(target in ij) / (1 – (0.8*P(target in xy)))

d) Update when examining a cell with hilly terrain and failing to find the target

The derivations for the xy update and the ij update are identical to the corresponding updates completed in part c. The only difference in the final result is that the false negative rate for hilly terrain is 0.5, and the true positive rate for hilly terrain is 0.5.

xy update:

P(target in xy | failed examination on xy of terrain type flat)

= 0.5*P(target in xy) / (1 – (0.5*P(target in xy)))

ij update:

P(target in ij | failed examination of xy of terrain type flat)

= P(target in ij) / (1 – (0.5*P(target in xy)))

e) Update when examining a cell with forest terrain and failing to find the target

The derivations for the xy update and the ij update are identical to the corresponding updates completed in part c. The only difference in the final result is that the false negative rate for forest terrain is 0.8, and the true positive rate for forest terrain is 0.2.

xy update:

P(target in xy | failed examination on xy of terrain type flat)

= 0.8*P(target in xy) / (1 − (0.2*P(target in xy)))

ij update:

P(target in ij | failed examination of xy of terrain type flat)

= P(target in ij) / (1 − (0.2*P(target in xy)))

f) Update when target is found

xy update:

P(target in xy | successful exam of xy) = 1

ij update:

P(target in ij | successful exam of xy) = 0

Problem 3

Question 3: (8 points) At time t, with probability Pi,j (t) of cell (i, j) containing the target, what is the probability of finding the target in cell (x, y):

– If (x, y) is hilly?

– If (x, y) is flat?

– If (x, y) is forest?

– If (x, y) has never been visited?

ANSWER:

All of the following equations are based on the corresponding true positive rates.

We are assessing:

P(finding target in ij) = P(successful exam of ij, target in ij)

= P(target in ij) * P(successful exam of ij | target in ij).

P(successful exam of ij | target in ij) is a known probability that is equal to 1 − false negative rate for given terrain type.

a) Probability of finding the target in a cell that is hilly:

P(finding target in ij) = 0.5*P(target in ij)

b) Probability of finding the target in a cell that is flat:

P(finding target in ij) = 0.8*P(target in ij)

c) Probability of finding the target in a cell that is forest:

P(finding target in ij) = 0.2*P(target in ij)

d) Probability of finding the target in a cell that has never been visited:

P(finding target in ij) = [(1/3)*0.8*P(target in ij) + (1/3)*0.5*P(target in ij) + (1/3)*0.2*P(target in ij)]

= (1/3)*P(target in ij) * (0.8 + 0.5 + 0.2) = (1/3)*P(target in ij) * (1.5)

= 0.5*P(target in ij)

Problem 4

Question 4: (30 points) Implement Agent 6 and 7. For both agents, repeatedly run each agent on a variety of randomly generated boards (at constant dimension) to estimate the number of actions (movement + examinations) each agent needs on average to find the target. You will need to collect enough data to determine which of these agents is superior. Do you notice anything about the movement/examinations distribution for each agent? Note, boards where the target is unreachable from the initial agent position should be discarded.

ANSWER:

After testing over 50 trials with a maze of dimensions 101x101, Agent 6 had average values of trajectory length of 141115.9, examinations of 15597.02, and total cost of 156712.92. Under the same conditions, Agent 7, we have average trajectory length of 79668.1, examinations of 12186, and total costs of 91854.08. We can clearly see that Agent 7 performed more optimally than Agent 6. In fact, in terms of the total cost, Agent 7 performed approximately 41.4 percent better than Agent 6.

The main difference between Agents 6 and 7 is the strategy in which the agents are choosing the next cell to travel towards and examine. For Agent 6, the agent is identifying the cell with the highest probability of containing the target based on what the Agent 6 have observed. On the other hand, Agent 7 is identifying the cell with the highest probability of successfully finding the target given what Agent 7 have observed.

Since the goal of the agents is to actually find the target, Agent 7 has an edge in choosing the most optimal places to examine and subsequently "rule out" versus Agent 6. Thus, Agent 7 was expected to yield better results than Agent 6 and the data supports this proposition that Agent 7 is indeed the better agent at finding the target.

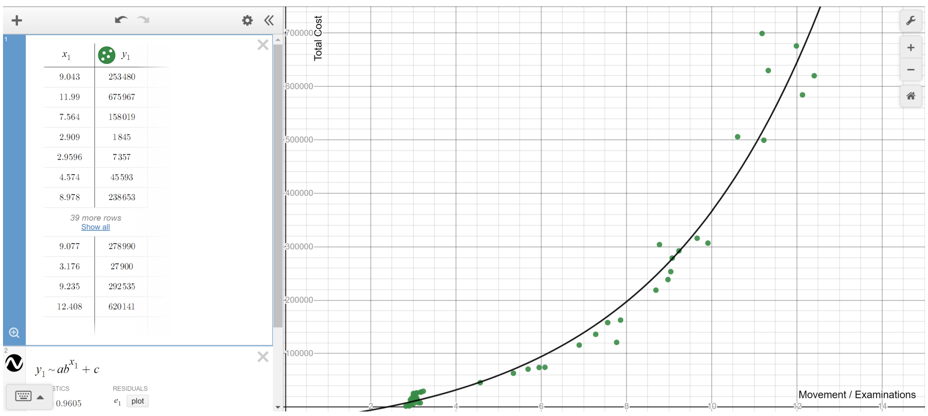

Figure 1: Agent 6 Trajectory Length over Examinations vs. Total Cost.

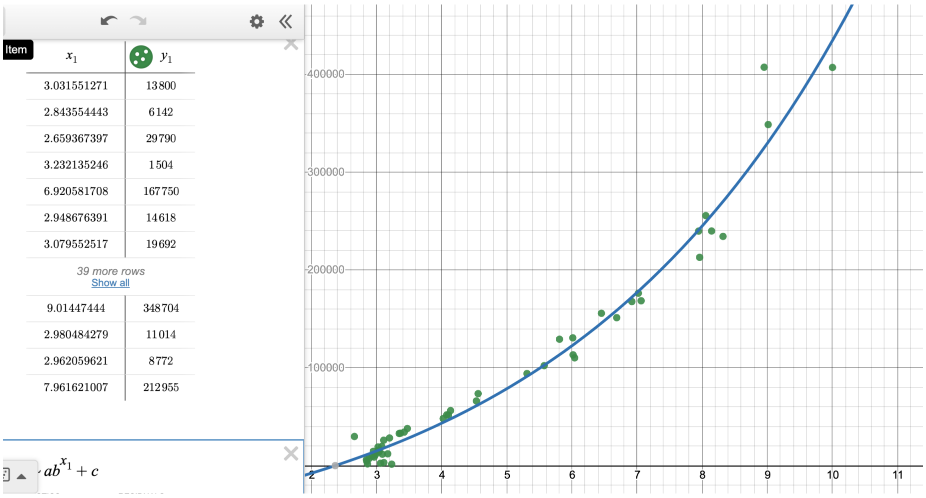

Figure 2: Agent 7 Trajectory Length over Examinations vs. Total Cost.

Problem 5

Question 5: (20 points) Describe your algorithm, be explicit as to what decisions it is making, how, and why. How does the belief state (Pi,j (t)) enter into the decision making? Do you need to calculate anything new that you didn't already have available?

ANSWER:

Question 5 Brainstorming Strategies for Agent 8:

Before approaching how we wanted to design Agent 8, we noticed that there were situations for both Agent 6 and 7 where the following would happen: the agent would examine a cell and have a negative result, the entire belief state is updated, and then it was determined that the new cell with the highest probability of containing the target (or the cell where it was most likely to find the target for Agent 7) was all the way on the other side of the board! More or less. So, we would see the trajectory length skyrocket for some trials, leading to a fairly high cost of operations that the agent must execute.

Our first thought for Agent 8 was something the professor had also expressed: "if you determine that there is a cell with a high probability that is far away from you that you should examine, as you are moving towards it, you may find cells of lower probability (but still good). Should you examine them prior to getting to the maximum probability cell?" Essentially, why don't we make some examinations along the way to potentially save a costly trip across the entire grid-world?

Naturally extended from this line of thinking, we asked: what if we were able to choose a new cell to travel towards from the very start (of planning a new path), considering not only cells with high probability of successfully locating the target but also the cost for the agent to get there and examine? And from there, the mantra of Agent 8 was cost-effective destination planning. To formalize this approach, we had to come up with an effective way of representing this idea of the most "cost-effective" destination for our agent's next planning phase. Of course, the probability still factors into this determination, but we also want to consider the cost (i.e. the trajectory length to travel to the destination cell as well as the examination once we arrive). The higher the probability, the more the agent wants to go there, but the higher the cost, the less the agent will want to go there. So, we created a relationship for a cell's "c" value being the following:

Probability of finding the target in this cell / ((planned path length from current position to there) + 1)

The +1 is for the examination that occurs at the end of the path which has a cost of 1 based on the metrics provided by the project description. With this relationship, we could determine the cell with the best probability to cost ratio, and this would be the next cell that we would want to travel to.

Algorithm:

A new destination cell is determined anytime there is some update to the belief system. This is triggered by either an examination, finding a cell to be blocked, or finding that a cell is unreachable via A*. (It's worth mentioning that given the Agent 7 behavior, updating the probability of finding the cell in a previously unvisited cell also would factor into this. However, in our current structure, if the agent finds that the cell it's currently in actually just became the most optimal place to find the target, the agent simply sets the new destination to be where it is at (very cost-effective planning), and it will examine anyway, which is covered by the examination trigger listed above).

The "event cell" (i.e. the cell that is examined, the cell that is blocked, or the cell that is found to be unreachable) has its probability updated, and then every other cell is updated as well. While we're updating all the probabilities of the cells in the grid-world, we determine the cell with the best metric to be our next destination. For Agent 6, it was P(target in ij). For Agent 7, it was P(finding target in ij). And now for Agent 8, it will be determined via the "c" value described above. Consider the following algorithm:

- Some event triggers an update of the knowledge base

- Event cell's probability is updated

- Event cell's c-value = event cell's updated probability / 1

// If the event cell is the current cell the agent occupies, that means the triggering event was an examination, and so the denominator, which represents the cost of the next plan to travel and examine, is essentially 0 steps to get from current position to event cell + 1 examination

// If the event cell is a blocked cell or if it is a cell that is unreachable, then the updated probability will be 0, and the corresponding c-value will be 0.

- Current destination = event cell

- Current max c-value = event cell's c-value

// Based on our belief system update so far, this is the best place to go and examine, but except in rare circumstances, it will almost surely change

- For all remaining cells, "temp", to be updated that aren't the event cell

o Temp's probability is updated

o Temp's c-value = temp's updated probability / ((Manhattan distance from current cell occupied by agent to temp) + 1)

// The original idea was to attempt planning using A* so we know the actual path length to get there, but this turned out to be way too costly to the runtime (with potentially upwards of 200 A* runs after every update to the belief state, even after filtering out unpromising cells). This just wasn't feasible. In the same way that it's used as a heuristic anyway, the Manhattan distance is used to save having to plan a path for many cells in the grid-world which would be quite costly. This is still an appropriate and helpful metric to gauge cost-effectiveness because if the Manhattan distance is a lower bound for the path length and a cell's c-value is already lower than the "current max c-value", then the actual path length can only be equal to or longer, and we know that this cell is definitely not going to be our next destination.

-If temp's c-value > current max c-value

o Current destination = temp

o Current max c-value = temp's c-value

- Using the destination determined above, plan a new path to get there

- Normal probability agent behavior, etc.

So, by the end of the update to the belief system, agent 8 will have determined the most cost-effective destination in the grid-world to shoot for next, factoring in not only the probability of finding the target there but also the cost of travelling there and examining it. This will cut down on the overall cost endured by the agent while searching for the target.

Problem 6

Question 6: (25 points) Implement Agent 8, run it sufficiently many times to give a valid comparison to Agents 6 and 7, and verify that Agent 8 is superior.

ANSWER:

Agent 8 produced an average of 42815.54, examinations of 25439.8, and total cost of 62615.34.

Surprisingly, Agent 8's graph has a decreasing slope. We believe this shows that the lower the difference between the trajectory length and examinations, the higher the overall cost for Agent 8.

Thus, in simpler terms, if the agent 8 is able to find the target quickly, then there is a high variance between the trajectory length/ examinations.

However, as the total cost for agent 8 increases, the agent is able to get the trajectory length with in double the amount (2x) of examinations. This indicates that agent 8 is being 'smart' about deciding its next destination to examine.

Because of this difference of Agent 8 to the other two agents 6 and 7 (as described in the previous problem), we clearly see that this has lasting effects as total cost increase to infinity, making Agent 8 the better agent than agent 6 and 7.

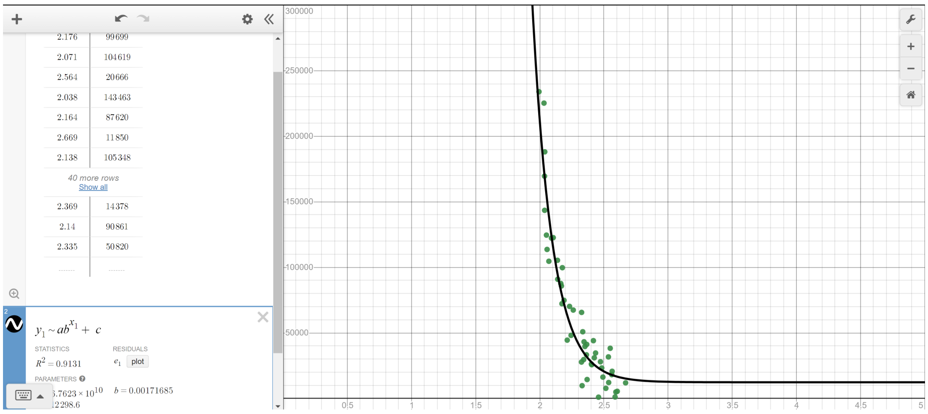

Figure 3: Agent 8 Trajectory Length over Examinations vs. Total Cost.

Problem 7

Question 7: (5 points) How could you improve Agent 8 even further? Be explicit as to what you could do, how, and what you would need.

ANSWER:

We mentioned in the algorithm portion of Question 5 that when calculating our "c-value" (our new metric for judging which cell to plan to go to next, which is equal to the probability of finding the target at the cell / ((the Manhattan distance from the agent's current position to this cell) + 1), we had initially planned to use A* to try to plan paths so that the c-value could be determined with planned path length in the denominator as opposed to the Manhattan distance. The reason for this is that in specific cases, the Manhattan distance can be misleading (i.e., the agent must get around a very long wall despite only being 2 steps away from the destination cell).

However, we determined that there was not a feasible way to have this planning happen when considering a new destination, even when filtered and optimized so we didn't call A* for every cell in the grid, because it drove up the runtime and there would be a great deal of extra computations that the program would have to execute.

In an ideal world though, to avoid situations where the Manhattan distance misleads our "c-value" calculation, we would have some system to be able to determine the actual distance very quickly between the current position and a potential destination. This would maybe result from extra processing power, divided workload via different threads/processes, a data structure to store the best-known distance from a cell to every other cell in the maze, etc.

Obviously, not all these things are rapidly available or feasible for the average college student working off a 6-year-old laptop, but if we wanted to truly optimize Agent 8, being able to know the actual distances between a current position and a potential destination cell when updating the belief state could result in even more efficient costs.

Problem 8

Bonus: (30 points) For Agent 9, we let the target move each time step, but only to one of its immediate neighbors (uniformly at random among unblocked neighbors). The Agent can additionally sense in each cell whether the target is in one of its 8 immediate neighbors, but not which one. Build an Agent that adapts to this moving target and extra partial sensing. How does it decide where to go next, and what to do? How is the belief state updated? Implement, and generate enough data for a good comparison.

ANSWER:

Design:

The general design for adapting to the moving target involves keeping two belief states: one for time t and one for time t+1. We refer to these as the current and imminent belief states respectively. For each movement into a new cell, Agent 9 examines the cell it's currently in (no false negatives this time around per Aravind's suggestion in the Discord server) and sense around in the adjacent 8 neighbors to see if the target is in one of the cells. There are a few scenarios that could unfold:

- The target is found in the current cell, and we terminate the program with success.
- The target is sensed in a neighboring cell and we update the current belief state accordingly.
- The target isn't sensed in a neighboring cell, and we update the current belief state accordingly.

For the first scenario, it's simple what happens next.

For the second, we know for certain that the target is in one of the 8 surrounding cells and that the target is not in the current cell nor any other non-neighboring cell in the grid-world. With this in mind, we can update the current and every non-neighboring cell to have a current probability of 0. What happens with the neighboring cells? The simple but misguided answer would be to assign them all a current probability of 0.125. Intuitively, it seems that it could be right, but doing this disregards the prior probabilities that the target was in each of the cells (i.e., if there was a neighboring cell that definitely didn't contain the target based on our prior path and sensing, why would be lend it more belief?). With this in mind, we simply update the cell's current probability = cell's current probability / the neighbors' collective probability. This is also derived from Bayes' theorem, where the denominator represents the probability that the target is in one of the neighboring cells, which is simply the addition of all their probabilities together (like a giant OR statement).

For the third scenario, the inverse happens. Now, all of the neighboring cells as well as the current cell are set to a current probability of 0, and we update every other cell. Every other cell's current probability = that cell's current probability / (1 − the neighboring cells' and current cell's current probabilities). Since we're in the reverse world, where the scenario in the denominator of Bayes' equation represents the probability that the cell is not in one of the 9 cells that the agent can immediately see, the denominator is adjusted accordingly.

Now that we've updated the current belief state, for the agent to decide where to go next, we have to update the imminent belief state. For this, we recognize that the agent and target can only move in 4 directions (the cardinal directions), and the target has an equal chance of travelling in any given direction. With this in mind, we know that a cell's imminent probability can be represented by the following:

for all N cardinal neighbors, Summation (P(target is in n currently)) / (n's viable neighbors) .

By n's viable neighbors, we recognize that not every cell has 4 options to move to (i.e., a corner, an edge, or adjacent to a blocked cell), so we have to weight the distribution of the probability that the target is currently in that cell accordingly to represent the chance it travels to any of its neighbors.

There are scenarios for finding that a cell is unreachable (either a blocked cell or the agent is blocked from getting there) that's very similar to the prior agents.

The agent decides where to go next in a very similar way to the other agents as well, but with a twist. The agent looks for the cell with the highest imminent probability of containing the target, seeing as when it moves, those imminent probabilities will now be current, so those drive the decision-making process for the agent.

Last thing for the design discussion is our decision to have Agent 9 examine at every step. Yes, examinations are costly, but what would be more costly is missing the target and having to chase it around for many more iterations. Given that there is a 0 percent chance the agent will miss the target if it's in the same cell, we decided that it benefitted the agent more to examine at every step.

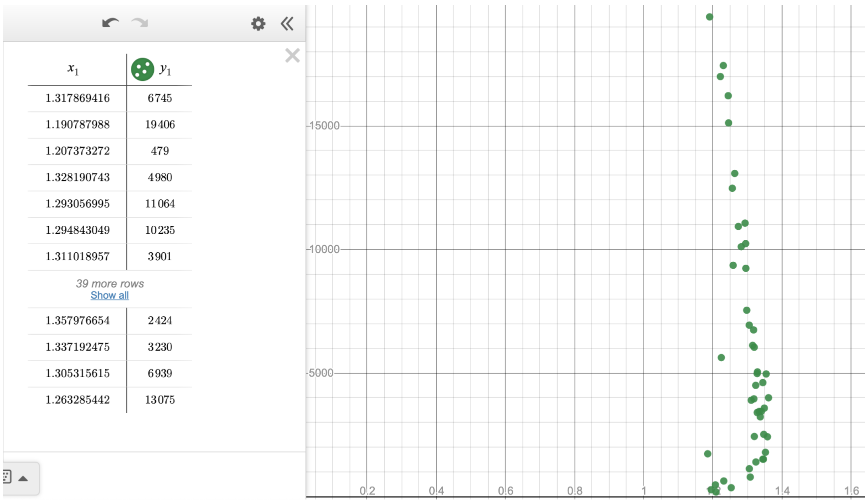

Figure 4: Agent 9 Trajectory Length over Examinations vs. Total Cost.

Looking at the graph, we can see that there is no clear relationship of between the trajectory length/examination vs. total cost. We believe this is due to the fact that we examined for the target at every stop, and so, the relationship of trajectory length/examinations is almost always the same and close to 1. Thus this is why agent 9's results do not look like the results of agents 6, 7, and 8.

After evaluating Agent 9 in the same way as the other agents, the average trajectory length was 3927.64, the average number of examinations was 3150.08 and the average total cost was 7078.72. This shows that the performance for agent 9 was drastically better than those of agents 6, 7, and 8. However, due to the added partial sensing abilities, this is almost like comparing apples and oranges. Nevertheless, the findings are interesting and go to show that implementing multiple strategies that work together can often lead to the most optimal result.

Problem 9

APPENDIX (includes code for Agents 6, 7, 8, and 9)

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

/**
 *
 * @author Zachary Tarman
 * Handles Agent 6 responsibilities in accordance with the descriptions
 * associated with Project 3 of CS520 Fall 2021.
 *
 */
public class Agent6 {

        /**
         * The actual maze object
         */
        public Maze maze;
        /**
         * The row dimension of the maze
         */
        public int rows;
        /**
         * The column dimension of the maze
         */
        public int cols;
        /**
         * The probability held by the cell with the highest probability in the maze
         */
        public double highestProb;
        /**
         * The cell in which the highest probability is held
         */
        public CellInfo cellOfHighestProb;

        /**
         * The number of blocks the agent physically hits
         */
        public int collisions = 0;
        /**
         * The number of cells that we examine (assessing if we can find the target in the given cell)
         */
    public int examinations = 0;
    /**
     * The trajectory length of the agent (includes collisions within this metric)
     */
        public int trajectoryLength = 0;
        /**
         * The runtime of the program to find a path to the goal
         */
    public long runtime = 0;


    /**
     * Prints the stats that might be useful in data collection for Project 3.
     * Cost is total effort exercised by the agent.
     * @see                     Agent6#trajectoryLength
     * @see                     Agent6#examinations
     */
    public void printStats() {
        System.out.println("Statistics for Maze Solution");
        System.out.println("Trajectory Length: " + trajectoryLength);
        System.out.println("Collisions: " + collisions);
        System.out.println("Examinations: " + examinations);
        int cost = trajectoryLength + examinations;
        System.out.println("Total agent cost: " + cost);
        System.out.println("Runtime: " + runtime);
        System.out.println();
        return;
    }



        /**
         * This method plans a route from the agent's current position to the given destination.
         * Think of it as a single iteration of A* without the agent physically moving.
         * It uses the knowledge it has of the explored gridworld and otherwise uses the freespace assumption.
         * @param start                 The start cell
         * @param dest                  The destination cell
         * @return                              The planned path from start to finish
         */
        public LinkedList<CellInfo> plan(CellInfo start, CellInfo dest) {

                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>(); // CELLS THAT HAVE ALREADY BEEN "EXPANDED"

                // System.out.println("Starting at " + start.getPos().getX() + "," + start.getPos().getY());
                // System.out.println("Destination at " + dest.getPos().getX() + "," + dest.getPos().getY());

                CellInfo curr = start; // PTR TO THE CURRENT CELL WE'RE EVALUATING TO MOVE ON FROM IN OUR PLAN
                curr.setG(0); // SINCE THIS IS THE NEW STARTING POINT, WE SET THE G-VALUE TO 0
```

```
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); // AND THIS IS THE FIRST CELL WE'RE GOING TO "EXPAND"

// BEGIN LOOP UNTIL PLANNING REACHES DESTINATION CELL
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 PROBABLY REDUNDANT, BUT HERE JUST
                // 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 DESTINATION??
        // IF SO, LET'S TRACE BACK TO OUR STARTING POSITION
        if (x == (int)dest.getPos().getX() && y == (int)dest.getPos().getY()) {
                CellInfo goal = maze.getCell(x, y);
                CellInfo ptr = goal;

                // LOOP BACK THROUGH, FOLLOWING THE PARENT CHAIN BACK TO THE START
                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;
        }

        // IF WE DIDN'T REACH THE DESTINATION, WE HAVE TO CHECK FOR VIABLE NEIGHBORS TO CONSIDER
        // 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
        // THE CHECKS ESSENTIALLY CONSIST OF THE FOLLOWING
                // IS THE CELL OUT OF BOUNDS? IF SO, DON'T ADD
                // IF NOT, HAVE WE VISITED THE CELL, AND IF WE HAVE, IS IT BLOCKED? IF BOTH ARE TRUE, DON'T ADD
                        // (THE AGENT ONLY KNOWS IT'S BLOCKED IF IT'S VISITED IT ALREADY)
                // IF WE'VE ALREADY ASSESSED THIS CELL WITHIN OUR PLANNING, THEN DON'T ADD IT TO THE LIST OF THINGS TO EXPLORE

        // CHECKS FOR NORTHBOUND NEIGHBOR
        if (!inBounds(up)) {
                addUp = false;
                // System.out.println("North is not in bounds.");
        } else if (maze.getCell((int) up.getX(), (int) up.getY()).isVisited() &&
                        maze.getCell((int)(up.getX()),(int) up.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THIS CE
                addUp = false;
                // System.out.println("North is blocked.");
        } else if (contains(maze.getCell((int) up.getX(), (int) up.getY()), doneWith)) {
                addUp = false;
        }

        // CHECKS FOR SOUTHBOUND NEIGHBOR
        if (!inBounds(down)) {
                addDown = false;
                // System.out.println("South is not in bounds.");
        } else if (maze.getCell((int) down.getX(), (int) down.getY()).isVisited() &&
                        maze.getCell((int)(down.getX()),(int) down.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THI
                addDown = false;
                // System.out.println("South is blocked.");
        } else if (contains(maze.getCell((int) down.getX(), (int) down.getY()), doneWith)) {
                addDown = false;
        }

        // CHECKS FOR WESTBOUND NEIGHBOR
        if (!inBounds(left)) {
                addLeft = false;
                // System.out.println("West is not in bounds.");
```

```java
		} else if (maze.getCell((int) left.getX(), (int) left.getY()).isVisited() &&
				maze.getCell((int)(left.getX()),(int) left.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THI
			addLeft = false;
			// System.out.println("West is blocked.");
		} else if (contains(maze.getCell((int) left.getX(), (int) left.getY()), doneWith)) {
			addLeft = false;
		}

		// CHECKS FOR EASTBOUND NEIGHBOR
		if (!inBounds(right)) {
			addRight = false;
			// System.out.println("East is not in bounds.");
		} else if (maze.getCell((int) right.getX(), (int) right.getY()).isVisited() &&
				maze.getCell((int)(right.getX()),(int) right.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT T
			addRight = false;
			// System.out.println("East is blocked.");
		} 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()); */
		}

		/* System.out.print("Cells to be explored: ");
		for (CellInfo ptr: toExplore) {
			System.out.print(ptr.getPos().toString() + "; ");
		}
		System.out.println(); */

	}

	return null; // TARGET IS UNREACHABLE ):
	// LOOK TO RUN METHOD TO SEE WHAT IS DONE WHEN THE PLANNED DESTINATION IS UNREACHABLE
}


/**
 * Used to examine a given cell to see if we can find the target. Updates belief state appropriately.
 * Based on the cell's terrain type, we may not actually see the target even if the agent is standing on it.
 * @param pos 			The cell to be examined
 * @return 				Whether the target has been found
 * @see 				Maze#isTarget(Point)
 * @see 				Agent6#updateRemainingBeliefState(CellInfo, CellInfo, double, double, boolean)
 */
public boolean examine(CellInfo pos) {

	// System.out.println("Agent currently examining " + pos.getPos().getX() + ", " + pos.getPos().getY());

	this.examinations++;

	int terrainType = pos.getTerrain().value; // THIS WILL SIGNAL TO US WHAT TERRAIN THIS CELL IS
		// TERRAIN DETERMINES HOW LIKELY WE WILL BE TO SENSE THE TARGET

	double oldProb = pos.getProb(); // CONTAINS THE OLD PROBABILITY OF THE CELL WE'RE EXAMINING
	double newProb; // WILL CONTAIN THE NEW PROBABILITY OF THE TARGET BEING CONTAINED IN THIS CELL (IF NEEDED)
	double rand = Math.random(); // A RANDOMLY GENERATED VALUE THAT WILL DETERMINE IF WE FOUND THE TARGET OR NOT
```

```
		// System.out.println("This cell's current probability of containing the target is " + oldProb);

		/*
		 * ALL OF THE EQUATIONS IN THE INNER ELSE CLAUSES WERE DERIVED USING
		 * BAYES' THEOREM FOR THE PROBABILITY OF FINDING THE TARGET IN THE CURRENT CELL
		 * GIVEN A FAILED EXAMINATION IN SOME TYPE OF TERRAIN (THE IF STATEMENTS TAKE
		 * CARE OF THE DIFFERENT SCENARIOS WE MIGHT ENCOUNTER)
		 *				SEE THE REPORT WRITE-UP TO SEE HOW WE DERIVED THESE UPDATES TO THE LIKELIHOODS
		 */
		if (terrainType == 0) { // TERRAIN IS FLAT

			// System.out.println("We're on flat terrain.");
			if (maze.isTarget(pos.getPos()) && rand <= 0.8) {
				// WE'VE FOUND THE TARGET
				return true;
			} else {
				// CALCULATING THE NEW LIKELIHOOD THAT WE FIND THE TARGET HERE AFTER FAILED EXAMINATION
				newProb = (0.2 * oldProb) / (1 - (0.8 * oldProb));
			}

		} else if (terrainType == 1) { // TERRAIN IS HILLY

			// System.out.println("We're on hilly terrain.");
			if (maze.isTarget(pos.getPos()) && rand <= 0.5) {
				// WE'VE FOUND THE TARGET
				return true;
			} else {
				// CALCULATING THE NEW LIKELIHOOD THAT WE FIND THE TARGET HERE AFTER FAILED EXAMINATION
				newProb = (0.5 * oldProb) / (1 - (0.5 * oldProb));
			}

		} else { // TERRAIN IS FOREST-Y

			// System.out.println("We're on forest terrain.");
			if (maze.isTarget(pos.getPos()) && rand <= 0.2) {
				// WE'VE FOUND THE TARGET
				return true;
			} else {
				// CALCULATING THE NEW LIKELIHOOD THAT WE FIND THE TARGET HERE AFTER FAILED EXAMINATION
				newProb = (0.8 * oldProb) / (1 - (0.2 * oldProb));
			}
		}

		// UPDATE THE BELIEF SYSTEM
		pos.updateProb(newProb);
		highestProb = newProb;
		// System.out.println("The updated probability of this cell is " + newProb);
		// System.out.println("The currX and currY variables equal " + currX + " & " + currY);
		updateRemainingBeliefState(pos, null, oldProb, 0, true); // SEE BELOW HELPER METHOD

		return false; // WE DIDN'T FIND THE TARGET
	}



	/**
	 * Used to update the remainder of the belief state given some event.
	 * An examination, a collision, or an unreachable cell can trigger this.
	 * A collision means that the blocked cell is unreachable, so the collision cell would be entered into the unreachable parameter.
	 *
	 * @param pos				The cell of the current position of the agent
	 * @param unreachable			The unreachable cell, if this is not an examination
	 * @param oldProb				The belief in the examined / unreachable cell containing the target at time t (the non-upd
	 * @param currManhattan			The current manhattan distance from the current cell to the (now former) cell with the highest pro
	 * @param examination			Signals whether we're looking at an examination or an unreachable situation
	 */
	public void updateRemainingBeliefState(CellInfo pos, CellInfo unreachable, double oldProb, double currManhattan, boolean examination) {

		int terrainType; // CONTAINS THE TERRAIN TYPE OF THE CELL WITH THE UPDATED PROBABILITY
		Point updatePosition; // THE POSITION OF THE CELL WITH THE UPDATED PROBABILITY
		if (examination) {
			terrainType = pos.getTerrain().value; // THE TERRAIN OF THE CURRENT CELL (WHICH WE JUST EXAMINED)
			updatePosition = pos.getPos();
		} else {
			terrainType = 3; // NOT NECESSARILY APPLICABLE, BUT THE CELL IS "VIRUTALLY" BLOCKED SINCE WE CAN'T OCCUPY IT
			updatePosition = unreachable.getPos();
		}

		// LOOP THROUGH ALL CELLS TO UPDATE ALL OF THEIR PROBABILITIES
		for (int i = 0; i < cols; i++) {
			for (int j = 0; j < rows; j++) {

				// System.out.println("i and j equal " + i + " & " + j);
				if (i == (int) updatePosition.getX() && j == (int) updatePosition.getY()) { // WE'VE ALREADY UPDATED THIS CELL ABO
					continue;
				}

				CellInfo temp = maze.getCell(i, j);
				double multiplier; // WILL CONTAIN THE APPLICABLE MULTIPLIER BASED ON THE SITUATION

				if (examination && terrainType == 0) { // THE TERRAIN OF THE FAILED EXAMINATION WAS FLAT
					multiplier = 0.8;
				} else if (examination && terrainType == 1) { // THE TERRAIN OF THE FAILED EXAMINATION WAS HILLY
					multiplier = 0.5;
				} else if (examination && terrainType == 2) { // THE TERRAIN OF THE FAILED EXAMINATION WAS FOREST-Y
					multiplier = 0.2;
				} else { // THE TERRAIN OF THE CELL IS EITHER BLOCKED OR THE CELL WAS UNREACHABLE
					multiplier = 1;
```

```
						}

						/* DERIVED FROM BAYES' THEOREM LOOKING AT PROBABILITY OF FINDING THE TARGET IN A CELL
						 * GIVEN NEW INFORMATION (WHICH IS A FAILED EXAMINATION IN CELL OF SOME TERRAIN TYPE --
						 * TERRAIN TYPE IS REFLECTED IN THE MULTIPLIER), AND THE PRIOR IS UPDATED TO THE
						 * POSTERIOR ACCORDINGLY
						 *					SEE REPORT WRITE-UP FOR DETAILS ON DERIVATION
						 */
						temp.updateProb(temp.getProb() / (1 - (multiplier *(oldProb))));
						// System.out.println("Cell " + temp.getPos().getX() + "," + temp.getPos().getY() + " has new prob of " + temp.get

						if (temp.getProb() >= highestProb) {
							// POTENTIALLY UPDATING THE CLOSEST CELL WITH THE HIGHEST PROBABILITY
								// ESSENTIALLY, THE FOLLOWING IS HANDLING TIE-BREAKERS
								// BASED ON MANHATTAN DISTANCE FROM CURRENT POSITION

							// System.out.println("We've found a cell with a higher (or equivalent) probability.");
							int tempx = (int) temp.getPos().getX();
							int tempy = (int) temp.getPos().getY();

							double tempOne = Math.abs(pos.getPos().getX() - tempx);
							double tempTwo = Math.abs(pos.getPos().getY() - tempy);
							double tempManhattan = tempOne + tempTwo;

							if (temp.getProb() > highestProb || (temp.getProb() == highestProb && tempManhattan < currManhattan)) { //
								// System.out.println("The current lowest manhattan distance is " + currManhattan);
								cellOfHighestProb = temp;
								currManhattan = tempManhattan;
								/* System.out.println("The new cell of highest probability is " +
											cellOfHighestProb.getPos().getX() + "," +
											cellOfHighestProb.getPos().getY() +
											" with probability of " + temp.getProb());
								System.out.println("The new lowest manhattan distance is " + currManhattan); */
							}

							highestProb = temp.getProb();
						}
					}
			}

			return;

		}



		/**
		 * Updates heuristics of all cells based on the new destination.
		 * Heuristic value is computed based on manhattan distance from given destination.
		 * @param dest			The new destination cell based on probability assessments
		 */
		public void updateHeur(CellInfo dest) {

			int x = (int) dest.getPos().getX();
			int y = (int) dest.getPos().getY();

			// LOOP THROUGH ALL CELLS
			for (int i = 0; i < cols; i++) {
				for (int j = 0; j < rows; j++) {
					double one = Math.abs(j - (y));
			double two = Math.abs(i - (x));
					maze.getCell(i, j).setH(one + two);
					// System.out.println("The new H-value for " + i + "," + j + " is " + (one + two));
				}
			}

			return;
		}

		/**
		 * Assesses if the agent can move into the next cell in the planned path.
		 * If it's blocked, obviously the agent cannot.
		 * @param pos			The current position
		 * @param path			The planned path from the current position
		 * @return				Whether we can move into the next cell in the planned path, false if unable
		 */
		public boolean canMove(CellInfo pos, LinkedList<CellInfo> path) { // CHECK IF A CELL CAN ACTUALLY MOVE TO THE NEXT CELL OR NOT

			if (path.peekFirst().getTerrain().value == 3) {
				// System.out.println("Cell " + path.peekFirst().getPos().getX() + ", " + path.peekFirst().getPos().getY() + " is blocked.
				return false;
			}
			return true;
		}


		/**
		 * Assesses whether a neighbor of a cell is actually within the bounds of the maze.
		 * If either x or y is less than 0,
		 * or if x or y are greater than or equal to the number of columns or the number of rows, respectively,
		 * then the neighbor is out of bounds.
		 * Saves the user from out-of-bounds exceptions.
		 * @param coor			The neighbor of the cell
		 * @return				Whether the cell is in bounds, false if not
		 */
		public boolean inBounds(Point coor) {
			if (coor.getX() < 0 || coor.getX() >= cols || coor.getY() < 0 || coor.getY() >= rows) {
				return false;
			}
```

```java
        return true;
}


/**
 * Checks if the cell of interest is already contained
 * within the list of cells that have been expanded already.
 * @param newCell                   The cell of interest
 * @param doneWith                  The list of already expanded cells
 * @return                                  True if found, false if not
 */
public boolean contains(CellInfo newCell, ArrayList<CellInfo> doneWith) {
        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;
}


/**
 * Inserts a cell within the list of cells to be explored by the planning phase.
 * The method prioritizes cells with the lowest f-values, based on the A* algorithm.
 * @param newCell                   The cell to be inserted
 * @param toExplore                 The current list of cells to be explored by the algorithm
 * @return                                  The list with the cell inserted
 */
public ArrayList<CellInfo> insertCell(CellInfo newCell, ArrayList<CellInfo> toExplore) {

        /* System.out.println("Inserting cell into the priority queue: "
        + newCell.getPos().getX() + "," + newCell.getPos().getY() +
                        "; f-value: " + newCell.getF()); */

        // 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 WE JUST REMOVED THE ONLY CELL WITHIN THE LIST,
        // THEN THIS MAKES SURE WE DON'T ACTUALLY CREATE AN OUT-OF-BOUNDS EXCEPTION
        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()) { // TIE-BREAKER WITH G-VALUE
                        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(newCell); // ADDING IT TO THE END OF THE LIST
        return toExplore;

}




/**
 * Essentially the method where it all happens. This is the driver for the agent.
 * Main behavior involves looping through the planned path from A* towards the cell with
 * the highest probability of containing the target.
 * If the agent arrives at the destination of the planned path, it examines the cell for the target.
 * If the target is not found, or we run into a block, or the replanned path is impossible,
 * we replan again based on the updated probabilities of each cell in the maze.
 * If the target is found, we simply return success.
 * @param rowNum                    The number of rows in the yet-to-be-built maze (provided by user)
 * @param colNum                    The number of columns in the yet-to-be-built maze (provided by user)
 * @return                                  'S' for a successful trial, 'F' for a failed trial
 * @see                                     Agent6#examine(CellInfo)
 * @see                                     Agent6#plan(CellInfo, CellInfo)
 * @see                                     Maze.java
 * @see                                     CellInfo.java
```

```
 */
public static char run(int rowNum, int colNum) {

        Agent6 mazeRunner = new Agent6(); // INSTANCE 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

        // SET UP MAZE
        mazeRunner.maze = new Maze(mazeRunner.rows, mazeRunner.cols);
        // System.out.println(mazeRunner.maze.toString());
        if (!mazeRunner.maze.targetIsReachable()) {
                System.out.println("Initial check: Maze is unsolvable.\n");
                return 'F';
        }

        // System.out.println("We've successfully made a maze that is solvable.");

        long begin = System.nanoTime();
        CellInfo start = mazeRunner.maze.getCell((int)mazeRunner.maze.agentStart.getX(), (int)mazeRunner.maze.agentStart.getY());

        // WE KNOW THAT INITIALLY THE HIGHEST PROBABILITY IS SHARED BY ALL CELLS
        // WE ALSO KNOW THAT THE CLOSEST CELL TO US IS THE CELL WE'RE STARTING IN
        // TO KEEP THE IMPLEMENTATION CONSISTENT, WE'LL JUST PLAN A PATH TO WHERE WE'RE ALREADY AT
        mazeRunner.highestProb = start.getProb();
        mazeRunner.cellOfHighestProb = start;
        LinkedList<CellInfo> plannedPath = mazeRunner.plan(start, mazeRunner.cellOfHighestProb); // STORES OUR BEST PATH THROUGH THE MAZE
        // System.out.println("We've gotten through the first plan.");

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

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

                double currOne;
                double currTwo;
                double currManhattan;

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

                // ARE WE STANDING ON THE TARGET? IF WE'RE AT OUR DESTINATION (CELL WITH HIGH PROBABILITY), LET'S EXAMINE TO TRY TO FIND O
                if (currCell.getPos().getX() == mazeRunner.cellOfHighestProb.getPos().getX()
                                && currCell.getPos().getY() == mazeRunner.cellOfHighestProb.getPos().getY()) { // WE'VE HIT THE GOAL CELL

                        if (mazeRunner.examine(currCell)) { // IF WE RETURN TRUE, THEN WE'VE FOUND THE TARGET!
                                break;
                        }

                        // OTHERWISE, WE HAVE TO REPLAN FOR WHERE TO GO NEXT AND THEN WE CONTINUE ON FROM THERE
                        do {
                                /* System.out.println("Our next destination after examination is " +
                                                mazeRunner.cellOfHighestProb.getPos().getX() + "," + mazeRunner.cellOfHighestProb.getPos(
                                mazeRunner.updateHeur(mazeRunner.cellOfHighestProb);
                                plannedPath = mazeRunner.plan(currCell, mazeRunner.cellOfHighestProb);

                                if (plannedPath == null) { // WE WEREN'T ABLE TO REACH THE CELL WITH THE HIGHEST PROBABILITY
                                        double unreachableOldProb = mazeRunner.cellOfHighestProb.getProb();
                                        mazeRunner.cellOfHighestProb.updateProb(0);
                                        mazeRunner.highestProb = 0;

                                        currOne = Math.abs(currCell.getPos().getX() - mazeRunner.cellOfHighestProb.getPos().getX());
                                        currTwo = Math.abs(currCell.getPos().getY() - mazeRunner.cellOfHighestProb.getPos().getY());
                                        currManhattan = currOne + currTwo;

                                        mazeRunner.updateRemainingBeliefState(currCell, mazeRunner.cellOfHighestProb, unreachableOldProb,
                                }
                        } while (plannedPath == null);

                        /* System.out.println("The new planned path is as follows: ");
                        for (CellInfo step: plannedPath) {
                                System.out.print(step.getPos().getX() + "," + step.getPos().getY() + "; ");
                        }
                        System.out.println(); */
                }

                mazeRunner.trajectoryLength++; // WE'RE COUNTING COLLISIONS AS PART OF THE TRAJECTORY LENGTH NOW

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

                        currOne = Math.abs(currCell.getPos().getX() - mazeRunner.cellOfHighestProb.getPos().getX());
                        currTwo = Math.abs(currCell.getPos().getY() - mazeRunner.cellOfHighestProb.getPos().getY());
                        currManhattan = currOne + currTwo;

                        // UPDATE KNOWLEDGE BASE NOW THAT WE'VE FOUND A BLOCKED CELL
                        double obsOldProb = obstruction.getProb();
                        obstruction.updateProb(0);

                        if (obstruction.getPos().getX() == mazeRunner.cellOfHighestProb.getPos().getX() &&
                                        obstruction.getPos().getY() == mazeRunner.cellOfHighestProb.getPos().getY()) {
```

```java
                                mazeRunner.highestProb = 0;
                        }

                        mazeRunner.updateRemainingBeliefState(currCell, obstruction, obsOldProb, currManhattan, false);

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

                        // AND WE NEED TO REPLAN AS WELL
                        do {
                                /* System.out.println("Our next planned destination is " +
                                        mazeRunner.cellOfHighestProb.getPos().getX() + "," + mazeRunner.cellOfHighestProb.getPos(
                                mazeRunner.updateHeur(mazeRunner.cellOfHighestProb);
                                plannedPath = mazeRunner.plan(currCell, mazeRunner.cellOfHighestProb);

                                if (plannedPath == null) {
                                        double unreachableOldProb = mazeRunner.cellOfHighestProb.getProb();
                                        mazeRunner.cellOfHighestProb.updateProb(0);
                                        mazeRunner.highestProb = 0;

                                        currOne = Math.abs(currCell.getPos().getX() - mazeRunner.cellOfHighestProb.getPos().getX());
                                        currTwo = Math.abs(currCell.getPos().getY() - mazeRunner.cellOfHighestProb.getPos().getY());
                                        currManhattan = currOne + currTwo;

                                        mazeRunner.updateRemainingBeliefState(currCell, mazeRunner.cellOfHighestProb, unreachableOldProb,
                                }
                        } while (plannedPath == null);

                        /* System.out.println("The new planned path is as follows: ");
                        for (CellInfo step: plannedPath) {
                                System.out.print(step.getPos().getX() + "," + step.getPos().getY() + "; ");
                        }
                        System.out.println(); */

                        continue;
                }

        }

        // 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;

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

        return 'S';
    }

    /**
     * Main method. Program takes number of rows, number of columns and number of successful trials wanted as arguments.
     * Density of blocks within the maze is fixed at 0.3
     * @param args                      Command line arguments (refer to method description)
     * @see                             Maze.java
     */
    public static void main(String args[]) {

        // ROWS, COLUMNS, 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]);
        int successfulTrials = Integer.parseInt(args[2]);

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

        return;

    }
}

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

/**
 *
 * @author Zachary Tarman
 * Handles Agent 7 responsibilities in accordance with the descriptions
 * associated with Project 3 of CS520 Fall 2021.
 *
 */
public class Agent7 {

    /**
     * The actual maze object
     */
    public Maze maze;
    /**
     * The row dimension of the maze
     */
    public int rows;
    /**
     * The column dimension of the maze
```

```java
     */
    public int cols;
    /**
     * The probability held by the cell with the highest probability in the maze
     */
    public double highestProb;
    /**
     * The cell in which the highest probability is held
     */
    public CellInfo cellOfHighestProb;

    /**
     * The number of blocks the agent physically hits
     */
    public int collisions = 0;
    /**
     * The number of cells that we examine (assessing if we can find the target in the given cell)
     */
public int examinations = 0;
/**
 * The trajectory length of the agent (includes collisions within this metric)
 */
    public int trajectoryLength = 0;
    /**
     * The runtime of the program to find a path to the goal
     */
public long runtime = 0;


/**
 * Prints the stats that might be useful in data collection for Project 3.
 * Cost is total effort exercised by the agent.
 * @see                    Agent7#trajectoryLength
 * @see                    Agent7#examinations
 */
public void printStats() {
    System.out.println("Statistics for Maze Solution");
    System.out.println("Trajectory Length: " + trajectoryLength);
    System.out.println("Collisions: " + collisions);
    System.out.println("Examinations: " + examinations);
    int cost = trajectoryLength + examinations;
    System.out.println("Total agent cost: " + cost);
    System.out.println("Runtime: " + runtime);
    System.out.println();
    return;
}




    /**
     * This method plans a route from the agent's current position to the given destination.
     * Think of it as a single iteration of A* without the agent physically moving.
     * It uses the knowledge it has of the explored gridworld and otherwise uses the freespace assumption.
     * @param start                  The start cell
     * @param dest                   The destination cell
     * @return                              The planned path from start to finish
     */
    public LinkedList<CellInfo> plan(CellInfo start, CellInfo dest) {

            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>(); // CELLS THAT HAVE ALREADY BEEN "EXPANDED"

            // System.out.println("Starting at " + start.getPos().getX() + "," + start.getPos().getY());
            // System.out.println("Destination at " + dest.getPos().getX() + "," + dest.getPos().getY());

            CellInfo curr = start; // PTR TO THE CURRENT CELL WE'RE EVALUATING TO MOVE ON FROM IN OUR PLAN
            curr.setG(0); // SINCE THIS IS THE NEW STARTING POINT, WE SET THE G-VALUE TO 0

            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); // AND THIS IS THE FIRST CELL WE'RE GOING TO "EXPAND"

            // BEGIN LOOP UNTIL PLANNING REACHES DESTINATION CELL
            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 PROBABLY REDUNDANT, BUT HERE JUST
                            // 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 DESTINATION??
// IF SO, LET'S TRACE BACK TO OUR STARTING POSITION
if (x == (int)dest.getPos().getX() && y == (int)dest.getPos().getY()) {
        CellInfo goal = maze.getCell(x, y);
        CellInfo ptr = goal;

        // LOOP BACK THROUGH, FOLLOWING THE PARENT CHAIN BACK TO THE START
        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;
}

// IF WE DIDN'T REACH THE DESTINATION, WE HAVE TO CHECK FOR VIABLE NEIGHBORS TO CONSIDER
// 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
// THE CHECKS ESSENTIALLY CONSIST OF THE FOLLOWING
        // IS THE CELL OUT OF BOUNDS? IF SO, DON'T ADD
        // IF NOT, HAVE WE VISITED THE CELL, AND IF WE HAVE, IS IT BLOCKED? IF BOTH ARE TRUE, DON'T ADD
                // (THE AGENT ONLY KNOWS IT'S BLOCKED IF IT'S VISITED IT ALREADY)
        // IF WE'VE ALREADY ASSESSED THIS CELL WITHIN OUR PLANNING, THEN DON'T ADD IT TO THE LIST OF THINGS TO EXPLORE

// CHECKS FOR NORTHBOUND NEIGHBOR
if (!inBounds(up)) {
        addUp = false;
        // System.out.println("North is not in bounds.");
} else if (maze.getCell((int) up.getX(), (int) up.getY()).isVisited() &&
                maze.getCell((int)(up.getX()),(int) up.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THIS CE
        addUp = false;
        // System.out.println("North is blocked.");
} else if (contains(maze.getCell((int) up.getX(), (int) up.getY()), doneWith)) {
        addUp = false;
}

// CHECKS FOR SOUTHBOUND NEIGHBOR
if (!inBounds(down)) {
        addDown = false;
        // System.out.println("South is not in bounds.");
} else if (maze.getCell((int) down.getX(), (int) down.getY()).isVisited() &&
                maze.getCell((int)(down.getX()),(int) down.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THI
        addDown = false;
        // System.out.println("South is blocked.");
} else if (contains(maze.getCell((int) down.getX(), (int) down.getY()), doneWith)) {
        addDown = false;
}

// CHECKS FOR WESTBOUND NEIGHBOR
if (!inBounds(left)) {
        addLeft = false;
        // System.out.println("West is not in bounds.");
} else if (maze.getCell((int) left.getX(), (int) left.getY()).isVisited() &&
                maze.getCell((int)(left.getX()),(int) left.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THI
        addLeft = false;
        // System.out.println("West is blocked.");
} else if (contains(maze.getCell((int) left.getX(), (int) left.getY()), doneWith)) {
        addLeft = false;
}

// CHECKS FOR EASTBOUND NEIGHBOR
if (!inBounds(right)) {
        addRight = false;
        // System.out.println("East is not in bounds.");
} else if (maze.getCell((int) right.getX(), (int) right.getY()).isVisited() &&
                maze.getCell((int)(right.getX()),(int) right.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT T
        addRight = false;
        // System.out.println("East is blocked.");
} 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()); */
            }

            /* System.out.print("Cells to be explored: ");
            for (CellInfo ptr: toExplore) {
                System.out.print(ptr.getPos().toString() + "; ");
            }
            System.out.println(); */

        }

        return null; // TARGET IS UNREACHABLE ):
        // LOOK TO RUN METHOD TO SEE WHAT IS DONE WHEN THE PLANNED DESTINATION IS UNREACHABLE
    }


    /**
     * Used to examine a given cell to see if we can find the target. Updates belief state appropriately.
     * Based on the cell's terrain type, we may not actually see the target even if the agent is standing on it.
     * @param pos                 The cell to be examined
     * @return                          Whether the target has been found
     * @see                             Maze#isTarget(Point)
     * @see                             Agent7#updateRemainingBeliefState(CellInfo, CellInfo, double, double, boolean)
     */
    public boolean examine(CellInfo pos) {

        // System.out.println("Agent currently examining " + pos.getPos().getX() + ", " + pos.getPos().getY());

        this.examinations++;

        int terrainType = pos.getTerrain().value; // THIS WILL SIGNAL TO US WHAT TERRAIN THIS CELL IS
            // TERRAIN DETERMINES HOW LIKELY WE WILL BE TO SENSE THE TARGET

        double oldProb = pos.getProbContain(); // CONTAINS THE OLD PROBABILITY OF THE CELL WE'RE EXAMINING
        double newProb; // WILL CONTAIN THE NEW PROBABILITY OF THE TARGET BEING CONTAINED IN THIS CELL (IF NEEDED)
        double rand = Math.random(); // A RANDOMLY GENERATED VALUE THAT WILL DETERMINE IF WE FOUND THE TARGET OR NOT
        // System.out.println("This cell's current probability of containing the target is " + oldProb);

        /*
         * ALL OF THE EQUATIONS IN THE INNER ELSE CLAUSES WERE DERIVED USING
         * BAYES' THEOREM FOR THE PROBABILITY OF FINDING THE TARGET IN THE CURRENT CELL
         * GIVEN A FAILED EXAMINATION IN SOME TYPE OF TERRAIN (THE IF STATEMENTS TAKE
         * CARE OF THE DIFFERENT SCENARIOS WE MIGHT ENCOUNTER)
         *              SEE THE REPORT WRITE-UP TO SEE HOW WE DERIVED THESE UPDATES TO THE LIKELIHOODS
         */
        if (terrainType == 0) { // TERRAIN IS FLAT

            // System.out.println("We're on flat terrain.");
            if (maze.isTarget(pos.getPos()) && rand <= 0.8) {
                // WE'VE FOUND THE TARGET
                return true;
            } else {
                // CALCULATING THE NEW LIKELIHOOD THAT WE FIND THE TARGET HERE AFTER FAILED EXAMINATION
                newProb = (0.2 * oldProb) / (1 - (0.8 * oldProb));
            }

        } else if (terrainType == 1) { // TERRAIN IS HILLY

            // System.out.println("We're on hilly terrain.");
            if (maze.isTarget(pos.getPos()) && rand <= 0.5) {
                // WE'VE FOUND THE TARGET
                return true;
            } else {
                // CALCULATING THE NEW LIKELIHOOD THAT WE FIND THE TARGET HERE AFTER FAILED EXAMINATION
                newProb = (0.5 * oldProb) / (1 - (0.5 * oldProb));
            }
```

```java
        } else { // TERRAIN IS FOREST-Y

                // System.out.println("We're on forest terrain.");
                if (maze.isTarget(pos.getPos()) && rand <= 0.2) {
                        // WE'VE FOUND THE TARGET
                        return true;
                } else {
                        // CALCULATING THE NEW LIKELIHOOD THAT WE FIND THE TARGET HERE AFTER FAILED EXAMINATION
                        newProb = (0.8 * oldProb) / (1 - (0.2 * oldProb));
                }
        }

        // UPDATE THE BELIEF SYSTEM
        pos.updateProb(newProb);
        highestProb = pos.getProbFind();
        // System.out.println("The updated probability of this cell is " + newProb);
        // System.out.println("The currX and currY variables equal " + currX + " & " + currY);
        updateRemainingBeliefState(pos, null, oldProb, 0, true); // SEE BELOW HELPER METHOD

        return false; // WE DIDN'T FIND THE TARGET
}




/**
 * Used to update the remainder of the belief state given some event.
 * An examination, a collision, or an unreachable cell can trigger this.
 * A collision means that the blocked cell is unreachable, so the collision cell would be entered into the unreachable parameter.
 *
 * @param pos                          The cell of the current position of the agent
 * @param unreachable                  The unreachable cell, if this is not an examination
 * @param oldProb                      The belief in the examined / unreachable cell containing the target at time t (the non-upd
 * @param currManhattan                The current manhattan distance from the current cell to the (now former) cell with the highest pro
 * @param examination                  Signals whether we're looking at an examination or an unreachable situation
 */
public void updateRemainingBeliefState(CellInfo pos, CellInfo unreachable, double oldProb, double currManhattan, boolean examination) {

        int terrainType; // CONTAINS THE TERRAIN TYPE OF THE CELL WITH THE UPDATED PROBABILITY
        Point updatePosition; // THE POSITION OF THE CELL WITH THE UPDATED PROBABILITY
        if (examination) {
                terrainType = pos.getTerrain().value; // THE TERRAIN OF THE CURRENT CELL (WHICH WE JUST EXAMINED)
                updatePosition = pos.getPos();
        } else {
                terrainType = 3; // NOT NECESSARILY APPLICABLE, BUT THE CELL IS "VIRUTALLY" BLOCKED SINCE WE CAN'T OCCUPY IT
                updatePosition = unreachable.getPos();
        }

        // LOOP THROUGH ALL CELLS TO UPDATE ALL OF THEIR PROBABILITIES
        for (int i = 0; i < cols; i++) {
                for (int j = 0; j < rows; j++) {

                        // System.out.println("i and j equal " + i + " & " + j);
                        if (i == (int) updatePosition.getX() && j == (int) updatePosition.getY()) { // WE'VE ALREADY UPDATED THIS CELL ABC
                                continue;
                        }

                        CellInfo temp = maze.getCell(i, j);
                        double multiplier; // WILL CONTAIN THE APPLICABLE MULTIPLIER BASED ON THE SITUATION

                        if (examination && terrainType == 0) { // THE TERRAIN OF THE FAILED EXAMINATION WAS FLAT
                                multiplier = 0.8;
                        } else if (examination && terrainType == 1) { // THE TERRAIN OF THE FAILED EXAMINATION WAS HILLY
                                multiplier = 0.5;
                        } else if (examination && terrainType == 2) { // THE TERRAIN OF THE FAILED EXAMINATION WAS FOREST-Y
                                multiplier = 0.2;
                        } else { // THE TERRAIN OF THE CELL IS EITHER BLOCKED OR THE CELL WAS UNREACHABLE
                                multiplier = 1;
                        }

                        /* DERIVED FROM BAYES' THEOREM LOOKING AT PROBABILITY OF FINDING THE TARGET IN A CELL
                         * GIVEN NEW INFORMATION (WHICH IS A FAILED EXAMINATION IN CELL OF SOME TERRAIN TYPE --
                         * TERRAIN TYPE IS REFLECTED IN THE MULTIPLIER), AND THE PRIOR IS UPDATED TO THE
                         * POSTERIOR ACCORDINGLY
                         *                      SEE REPORT WRITE-UP FOR DETAILS ON DERIVATION
                         */
                        temp.updateProb(temp.getProbContain() / (1 - (multiplier *(oldProb))));
                        // System.out.println("Cell " + temp.getPos().getX() + "," + temp.getPos().getY() + " has new prob of " + temp.get

                        if (temp.getProbFind() >= highestProb) {
                                // POTENTIALLY UPDATING THE CLOSEST CELL WITH THE HIGHEST PROBABILITY OF FINDING THE TARGET
                                        // ESSENTIALLY, THE FOLLOWING IS HANDLING TIE-BREAKERS
                                        // BASED ON MANHATTAN DISTANCE FROM CURRENT POSITION

                                // System.out.println("We've found a cell with a higher (or equivalent) probability.");
                                int tempx = (int) temp.getPos().getX();
                                int tempy = (int) temp.getPos().getY();

                                double tempOne = Math.abs(pos.getPos().getX() - tempx);
                                double tempTwo = Math.abs(pos.getPos().getY() - tempy);
                                double tempManhattan = tempOne + tempTwo;

                                if (temp.getProbFind() > highestProb || (temp.getProbFind() == highestProb && tempManhattan < currManhatta
                                        // System.out.println("The current lowest manhattan distance is " + currManhattan);
                                        cellOfHighestProb = temp;
                                        currManhattan = tempManhattan;
                                        /* System.out.println("The new cell of highest probability of finding the target is " +
                                                        cellOfHighestProb.getPos().getX() + "," +
```

```
                                                                        cellOfHighestProb.getPos().getY() +
                                                                        " with probability of " + temp.getProb());
                                                        System.out.println("The new lowest manhattan distance is " + currManhattan); */
                                                }

                                                highestProb = temp.getProbFind();
                                        }
                                }
                        }

                        return;

                }



                /**
                 * Updates heuristics of all cells based on the new destination.
                 * Heuristic value is computed based on manhattan distance from given destination.
                 * @param dest                  The new destination cell based on probability assessments
                 */
                public void updateHeur(CellInfo dest) {

                        int x = (int) dest.getPos().getX();
                        int y = (int) dest.getPos().getY();

                        // LOOP THROUGH ALL CELLS
                        for (int i = 0; i < cols; i++) {
                                for (int j = 0; j < rows; j++) {
                                        double one = Math.abs(j - (y));
                        double two = Math.abs(i - (x));
                                        maze.getCell(i, j).setH(one + two);
                                        // System.out.println("The new H-value for " + i + "," + j + " is " + (one + two));
                                }
                        }

                        return;
                }

                /**
                 * Assesses if the agent can move into the next cell in the planned path.
                 * If it's blocked, obviously the agent cannot.
                 * @param pos                   The current position
                 * @param path                  The planned path from the current position
                 * @return                              Whether we can move into the next cell in the planned path, false if unable
                 */
                public boolean canMove(CellInfo pos, LinkedList<CellInfo> path) { // CHECK IF A CELL CAN ACTUALLY MOVE TO THE NEXT CELL OR NOT

                        if (path.peekFirst().getTerrain().value == 3) {
                                // System.out.println("Cell " + path.peekFirst().getPos().getX() + ", " + path.peekFirst().getPos().getY() + " is blocked.
                                return false;
                        }
                        return true;
                }


                /**
                 * Assesses whether a neighbor of a cell is actually within the bounds of the maze.
                 * If either x or y is less than 0,
                 * or if x or y are greater than or equal to the number of columns or the number of rows, respectively,
                 * then the neighbor is out of bounds.
                 * Saves the user from out-of-bounds exceptions.
                 * @param coor                  The neighbor of the cell
                 * @return                              Whether the cell is in bounds, false if not
                 */
                public boolean inBounds(Point coor) {
                        if (coor.getX() < 0 || coor.getX() >= cols || coor.getY() < 0 || coor.getY() >= rows) {
                                return false;
                        }
                        return true;
                }


                /**
                 * Checks if the cell of interest is already contained
                 * within the list of cells that have been expanded already.
                 * @param newCell                       The cell of interest
                 * @param doneWith                      The list of already expanded cells
                 * @return                                      True if found, false if not
                 */
                public boolean contains(CellInfo newCell, ArrayList<CellInfo> doneWith) {
                        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;
                }


                /**
                 * Inserts a cell within the list of cells to be explored by the planning phase.
                 * The method prioritizes cells with the lowest f-values, based on the A* algorithm.
                 * @param newCell                       The cell to be inserted
                 * @param toExplore                     The current list of cells to be explored by the algorithm
                 * @return                                      The list with the cell inserted
                 */
                public ArrayList<CellInfo> insertCell(CellInfo newCell, ArrayList<CellInfo> toExplore) {
```

```java
		/* System.out.println("Inserting cell into the priority queue: "
		+ newCell.getPos().getX() + "," + newCell.getPos().getY() +
				"; f-value: " + newCell.getF()); */

		// 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 WE JUST REMOVED THE ONLY CELL WITHIN THE LIST,
		// THEN THIS MAKES SURE WE DON'T ACTUALLY CREATE AN OUT-OF-BOUNDS EXCEPTION
		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()) { // TIE-BREAKER WITH G-VALUE
				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(newCell); // ADDING IT TO THE END OF THE LIST
		return toExplore;

	}




	/**
	 * Essentially the method where it all happens. This is the driver for the agent.
	 * Main behavior involves looping through the planned path from A* towards the cell with
	 * the highest probability of containing the target.
	 * If the agent arrives at the destination of the planned path, it examines the cell for the target.
	 * If the target is not found, or we run into a block, or the replanned path is impossible,
	 * we replan again based on the updated probabilities of each cell in the maze.
	 * If the target is found, we simply return success.
	 * @param rowNum			The number of rows in the yet-to-be-built maze (provided by user)
	 * @param colNum			The number of columns in the yet-to-be-built maze (provided by user)
	 * @return				'S' for a successful trial, 'F' for a failed trial
	 * @see				Agent7#examine(CellInfo)
	 * @see				Agent7#plan(CellInfo, CellInfo)
	 * @see				Maze.java
	 * @see				CellInfo.java
	 */
	public static char run(int rowNum, int colNum) {

		Agent7 mazeRunner = new Agent7(); // INSTANCE 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

		// SET UP MAZE
		mazeRunner.maze = new Maze(mazeRunner.rows, mazeRunner.cols);
		// System.out.println(mazeRunner.maze.toString());
		if (!mazeRunner.maze.targetIsReachable()) {
			System.out.println("Initial check: Maze is unsolvable.\n");
			return 'F';
		}

		// System.out.println("We've successfully made a maze that is solvable.");

		long begin = System.nanoTime();
		CellInfo start = mazeRunner.maze.getCell((int)mazeRunner.maze.agentStart.getX(), (int)mazeRunner.maze.agentStart.getY());

		// WE KNOW THAT INITIALLY THE HIGHEST PROBABILITY IS SHARED BY ALL CELLS
		// WE ALSO KNOW THAT THE CLOSEST CELL TO US IS THE CELL WE'RE STARTING IN
		// TO KEEP THE IMPLEMENTATION CONSISTENT, WE'LL JUST PLAN A PATH TO WHERE WE'RE ALREADY AT
		mazeRunner.highestProb = start.getProbFind();
		mazeRunner.cellOfHighestProb = start;
		LinkedList<CellInfo> plannedPath = mazeRunner.plan(start, mazeRunner.cellOfHighestProb); // STORES OUR BEST PATH THROUGH THE MAZE
		// System.out.println("We've gotten through the first plan.");
```

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

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

        // IF THE CELL HASN'T BEEN VISITED YET, IT COULD POTENTIALLY AFFECT THE BELIEF STATE
        if (!currCell.isVisited()) {
                currCell.setVisited();
                currCell.updateProb(currCell.getProbContain());
                if (currCell.getProbFind() > mazeRunner.highestProb) {
                        mazeRunner.highestProb = currCell.getProbFind();
                        mazeRunner.cellOfHighestProb = currCell;
                        // THE CELL THAT WE'RE CURRENTLY IN IS NOW OUR DESTINATION (PROBABLY AS A RESULT OF IT HAVING FLAT TERRAIN
                }
        }

        double currOne;
        double currTwo;
        double currManhattan;

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

        // ARE WE STANDING ON THE TARGET? IF WE'RE AT OUR DESTINATION (CELL WITH HIGH PROBABILITY), LET'S EXAMINE TO TRY TO FIND O
        if (currCell.getPos().getX() == mazeRunner.cellOfHighestProb.getPos().getX()
                        && currCell.getPos().getY() == mazeRunner.cellOfHighestProb.getPos().getY()) { // WE'VE HIT THE GOAL CELL

                if (mazeRunner.examine(currCell)) { // IF WE RETURN TRUE, THEN WE'VE FOUND THE TARGET!
                        break;
                }

                // OTHERWISE, WE HAVE TO REPLAN FOR WHERE TO GO NEXT AND THEN WE CONTINUE ON FROM THERE
                do {
                        /* System.out.println("Our next destination after examination is " +
                                        mazeRunner.cellOfHighestProb.getPos().getX() + "," + mazeRunner.cellOfHighestProb.getPos(
                        mazeRunner.updateHeur(mazeRunner.cellOfHighestProb);
                        plannedPath = mazeRunner.plan(currCell, mazeRunner.cellOfHighestProb);

                        if (plannedPath == null) { // WE WEREN'T ABLE TO REACH THE CELL WITH THE HIGHEST PROBABILITY
                                double unreachableOldProb = mazeRunner.cellOfHighestProb.getProbContain();
                                mazeRunner.cellOfHighestProb.updateProb(0);
                                mazeRunner.highestProb = 0;

                                currOne = Math.abs(currCell.getPos().getX() - mazeRunner.cellOfHighestProb.getPos().getX());
                                currTwo = Math.abs(currCell.getPos().getY() - mazeRunner.cellOfHighestProb.getPos().getY());
                                currManhattan = currOne + currTwo;

                                mazeRunner.updateRemainingBeliefState(currCell, mazeRunner.cellOfHighestProb, unreachableOldProb,
                        }
                } while (plannedPath == null);

                /* System.out.println("The new planned path is as follows: ");
                for (CellInfo step: plannedPath) {
                        System.out.print(step.getPos().getX() + "," + step.getPos().getY() + "; ");
                }
                System.out.println(); */
        }

        mazeRunner.trajectoryLength++; // WE'RE COUNTING COLLISIONS AS PART OF THE TRAJECTORY LENGTH NOW

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

                currOne = Math.abs(currCell.getPos().getX() - mazeRunner.cellOfHighestProb.getPos().getX());
                currTwo = Math.abs(currCell.getPos().getY() - mazeRunner.cellOfHighestProb.getPos().getY());
                currManhattan = currOne + currTwo;

                // UPDATE KNOWLEDGE BASE NOW THAT WE'VE FOUND A BLOCKED CELL
                double obsOldProb = obstruction.getProbContain();
                obstruction.updateProb(0);

                if (obstruction.getPos().getX() == mazeRunner.cellOfHighestProb.getPos().getX() &&
                                obstruction.getPos().getY() == mazeRunner.cellOfHighestProb.getPos().getY()) {
                        mazeRunner.highestProb = 0;
                }

                mazeRunner.updateRemainingBeliefState(currCell, obstruction, obsOldProb, currManhattan, false);

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

                // AND WE NEED TO REPLAN AS WELL
                do {
                        /* System.out.println("Our next planned destination is " +
                                        mazeRunner.cellOfHighestProb.getPos().getX() + "," + mazeRunner.cellOfHighestProb.getPos(
                        mazeRunner.updateHeur(mazeRunner.cellOfHighestProb);
                        plannedPath = mazeRunner.plan(currCell, mazeRunner.cellOfHighestProb);

                        if (plannedPath == null) {
                                double unreachableOldProb = mazeRunner.cellOfHighestProb.getProbContain();
                                mazeRunner.cellOfHighestProb.updateProb(0);
                                mazeRunner.highestProb = 0;
```

```
                    currOne = Math.abs(currCell.getPos().getX() - mazeRunner.cellOfHighestProb.getPos().getX());
                    currTwo = Math.abs(currCell.getPos().getY() - mazeRunner.cellOfHighestProb.getPos().getY());
                    currManhattan = currOne + currTwo;

                    mazeRunner.updateRemainingBeliefState(currCell, mazeRunner.cellOfHighestProb, unreachableOldProb,
                }
            } while (plannedPath == null);

            /* System.out.println("The new planned path is as follows: ");
            for (CellInfo step: plannedPath) {
                System.out.print(step.getPos().getX() + "," + step.getPos().getY() + "; ");
            }
            System.out.println(); */

            continue;
        }

    }

    // 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;

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

    return 'S';
}

/**
 * Main method. Program takes number of rows, number of columns and number of successful trials wanted as arguments.
 * Density of blocks within the maze is fixed at 0.3
 * @param args          Command line arguments (refer to method description)
 * @see                 Maze.java
 */
public static void main(String args[]) {

    // ROWS, COLUMNS, 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]);
    int successfulTrials = Integer.parseInt(args[2]);

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

    return;

}
}

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

/**
 *
 * @author Zachary Tarman
 * Handles Agent 8 responsibilities in accordance with the descriptions
 * associated with Project 3 of CS520 Fall 2021.
 *
 */
public class Agent8 {

    /**
     * The actual maze object
     */
    public Maze maze;
    /**
     * The row dimension of the maze
     */
    public int rows;
    /**
     * The column dimension of the maze
     */
    public int cols;
    /**
     * Stores the highest value in deciding what the most cost-effective destination to explore next is
     * @see         CellInfo#getC()
     */
    public double highestC;
    /**
     * The cell in which the highest c-value is held
     */
    public CellInfo cellOfHighestC;

    /**
     * The number of blocks the agent physically hits
     */
    public int collisions = 0;
    /**
     * The number of cells that we examine (assessing if we can find the target in the given cell)
     */
  public int examinations = 0;
```

```
/**
 * The trajectory length of the agent (includes collisions within this metric)
 */
    public int trajectoryLength = 0;
    /**
     * The runtime of the program to find a path to the goal
     */
public long runtime = 0;


/**
 * Prints the stats that might be useful in data collection for Project 3.
 * Cost is total effort exercised by the agent.
 * @see                     Agent8#trajectoryLength
 * @see                     Agent8#examinations
 */
public void printStats() {
    System.out.println("Statistics for Maze Solution");
    System.out.println("Trajectory Length: " + trajectoryLength);
    System.out.println("Collisions: " + collisions);
    System.out.println("Examinations: " + examinations);
    int cost = trajectoryLength + examinations;
    System.out.println("Total agent cost: " + cost);
    System.out.println("Runtime: " + runtime);
    System.out.println();
    return;
}



    /**
     * This method plans a route from the agent's current position to the given destination.
     * Think of it as a single iteration of A* without the agent physically moving.
     * It uses the knowledge it has of the explored gridworld and otherwise uses the freespace assumption.
     * @param start                 The start cell
     * @param dest                  The destination cell
     * @return                              The planned path from start to finish
     */
    public LinkedList<CellInfo> plan(CellInfo start, CellInfo dest) {

            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>(); // CELLS THAT HAVE ALREADY BEEN "EXPANDED"

            // System.out.println("Starting at " + start.getPos().getX() + "," + start.getPos().getY());
            // System.out.println("Destination at " + dest.getPos().getX() + "," + dest.getPos().getY());

            CellInfo curr = start; // PTR TO THE CURRENT CELL WE'RE EVALUATING TO MOVE ON FROM IN OUR PLAN
            curr.setG(0); // SINCE THIS IS THE NEW STARTING POINT, WE SET THE G-VALUE TO 0

            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); // AND THIS IS THE FIRST CELL WE'RE GOING TO "EXPAND"

            // BEGIN LOOP UNTIL PLANNING REACHES DESTINATION CELL
            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 PROBABLY REDUNDANT, BUT HERE JUST
                            // 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 DESTINATION??
                    // IF SO, LET'S TRACE BACK TO OUR STARTING POSITION
                    if (x == (int)dest.getPos().getX() && y == (int)dest.getPos().getY()) {
                            CellInfo goal = maze.getCell(x, y);
                            CellInfo ptr = goal;

                            // LOOP BACK THROUGH, FOLLOWING THE PARENT CHAIN BACK TO THE START
                            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;
```

```
}

// IF WE DIDN'T REACH THE DESTINATION, WE HAVE TO CHECK FOR VIABLE NEIGHBORS TO CONSIDER
// 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
// THE CHECKS ESSENTIALLY CONSIST OF THE FOLLOWING
        // IS THE CELL OUT OF BOUNDS? IF SO, DON'T ADD
        // IF NOT, HAVE WE VISITED THE CELL, AND IF WE HAVE, IS IT BLOCKED? IF BOTH ARE TRUE, DON'T ADD
                // (THE AGENT ONLY KNOWS IT'S BLOCKED IF IT'S VISITED IT ALREADY)
        // IF WE'VE ALREADY ASSESSED THIS CELL WITHIN OUR PLANNING, THEN DON'T ADD IT TO THE LIST OF THINGS TO EXPLORE

// CHECKS FOR NORTHBOUND NEIGHBOR
if (!inBounds(up)) {
        addUp = false;
        // System.out.println("North is not in bounds.");
} else if (maze.getCell((int) up.getX(), (int) up.getY()).isVisited() &&
                maze.getCell((int)(up.getX()),(int) up.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THIS CE
        addUp = false;
        // System.out.println("North is blocked.");
} else if (contains(maze.getCell((int) up.getX(), (int) up.getY()), doneWith)) {
        addUp = false;
}

// CHECKS FOR SOUTHBOUND NEIGHBOR
if (!inBounds(down)) {
        addDown = false;
        // System.out.println("South is not in bounds.");
} else if (maze.getCell((int) down.getX(), (int) down.getY()).isVisited() &&
                maze.getCell((int)(down.getX()),(int) down.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THI
        addDown = false;
        // System.out.println("South is blocked.");
} else if (contains(maze.getCell((int) down.getX(), (int) down.getY()), doneWith)) {
        addDown = false;
}

// CHECKS FOR WESTBOUND NEIGHBOR
if (!inBounds(left)) {
        addLeft = false;
        // System.out.println("West is not in bounds.");
} else if (maze.getCell((int) left.getX(), (int) left.getY()).isVisited() &&
                maze.getCell((int)(left.getX()),(int) left.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THI
        addLeft = false;
        // System.out.println("West is blocked.");
} else if (contains(maze.getCell((int) left.getX(), (int) left.getY()), doneWith)) {
        addLeft = false;
}

// CHECKS FOR EASTBOUND NEIGHBOR
if (!inBounds(right)) {
        addRight = false;
        // System.out.println("East is not in bounds.");
} else if (maze.getCell((int) right.getX(), (int) right.getY()).isVisited() &&
                maze.getCell((int)(right.getX()),(int) right.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT T
        addRight = false;
        // System.out.println("East is blocked.");
} 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);
```

```java
                // 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()); */
            }

            /* System.out.print("Cells to be explored: ");
            for (CellInfo ptr: toExplore) {
                System.out.print(ptr.getPos().toString() + "; ");
            }
            System.out.println(); */

        }

        return null; // TARGET IS UNREACHABLE ):
        // LOOK TO RUN METHOD TO SEE WHAT IS DONE WHEN THE PLANNED DESTINATION IS UNREACHABLE
    }


    /**
     * Used to examine a given cell to see if we can find the target. Updates belief state appropriately.
     * Based on the cell's terrain type, we may not actually see the target even if the agent is standing on it.
     * @param pos           The cell to be examined
     * @return              Whether the target has been found
     * @see                 Maze#isTarget(Point)
     * @see                 Agent8#updateRemainingBeliefState(CellInfo, CellInfo, double, double, boolean)
     */
    public boolean examine(CellInfo pos) {

        // System.out.println("Agent currently examining " + pos.getPos().getX() + ", " + pos.getPos().getY());

        this.examinations++;

        int terrainType = pos.getTerrain().value; // THIS WILL SIGNAL TO US WHAT TERRAIN THIS CELL IS
            // TERRAIN DETERMINES HOW LIKELY WE WILL BE TO SENSE THE TARGET

        double oldProb = pos.getProbContain(); // CONTAINS THE OLD PROBABILITY OF THE CELL WE'RE EXAMINING
        double newProb; // WILL CONTAIN THE NEW PROBABILITY OF THE TARGET BEING CONTAINED IN THIS CELL (IF NEEDED)
        double rand = Math.random(); // A RANDOMLY GENERATED VALUE THAT WILL DETERMINE IF WE FOUND THE TARGET OR NOT
        // System.out.println("This cell's current probability of containing the target is " + oldProb);

        /*
         * ALL OF THE EQUATIONS IN THE INNER ELSE CLAUSES WERE DERIVED USING
         * BAYES' THEOREM FOR THE PROBABILITY OF FINDING THE TARGET IN THE CURRENT CELL
         * GIVEN A FAILED EXAMINATION IN SOME TYPE OF TERRAIN (THE IF STATEMENTS TAKE
         * CARE OF THE DIFFERENT SCENARIOS WE MIGHT ENCOUNTER)
         *              SEE THE REPORT WRITE-UP TO SEE HOW WE DERIVED THESE UPDATES TO THE LIKELIHOODS
         */
        if (terrainType == 0) { // TERRAIN IS FLAT

            // System.out.println("We're on flat terrain.");
            if (maze.isTarget(pos.getPos()) && rand <= 0.8) {
                // WE'VE FOUND THE TARGET
                return true;
            } else {
                // CALCULATING THE NEW LIKELIHOOD THAT WE FIND THE TARGET HERE AFTER FAILED EXAMINATION
                newProb = (0.2 * oldProb) / (1 - (0.8 * oldProb));
            }

        } else if (terrainType == 1) { // TERRAIN IS HILLY

            // System.out.println("We're on hilly terrain.");
            if (maze.isTarget(pos.getPos()) && rand <= 0.5) {
                // WE'VE FOUND THE TARGET
                return true;
            } else {
                // CALCULATING THE NEW LIKELIHOOD THAT WE FIND THE TARGET HERE AFTER FAILED EXAMINATION
                newProb = (0.5 * oldProb) / (1 - (0.5 * oldProb));
            }

        } else { // TERRAIN IS FOREST-Y

            // System.out.println("We're on forest terrain.");
            if (maze.isTarget(pos.getPos()) && rand <= 0.2) {
                // WE'VE FOUND THE TARGET
                return true;
            } else {
                // CALCULATING THE NEW LIKELIHOOD THAT WE FIND THE TARGET HERE AFTER FAILED EXAMINATION
                newProb = (0.8 * oldProb) / (1 - (0.2 * oldProb));
            }
        }

        // UPDATE THE BELIEF SYSTEM
        pos.updateProb(newProb);
        highestC = pos.getProbFind() / 1;
        // System.out.println("The updated probability of this cell is " + newProb);
        // System.out.println("The currX and currY variables equal " + currX + " & " + currY);
        updateRemainingBeliefState(pos, null, oldProb, 0, true); // SEE BELOW HELPER METHOD
```

```
        return false; // WE DIDN'T FIND THE TARGET
}




/**
 * Used to update the remainder of the belief state given some event.
 * An examination, a collision, or an unreachable cell can trigger this.
 * A collision means that the blocked cell is unreachable, so the collision cell would be entered into the unreachable parameter.
 *
 * @param pos                                The cell of the current position of the agent
 * @param unreachable                   The unreachable cell, if this is not an examination
 * @param oldProb                            The belief in the examined / unreachable cell containing the target at time t (the non-upd
 * @param currManhattan                 The current manhattan distance from the current cell to the (now former) cell with the highest pro
 * @param examination                   Signals whether we're looking at an examination or an unreachable situation
 */
public void updateRemainingBeliefState(CellInfo pos, CellInfo unreachable, double oldProb, double currManhattan, boolean examination) {

        int terrainType; // CONTAINS THE TERRAIN TYPE OF THE CELL WITH THE UPDATED PROBABILITY
        Point updatePosition; // THE POSITION OF THE CELL WITH THE UPDATED PROBABILITY
        if (examination) {
                terrainType = pos.getTerrain().value; // THE TERRAIN OF THE CURRENT CELL (WHICH WE JUST EXAMINED)
                updatePosition = pos.getPos();
        } else {
                terrainType = 3; // NOT NECESSARILY APPLICABLE, BUT THE CELL IS "VIRUTALLY" BLOCKED SINCE WE CAN'T OCCUPY IT
                updatePosition = unreachable.getPos();
        }

        // LOOP THROUGH ALL CELLS TO UPDATE ALL OF THEIR PROBABILITIES
        for (int i = 0; i < cols; i++) {
                for (int j = 0; j < rows; j++) {

                        // System.out.println("i and j equal " + i + " & " + j);
                        if (i == (int) updatePosition.getX() && j == (int) updatePosition.getY()) { // WE'VE ALREADY UPDATED THIS CELL ABO
                                continue;
                        }

                        CellInfo temp = maze.getCell(i, j);
                        double multiplier; // WILL CONTAIN THE APPLICABLE MULTIPLIER BASED ON THE SITUATION

                        if (examination && terrainType == 0) { // THE TERRAIN OF THE FAILED EXAMINATION WAS FLAT
                                multiplier = 0.8;
                        } else if (examination && terrainType == 1) { // THE TERRAIN OF THE FAILED EXAMINATION WAS HILLY
                                multiplier = 0.5;
                        } else if (examination && terrainType == 2) { // THE TERRAIN OF THE FAILED EXAMINATION WAS FOREST-Y
                                multiplier = 0.2;
                        } else { // THE TERRAIN OF THE CELL IS EITHER BLOCKED OR THE CELL WAS UNREACHABLE
                                multiplier = 1;
                        }

                        /* DERIVED FROM BAYES' THEOREM LOOKING AT PROBABILITY OF FINDING THE TARGET IN A CELL
                         * GIVEN NEW INFORMATION (WHICH IS A FAILED EXAMINATION IN CELL OF SOME TERRAIN TYPE --
                         * TERRAIN TYPE IS REFLECTED IN THE MULTIPLIER), AND THE PRIOR IS UPDATED TO THE
                         * POSTERIOR ACCORDINGLY
                         *                              SEE REPORT WRITE-UP FOR DETAILS ON DERIVATION
                         */
                        temp.updateProb(temp.getProbContain() / (1 - (multiplier *(oldProb))));
                        // System.out.println("Cell " + temp.getPos().getX() + "," + temp.getPos().getY() + " has new prob of " + temp.get
                        int tempx = (int) temp.getPos().getX();
                        int tempy = (int) temp.getPos().getY();
                        double tempOne = Math.abs(pos.getPos().getX() - tempx);
                        double tempTwo = Math.abs(pos.getPos().getY() - tempy);
                        double tempManhattan = tempOne + tempTwo;
                        temp.updateC(temp.getProbFind() / (tempManhattan + 1));

                        if (temp.getC() >= highestC) {

                                // UPDATING THE MOST COST-EFFECTIVE CELL TO TRAVEL TO
                                cellOfHighestC = temp;
                                highestC = temp.getC();
                                /* System.out.println("The new cell of highest probability of finding the target is " +
                                                cellOfHighestProb.getPos().getX() + "," +
                                                cellOfHighestProb.getPos().getY() +
                                                " with probability of " + temp.getProb());
                                System.out.println("The new lowest manhattan distance is " + currManhattan); */
                        }
                }
        }

        return;

}




/**
 * Updates heuristics of all cells based on the new destination.
 * Heuristic value is computed based on manhattan distance from given destination.
 * @param dest                  The new destination cell based on probability assessments
 */
public void updateHeur(CellInfo dest) {

        int x = (int) dest.getPos().getX();
        int y = (int) dest.getPos().getY();

        // LOOP THROUGH ALL CELLS
        for (int i = 0; i < cols; i++) {
```

```java
			for (int j = 0; j < rows; j++) {
				double one = Math.abs(j - (y));
		double two = Math.abs(i - (x));
				maze.getCell(i, j).setH(one + two);
				// System.out.println("The new H-value for " + i + "," + j + " is " + (one + two));
			}
		}

		return;
	}

	/**
	 * Assesses if the agent can move into the next cell in the planned path.
	 * If it's blocked, obviously the agent cannot.
	 * @param pos			The current position
	 * @param path			The planned path from the current position
	 * @return				Whether we can move into the next cell in the planned path, false if unable
	 */
	public boolean canMove(CellInfo pos, LinkedList<CellInfo> path) { // CHECK IF A CELL CAN ACTUALLY MOVE TO THE NEXT CELL OR NOT

		if (path.peekFirst().getTerrain().value == 3) {
			// System.out.println("Cell " + path.peekFirst().getPos().getX() + ", " + path.peekFirst().getPos().getY() + " is blocked.
			return false;
		}
		return true;
	}


	/**
	 * Assesses whether a neighbor of a cell is actually within the bounds of the maze.
	 * If either x or y is less than 0,
	 * or if x or y are greater than or equal to the number of columns or the number of rows, respectively,
	 * then the neighbor is out of bounds.
	 * Saves the user from out-of-bounds exceptions.
	 * @param coor			The neighbor of the cell
	 * @return				Whether the cell is in bounds, false if not
	 */
	public boolean inBounds(Point coor) {
		if (coor.getX() < 0 || coor.getX() >= cols || coor.getY() < 0 || coor.getY() >= rows) {
			return false;
		}
		return true;
	}


	/**
	 * Checks if the cell of interest is already contained
	 * within the list of cells that have been expanded already.
	 * @param newCell				The cell of interest
	 * @param doneWith				The list of already expanded cells
	 * @return					True if found, false if not
	 */
	public boolean contains(CellInfo newCell, ArrayList<CellInfo> doneWith) {
		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;
	}


	/**
	 * Inserts a cell within the list of cells to be explored by the planning phase.
	 * The method prioritizes cells with the lowest f-values, based on the A* algorithm.
	 * @param newCell				The cell to be inserted
	 * @param toExplore				The current list of cells to be explored by the algorithm
	 * @return					The list with the cell inserted
	 */
	public ArrayList<CellInfo> insertCell(CellInfo newCell, ArrayList<CellInfo> toExplore) {

		/* System.out.println("Inserting cell into the priority queue: "
		+ newCell.getPos().getX() + "," + newCell.getPos().getY() +
				"; f-value: " + newCell.getF()); */

		// 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 WE JUST REMOVED THE ONLY CELL WITHIN THE LIST,
		// THEN THIS MAKES SURE WE DON'T ACTUALLY CREATE AN OUT-OF-BOUNDS EXCEPTION
		if (toExplore.isEmpty()) {
			toExplore.add(newCell);
			return toExplore;
```

```java
        }

        // 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()) { // TIE-BREAKER WITH G-VALUE
                        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(newCell); // ADDING IT TO THE END OF THE LIST
        return toExplore;

}




/**
 * Essentially the method where it all happens. This is the driver for the agent.
 * Main behavior involves looping through the planned path from A* towards the cell with
 * the highest probability of containing the target.
 * If the agent arrives at the destination of the planned path, it examines the cell for the target.
 * If the target is not found, or we run into a block, or the replanned path is impossible,
 * we replan again based on the updated probabilities of each cell in the maze.
 * If the target is found, we simply return success.
 * @param rowNum                    The number of rows in the yet-to-be-built maze (provided by user)
 * @param colNum                    The number of columns in the yet-to-be-built maze (provided by user)
 * @return                                  'S' for a successful trial, 'F' for a failed trial
 * @see                                     Agent8#examine(CellInfo)
 * @see                                     Agent8#plan(CellInfo, CellInfo)
 * @see                                     Maze.java
 * @see                                     CellInfo.java
 */
public static char run(int rowNum, int colNum) {

        Agent8 mazeRunner = new Agent8(); // INSTANCE 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

        // SET UP MAZE
        mazeRunner.maze = new Maze(mazeRunner.rows, mazeRunner.cols);
        // System.out.println(mazeRunner.maze.toString());
        if (!mazeRunner.maze.targetIsReachable()) {
                System.out.println("Initial check: Maze is unsolvable.\n");
                return 'F';
        }

        // System.out.println("We've successfully made a maze that is solvable.");

        long begin = System.nanoTime();
        CellInfo start = mazeRunner.maze.getCell((int)mazeRunner.maze.agentStart.getX(), (int)mazeRunner.maze.agentStart.getY());

        // WE KNOW THAT INITIALLY THE HIGHEST PROBABILITY IS SHARED BY ALL CELLS
        // WE ALSO KNOW THAT THE CLOSEST CELL TO US IS THE CELL WE'RE STARTING IN
        // TO KEEP THE IMPLEMENTATION CONSISTENT, WE'LL JUST PLAN A PATH TO WHERE WE'RE ALREADY AT
        start.updateC(start.getProbFind() / 1);
        mazeRunner.highestC = start.getC();
        mazeRunner.cellOfHighestC = start;
        LinkedList<CellInfo> plannedPath = mazeRunner.plan(start, mazeRunner.cellOfHighestC); // STORES OUR BEST PATH THROUGH THE MAZE
        // System.out.println("We've gotten through the first plan.");

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

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

                // IF THE CELL HASN'T BEEN VISITED YET, IT COULD POTENTIALLY AFFECT THE BELIEF STATE
                if (!currCell.isVisited()) {
                        currCell.setVisited();
                        currCell.updateProb(currCell.getProbContain());
                        if ((currCell.getProbFind() / 1) > mazeRunner.highestC) {
                                mazeRunner.highestC = currCell.getProbFind();
                                mazeRunner.cellOfHighestC = currCell;
                                // THE CELL THAT WE'RE CURRENTLY IN IS NOW OUR DESTINATION (PROBABLY AS A RESULT OF IT HAVING FLAT TERRAIN
                        }
                }

                double currOne;
                double currTwo;
                double currManhattan;

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

                // ARE WE STANDING ON THE TARGET? IF WE'RE AT OUR DESTINATION (CELL WITH HIGH PROBABILITY), LET'S EXAMINE TO TRY TO FIND O
                if (currCell.getPos().getX() == mazeRunner.cellOfHighestC.getPos().getX()
                                && currCell.getPos().getY() == mazeRunner.cellOfHighestC.getPos().getY()) { // WE'VE HIT THE GOAL CELL
```

```java
                if (mazeRunner.examine(currCell)) { // IF WE RETURN TRUE, THEN WE'VE FOUND THE TARGET!
                        break;
                }

                // OTHERWISE, WE HAVE TO REPLAN FOR WHERE TO GO NEXT AND THEN WE CONTINUE ON FROM THERE
                do {
                        /* System.out.println("Our next destination after examination is " +
                                        mazeRunner.cellOfHighestProb.getPos().getX() + "," +  mazeRunner.cellOfHighestProb.getPos(
                        mazeRunner.updateHeur(mazeRunner.cellOfHighestC);
                        plannedPath = mazeRunner.plan(currCell, mazeRunner.cellOfHighestC);

                        if (plannedPath == null) { // WE WEREN'T ABLE TO REACH THE CELL WITH THE HIGHEST PROBABILITY
                                double unreachableOldProb = mazeRunner.cellOfHighestC.getProbContain();
                                mazeRunner.cellOfHighestC.updateProb(0);
                                mazeRunner.cellOfHighestC.updateC(0);
                                mazeRunner.highestC = 0;

                                currOne = Math.abs(currCell.getPos().getX() - mazeRunner.cellOfHighestC.getPos().getX());
                                currTwo = Math.abs(currCell.getPos().getY() - mazeRunner.cellOfHighestC.getPos().getY());
                                currManhattan = currOne + currTwo;

                                mazeRunner.updateRemainingBeliefState(currCell, mazeRunner.cellOfHighestC, unreachableOldProb, cur
                        }
                } while (plannedPath == null);

                /* System.out.println("The new planned path is as follows: ");
                for (CellInfo step: plannedPath) {
                        System.out.print(step.getPos().getX() + "," + step.getPos().getY() + "; ");
                }
                System.out.println(); */
        }

        mazeRunner.trajectoryLength++; // WE'RE COUNTING COLLISIONS AS PART OF THE TRAJECTORY LENGTH NOW

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

                currOne = Math.abs(currCell.getPos().getX() - mazeRunner.cellOfHighestC.getPos().getX());
                currTwo = Math.abs(currCell.getPos().getY() - mazeRunner.cellOfHighestC.getPos().getY());
                currManhattan = currOne + currTwo;

                // UPDATE KNOWLEDGE BASE NOW THAT WE'VE FOUND A BLOCKED CELL
                double obsOldProb = obstruction.getProbContain();
                obstruction.updateProb(0);
                obstruction.updateC(0);

                if (obstruction.getPos().getX() == mazeRunner.cellOfHighestC.getPos().getX() &&
                                obstruction.getPos().getY() == mazeRunner.cellOfHighestC.getPos().getY()) {
                        mazeRunner.highestC = 0;
                }

                mazeRunner.updateRemainingBeliefState(currCell, obstruction, obsOldProb, currManhattan, false);

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

                // AND WE NEED TO REPLAN AS WELL
                do {
                        /* System.out.println("Our next planned destination is " +
                                        mazeRunner.cellOfHighestProb.getPos().getX() + "," +  mazeRunner.cellOfHighestProb.getPos(
                        mazeRunner.updateHeur(mazeRunner.cellOfHighestC);
                        plannedPath = mazeRunner.plan(currCell, mazeRunner.cellOfHighestC);

                        if (plannedPath == null) {
                                double unreachableOldProb = mazeRunner.cellOfHighestC.getProbContain();
                                mazeRunner.cellOfHighestC.updateProb(0);
                                mazeRunner.highestC = 0;

                                currOne = Math.abs(currCell.getPos().getX() - mazeRunner.cellOfHighestC.getPos().getX());
                                currTwo = Math.abs(currCell.getPos().getY() - mazeRunner.cellOfHighestC.getPos().getY());
                                currManhattan = currOne + currTwo;

                                mazeRunner.updateRemainingBeliefState(currCell, mazeRunner.cellOfHighestC, unreachableOldProb, cur
                        }
                } while (plannedPath == null);

                /* System.out.println("The new planned path is as follows: ");
                for (CellInfo step: plannedPath) {
                        System.out.print(step.getPos().getX() + "," + step.getPos().getY() + "; ");
                }
                System.out.println(); */

                continue;
        }

}

// 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;

System.out.println("Target Found!");
System.out.println(mazeRunner.maze.toString());
```

```
                mazeRunner.printStats();

                return 'S';
        }

        /**
         * Main method. Program takes number of rows, number of columns and number of successful trials wanted as arguments.
         * Density of blocks within the maze is fixed at 0.3
         * @param args                  Command line arguments (refer to method description)
         * @see                                 Maze.java
         */
        public static void main(String args[]) {

                // ROWS, COLUMNS, 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]);
                int successfulTrials = Integer.parseInt(args[2]);

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

                return;

        }
}

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

/**
 *
 * @author Zachary Tarman
 * Handles Agent 9 responsibilities in accordance with the descriptions
 * associated with Project 3 of CS520 Fall 2021.
 *
 */
public class Agent9 {

        /**
         * The actual maze object
         */
        public Maze maze;
        /**
         * The row dimension of the maze
         */
        public int rows;
        /**
         * The column dimension of the maze
         */
        public int cols;
        /**
         * The probability held by the cell with the highest probability in the maze
         */
        public double highestProb;
        /**
         * The cell in which the highest probability is held
         */
        public CellInfo cellOfHighestProb;

        /**
         * The number of blocks the agent physically hits
         */
        public int collisions = 0;
        /**
         * The number of cells that we examine (assessing if we can find the target in the given cell)
         */
    public int examinations = 0;
    /**
     * The trajectory length of the agent (includes collisions within this metric)
     */
        public int trajectoryLength = 0;
        /**
         * The runtime of the program to find a path to the goal
         */
    public long runtime = 0;


    /**
     * Prints the stats that might be useful in data collection for Project 3.
     * Cost is total effort exercised by the agent.
     * @see                         Agent9#trajectoryLength
     * @see                         Agent9#examinations
     */
    public void printStats() {
        System.out.println("Statistics for Maze Solution");
        System.out.println("Trajectory Length: " + trajectoryLength);
        System.out.println("Collisions: " + collisions);
        System.out.println("Examinations: " + examinations);
        int cost = trajectoryLength + examinations;
        System.out.println("Total agent cost: " + cost);
        System.out.println("Runtime: " + runtime);
        System.out.println();
        return;
```

```java
}


/**
    * This method plans a route from the agent's current position to the given destination.
    * Think of it as a single iteration of A* without the agent physically moving.
    * It uses the knowledge it has of the explored gridworld and otherwise uses the freespace assumption.
    * @param start                  The start cell
    * @param dest                   The destination cell
    * @return                               The planned path from start to finish
    */
    public LinkedList<CellInfo> plan(CellInfo start, CellInfo dest) {

            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>(); // CELLS THAT HAVE ALREADY BEEN "EXPANDED"

            // System.out.println("Starting at " + start.getPos().getX() + "," + start.getPos().getY());
            // System.out.println("Destination at " + dest.getPos().getX() + "," + dest.getPos().getY());

            CellInfo curr = start; // PTR TO THE CURRENT CELL WE'RE EVALUATING TO MOVE ON FROM IN OUR PLAN
            curr.setG(0); // SINCE THIS IS THE NEW STARTING POINT, WE SET THE G-VALUE TO 0

            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); // AND THIS IS THE FIRST CELL WE'RE GOING TO "EXPAND"

            // BEGIN LOOP UNTIL PLANNING REACHES DESTINATION CELL
            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 PROBABLY REDUNDANT, BUT HERE JUST
                            // 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 DESTINATION??
                    // IF SO, LET'S TRACE BACK TO OUR STARTING POSITION
                    if (x == (int)dest.getPos().getX() && y == (int)dest.getPos().getY()) {
                            CellInfo goal = maze.getCell(x, y);
                            CellInfo ptr = goal;

                            // LOOP BACK THROUGH, FOLLOWING THE PARENT CHAIN BACK TO THE START
                            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);
                            return plannedPath;
                    }

                    // IF WE DIDN'T REACH THE DESTINATION, WE HAVE TO CHECK FOR VIABLE NEIGHBORS TO CONSIDER
                    // 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
                    // THE CHECKS ESSENTIALLY CONSIST OF THE FOLLOWING
                            // IS THE CELL OUT OF BOUNDS? IF SO, DON'T ADD
                            // IF NOT, HAVE WE VISITED THE CELL, AND IF WE HAVE, IS IT BLOCKED? IF BOTH ARE TRUE, DON'T ADD
                                    // (THE AGENT ONLY KNOWS IT'S BLOCKED IF IT'S VISITED IT ALREADY)
                            // IF WE'VE ALREADY ASSESSED THIS CELL WITHIN OUR PLANNING, THEN DON'T ADD IT TO THE LIST OF THINGS TO EXPLORE

                    // CHECKS FOR NORTHBOUND NEIGHBOR
                    if (!inBounds(up)) {
                            addUp = false;
                            // System.out.println("North is not in bounds.");
                    } else if (maze.getCell((int) up.getX(), (int) up.getY()).isVisited() &&
                                    maze.getCell((int)(up.getX()),(int) up.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THIS CE
```

```
                                addUp = false;
                                // System.out.println("North is blocked.");
                        } else if (contains(maze.getCell((int) up.getX(), (int) up.getY()), doneWith)) {
                                addUp = false;
                        }

                        // CHECKS FOR SOUTHBOUND NEIGHBOR
                        if (!inBounds(down)) {
                                addDown = false;
                                // System.out.println("South is not in bounds.");
                        } else if (maze.getCell((int) down.getX(), (int) down.getY()).isVisited() &&
                                        maze.getCell((int)(down.getX()),(int) down.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THI
                                addDown = false;
                                // System.out.println("South is blocked.");
                        } else if (contains(maze.getCell((int) down.getX(), (int) down.getY()), doneWith)) {
                                addDown = false;
                        }

                        // CHECKS FOR WESTBOUND NEIGHBOR
                        if (!inBounds(left)) {
                                addLeft = false;
                                // System.out.println("West is not in bounds.");
                        } else if (maze.getCell((int) left.getX(), (int) left.getY()).isVisited() &&
                                        maze.getCell((int)(left.getX()),(int) left.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT THI
                                addLeft = false;
                                // System.out.println("West is blocked.");
                        } else if (contains(maze.getCell((int) left.getX(), (int) left.getY()), doneWith)) {
                                addLeft = false;
                        }

                        // CHECKS FOR EASTBOUND NEIGHBOR
                        if (!inBounds(right)) {
                                addRight = false;
                                // System.out.println("East is not in bounds.");
                        } else if (maze.getCell((int) right.getX(), (int) right.getY()).isVisited() &&
                                        maze.getCell((int)(right.getX()),(int) right.getY()).getTerrain().value == 3) { // WE'VE DETERMINED THAT T
                                addRight = false;
                                // System.out.println("East is blocked.");
                        } 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()); */
                        }

                        /* System.out.print("Cells to be explored: ");
                        for (CellInfo ptr: toExplore) {
                                System.out.print(ptr.getPos().toString() + "; ");
                        }
                        System.out.println(); */

                }

                return null; // TARGET IS UNREACHABLE ):
                // LOOK TO RUN METHOD TO SEE WHAT IS DONE WHEN THE PLANNED DESTINATION IS UNREACHABLE
        }
```

```java
    /**
     * Handles a situation where the agent discovers a cell that is "unreachable",
     * either by collision or through planning.
     * @param pos                              The current position of the agent
     * @param obstruction            The position of the unreachable cell
     * @param collision                        Indicates whether this was a legitimate collision or not
     */
public void collision(CellInfo pos, CellInfo obstruction, boolean collision) {

    obstruction.setVisited();
    obstruction.unreachable = true;
        if (collision) {
            collisions++;
            trajectoryLength++;
        }

        double blockProb = obstruction.getCurrentProb();
        obstruction.updateCurrentProb(0);
        obstruction.updateImminentProb(0);

        // System.out.println("Current beliefs based on collision update:");

        for (int i = 0; i < cols; i++) {
            for (int j = 0; j < rows; j++) {

                // System.out.println("i and j equal " + i + " & " + j);
                if (i == (int) obstruction.getPos().getX() && j == (int) obstruction.getPos().getY()) { // WE'VE ALREADY UPDATED T
                    continue;
                }

                CellInfo temp = maze.getCell(i, j);

                /* DERIVED FROM BAYES' THEOREM
                 *                          SEE REPORT WRITE-UP FOR DETAILS ON DERIVATION
                 */
                temp.updateCurrentProb(temp.getCurrentProb() / (1 - blockProb));
                // System.out.println("Cell " + temp.getPos().getX() + "," + temp.getPos().getY() + " has new current prob of " +

            }
        }

        updateImminentBeliefState(pos); // SEE BELOW METHOD

    return;
}


/**
 * Examines a cell to see if the target is there (no false negatives)
 * @param pos                The current position of the agent
 * @return                   Whether or not the target is there
 */
public boolean examine(CellInfo pos) {
    this.examinations++;
        if (maze.isTarget(pos.getPos())) { // ASSUMING NO FALSE NEGATIVES PER ARAVIND'S NOTE IN THE DISCORD CHANNEL
            return true;
        }
        return false;
}


/**
 * Provides partial sensing of whether one of the 8 neighbors
 * of the agent currently contains the target.
 * @param pos                The current position
 * @return                   Whether or not the target was found in the current cell
 * @see                          Agent9#examine(CellInfo)
 */
    public boolean sense(CellInfo pos) {

        // System.out.println("Agent currently sensing " + pos.getPos().getX() + ", " + pos.getPos().getY());

        double posOriginalProb = 0;
        double collectiveProbability = 0; // STORES THE TOTAL PROBABILITY OF CURRENT AND NEIGHBORING CELLS

        if (examine(pos)) {
            return true;
        }
        posOriginalProb = pos.getCurrentProb();
        collectiveProbability = posOriginalProb;
        pos.updateCurrentProb(0);

        // THE TARGET ISN'T IN THE CURRENT CELL, SO LET'S LOOK AROUND US (PARTIAL SENSING)
        int x = (int) pos.getPos().getX();
        int y = (int) pos.getPos().getY();
        boolean targetSensed = false; // INDICATES WHETHER THE TARGET HAS BEEN SENSED AROUND THE AGENT OR NOT

        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.isTarget(neighbors.get(i))) {
                                targetSensed = true;
                                // System.out.println("Target sensed in the surrounding neighbors.");
                        }
                        collectiveProbability += maze.getCell((int) neighbors.get(i).getX(), (int) neighbors.get(i).getY()).getCurrentProb
                } else {
                        neighbors.set(i, null);
                        // System.out.println("This neighbor wasn't in the bounds of the maze.");
                }
        }

        // System.out.println("Current beliefs based on examination and sensing:");

        // UPDATING THE BELIEF SYSTEM FOR THE CURRENT STATE OF THE GRID-WORLD
                // STARTING WITH THE NEIGHBORING CELLS
        if (targetSensed) { // WE KNOW THAT THE TARGET IS ONE OF THE NEIGHBORS, AND THEY SHOULD BE WEIGHTED ACCORDINGLY
                        collectiveProbability -= posOriginalProb;
                for (Point neighbor: neighbors) {
                        if (neighbor != null) {

                                CellInfo temp = maze.getCell((int) neighbor.getX(), (int) neighbor.getY());
                                /* DERIVED FROM BAYES' THEOREM
                                 *                                      SEE REPORT WRITE-UP FOR DETAILS ON DERIVATION
                                 */
                                temp.updateCurrentProb(temp.getCurrentProb() / (collectiveProbability));
                                // System.out.println("Cell " + temp.getPos().getX() + "," + temp.getPos().getY() + " has new current prob
                        }
                }
        } else { // THE TARGET IS DEFINITIELY NOT IN ONE OF THE NEIGHBORS AND SO THEY SHOULD GO TO 0
                for (Point neighbor: neighbors) {
                        if (neighbor != null) {
                                CellInfo temp = maze.getCell((int) neighbor.getX(), (int) neighbor.getY());
                                temp.updateCurrentProb(0);
                                // System.out.println("Cell " + temp.getPos().getX() + "," + temp.getPos().getY() + " has new current prob
                        }
                }
        }

                // NOW FOR THE REST OF THE CELLS IN THE GRIDWORLD
        for (int i = 0; i < cols; i++) {
                for (int j = 0; j < rows; j++) {

                        // System.out.println("i and j equal " + i + " & " + j);
                        if (i == (int) pos.getPos().getX() && j == (int) pos.getPos().getY()) { // WE'VE ALREADY UPDATED THIS CELL ABOVE
                                continue;
                        }

                        boolean alreadyUpdated = false;

                        for (Point neighbor: neighbors) {
                                if (neighbor != null) {
                                        if (i == (int) neighbor.getX() && j == (int) neighbor.getY()) { // WE'VE ALREADY UPDATED THIS CELL
                                                alreadyUpdated = true;
                                                break;
                                        }
                                }
                        }

                        if (alreadyUpdated) {
                                continue;
                        }

                        CellInfo temp = maze.getCell(i, j);
                        if (targetSensed) { // WE KNOW THAT THIS CELL DEFINITIELY DOESN'T CONTAIN THE TARGET
                                temp.updateCurrentProb(0);
                        } else { // WE KNOW THAT THE NEIGHBORS OF THE CURRENT CELL DIDN'T CONTAIN THE TARGET, SO THIS CHANGES ACCORDINGLY
                                /* DERIVED FROM BAYES' THEOREM
                                 *                                      SEE REPORT WRITE-UP FOR DETAILS ON DERIVATION
                                 */
                                temp.updateCurrentProb(temp.getCurrentProb() / (1 - collectiveProbability));
                        }

                        // System.out.println("Cell " + temp.getPos().getX() + "," + temp.getPos().getY() + " has new current prob of " +

                }
        }

        updateImminentBeliefState(pos); // SEE BELOW HELPER METHOD

        return false; // WE DIDN'T FIND THE TARGET YET
}



/**
 * Updates the beliefs of where the target will be in the next time unit.
 * Used for deciding where to plan to go next.
```

```
 * @param pos             The current position of the agent
 */
public void updateImminentBeliefState(CellInfo pos) {

        // System.out.println("Imminent beliefs:");
        highestProb = 0;

        // LOOP THROUGH ALL CELLS TO UPDATE ALL OF THEIR PROBABILITIES
        for (int i = 0; i < cols; i++) {
                for (int j = 0; j < rows; j++) {

                        CellInfo temp = maze.getCell(i, j);
                        if (temp.unreachable) { // THIS CELL IS KNOWN TO BE BLOCKED, SO THE TARGET CANNOT GO HERE
                                continue;
                        }

                        // THIS WILL STORE THE NEXT "IMMINENT" PROBABILITY
                                // TAKES THE SUM OF EACH OF THE NEIGHBORS' "CURRENT" PROBABILITIES
                                // WITH EACH TERM DIVIDED BY THE NUMBER OF VALID NEIGHBORS IT HAS
                                // THIS IS BECAUSE IF THE TARGET WERE IN THAT CELL CURRENTLY,
                                // IT HAS AN EQUAL PROBABILITY OF MOVING INTO ANY VALID NEIGHBOR
                                // SO IT MUST BE DIVIDED UP APPROPRIATELY
                        double cumulativeProb = 0;

                        // THE AGENT AND THE TARGET CAN ONLY MOVE IN THE FOUR CARDINAL DIRECTIONS
                        Point n = new Point(i, j - 1);
                        Point w = new Point(i - 1, j);
                        Point s = new Point(i, j + 1);
                        Point e = new Point(i + 1, j);

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

                        for (Point point: neighbors) {

                                if (!inBounds(point)) { // A NEIGHBOR NEEDS TO BE IN BOUNDS
                                        continue;
                                }

                                CellInfo temp2 = maze.getCell((int) point.getX(), (int) point.getY());

                                int validNeighbors = 0; // COUNTS THE NUMBER OF VALID NEIGHBORS THAT THIS NEIGHBOR HAS ITSELF
                                int x = (int) point.getX();
                                int y = (int) point.getY();

                                Point nneighbor = new Point(x, y - 1);
                                Point wneighbor = new Point(x - 1, y);
                                Point sneighbor = new Point(x, y + 1);
                                Point eneighbor = new Point(x + 1, y);

                                ArrayList<Point> neighborsOfNeighbor = new ArrayList<Point>();
                                neighborsOfNeighbor.add(nneighbor);
                                neighborsOfNeighbor.add(wneighbor);
                                neighborsOfNeighbor.add(sneighbor);
                                neighborsOfNeighbor.add(eneighbor);

                                for (Point point2: neighborsOfNeighbor) {
                                        if (inBounds(point2)) {
                                                if ((!maze.getCell((int) point2.getX(), (int) point2.getY()).isVisited()) ||
                                                                maze.getCell((int) point2.getX(), (int) point2.getY()).getTerrain().value
                                                        validNeighbors++;
                                                }
                                        }
                                }

                                cumulativeProb = cumulativeProb + ((temp2.getCurrentProb()) / validNeighbors);
                                // System.out.println("Cell " + temp2.getPos().getX() + "," + temp2.getPos().getY() + " has " +
                                                // validNeighbors + " valid neighbors.");

                        }

                        temp.updateImminentProb(cumulativeProb);
                        if (temp.getPos().getX() != pos.getPos().getX() || temp.getPos().getY() != pos.getPos().getY()) {
                                if (temp.getImminentProb() > highestProb) {
                                        highestProb = temp.getImminentProb();
                                        cellOfHighestProb = temp;
                                        // System.out.println("The new cell of highest probability is " + temp.getPos().getX() + "," + tem
                                }
                        }
                        // System.out.println("Cell " + temp.getPos().getX() + "," + temp.getPos().getY() + " has new imminent prob of " +
                }
        }

        return;

}


/**
 * Refreshes the current probabilities from the given imminent probabilities.
 * Simulates the agent's knowledge transferring to the next time unit.
 * @param pos             The current position of the agent
 */
public void nextTimeUnit(CellInfo pos) {

        highestProb = 0;
```

```
		for (int i = 0; i < cols; i++) {
			for (int j = 0; j < rows; j++) {

				CellInfo temp = maze.getCell(i, j);
				temp.updateCurrentProb(temp.getImminentProb());
				// System.out.println("Cell " + temp.getPos().getX() + "," + temp.getPos().getY() + " has new prob of " + temp.get
			}
		}

		// System.out.println();
		// System.out.println("---Next time unit---");

		return;

	}


	/**
	 * Assesses if the agent can move into the next cell in the planned path.
	 * If it's blocked, obviously the agent cannot.
	 * @param pos			The current position
	 * @param path			The planned path from the current position
	 * @return				Whether we can move into the next cell in the planned path, false if unable
	 */
	public boolean canMove(CellInfo pos, LinkedList<CellInfo> path) { // CHECK IF A CELL CAN ACTUALLY MOVE TO THE NEXT CELL OR NOT

		if (path.peekFirst().getTerrain().value == 3) {
			// System.out.println("Cell " + path.peekFirst().getPos().getX() + ", " + path.peekFirst().getPos().getY() + " is blocked.
			return false;
		}
		return true;
	}



	/**
	 * Updates heuristics of all cells based on the new destination.
	 * Heuristic value is computed based on manhattan distance from given destination.
	 * @param dest			The new destination cell based on probability assessments
	 */
	public void updateHeur(CellInfo dest) {

		int x = (int) dest.getPos().getX();
		int y = (int) dest.getPos().getY();

		// LOOP THROUGH ALL CELLS
		for (int i = 0; i < cols; i++) {
			for (int j = 0; j < rows; j++) {
				double one = Math.abs(j - (y));
		double two = Math.abs(i - (x));
				maze.getCell(i, j).setH(one + two);
				// System.out.println("The new H-value for " + i + "," + j + " is " + (one + two));
			}
		}

		return;
	}


	/**
	 * Assesses whether a neighbor of a cell is actually within the bounds of the maze.
	 * If either x or y is less than 0,
	 * or if x or y are greater than or equal to the number of columns or the number of rows, respectively,
	 * then the neighbor is out of bounds.
	 * Saves the user from out-of-bounds exceptions.
	 * @param coor			The neighbor of the cell
	 * @return				Whether the cell is in bounds, false if not
	 */
	public boolean inBounds(Point coor) {
		if (coor.getX() < 0 || coor.getX() >= cols || coor.getY() < 0 || coor.getY() >= rows) {
			return false;
		}
		return true;
	}


	/**
	 * Checks if the cell of interest is already contained
	 * within the list of cells that have been expanded already.
	 * @param newCell				The cell of interest
	 * @param doneWith				The list of already expanded cells
	 * @return					True if found, false if not
	 */
	public boolean contains(CellInfo newCell, ArrayList<CellInfo> doneWith) {
		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;
	}


	/**
	 * Inserts a cell within the list of cells to be explored by the planning phase.
	 * The method prioritizes cells with the lowest f-values, based on the A* algorithm.
	 * @param newCell				The cell to be inserted
```

```
 * @param toExplore                    The current list of cells to be explored by the algorithm
 * @return                                 The list with the cell inserted
 */
public ArrayList<CellInfo> insertCell(CellInfo newCell, ArrayList<CellInfo> toExplore) {

        /* System.out.println("Inserting cell into the priority queue: "
        + newCell.getPos().getX() + "," + newCell.getPos().getY() +
                        "; f-value: " + newCell.getF()); */

        // 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 WE JUST REMOVED THE ONLY CELL WITHIN THE LIST,
        // THEN THIS MAKES SURE WE DON'T ACTUALLY CREATE AN OUT-OF-BOUNDS EXCEPTION
        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()) { // TIE-BREAKER WITH G-VALUE
                        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(newCell); // ADDING IT TO THE END OF THE LIST
        return toExplore;

}


/**
 * Used to make sure that our total probability is still summing to 1
 * @return              Whether or not something has gone wrong with our probability calculations
 */
public boolean errorCheck() {

        double cumulativeProb = 0;
        for (int i = 0; i < cols; i++) {
                for (int j = 0; j < rows; j++) {
                        cumulativeProb += maze.getCell(i, j).getCurrentProb();
                }
        }

        System.out.println("Total probability = " + cumulativeProb);

        if (cumulativeProb < 0.999 || cumulativeProb > 1.001) {
                return true;
        }

        return false;
}



/**
 * Essentially the method where it all happens. This is the driver for the agent.
 * Main behavior involves looping through the planned path from A* towards the cell with
 * the highest probability of containing the target.
 * At each step, the agent examines and senses for the target.
 * If the target is not found, or we run into a block, or the replanned path is impossible,
 * we replan again based on the updated probabilities of each cell in the maze.
 * If the agent arrives at the destination of the planned path, it replans.
 * If the target is found, we simply return success.
 * @param rowNum                       The number of rows in the yet-to-be-built maze (provided by user)
 * @param colNum                       The number of columns in the yet-to-be-built maze (provided by user)
 * @return                                  'S' for a successful trial, 'F' for a failed trial
 * @see                                     Agent9#examine(CellInfo)
 * @see                                     Agent9#plan(CellInfo, CellInfo)
 * @see                                     Maze.java
 * @see                                     CellInfo.java
 */
public static char run(int rowNum, int colNum) {

        Agent9 mazeRunner = new Agent9(); // INSTANCE 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

// SET UP MAZE
mazeRunner.maze = new Maze(mazeRunner.rows, mazeRunner.cols);
// System.out.println(mazeRunner.maze.toString());
if (!mazeRunner.maze.targetIsReachable()) {
        System.out.println("Initial check: Maze is unsolvable.\n");
        return 'F';
}

// System.out.println("We've successfully made a maze that is solvable.");

long begin = System.nanoTime();
CellInfo start = mazeRunner.maze.getCell((int) mazeRunner.maze.agentStart.getX(), (int) mazeRunner.maze.agentStart.getY());
CellInfo currCell = start;

// WE KNOW THAT INITIALLY THE HIGHEST PROBABILITY IS SHARED BY ALL CELLS
// WE ALSO KNOW THAT THE CLOSEST CELL TO US IS THE CELL WE'RE STARTING IN
// TO KEEP THE IMPLEMENTATION CONSISTENT, WE'LL JUST PLAN A PATH TO WHERE WE'RE ALREADY AT
mazeRunner.highestProb = start.getCurrentProb();
mazeRunner.cellOfHighestProb = start;
LinkedList<CellInfo> plannedPath = mazeRunner.plan(start, mazeRunner.cellOfHighestProb); // STORES OUR BEST PATH THROUGH THE MAZE

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

        currCell = plannedPath.poll();

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

        // IF THE CELL HASN'T BEEN VISITED YET, IT COULD POTENTIALLY AFFECT THE BELIEF STATE
        if (!currCell.isVisited()) {
                currCell.setVisited();
        }

        if (mazeRunner.sense(currCell)) { // IF WE RETURN TRUE, THEN WE'VE FOUND THE TARGET!
                break;
        }

        // DO WE NEED TO REPLAN?
        if (plannedPath.isEmpty() ||
                        (plannedPath.peekLast().getPos().getX() != mazeRunner.cellOfHighestProb.getPos().getX() ||
                                        plannedPath.peekLast().getPos().getY() != mazeRunner.cellOfHighestProb.getPos().getY())) {

                int stuck = 0;

                do {

                        if (stuck > 10) {
                                System.out.println("The agent has gotten stuck somehow. Try again.");
                                return 'F';
                        }
                        /*System.out.println("Our next destination after examination is " +
                                        mazeRunner.cellOfHighestProb.getPos().getX() + "," + mazeRunner.cellOfHighestProb.getPos(
                                        " which has imminent probability of " + mazeRunner.highestProb +
                                        " (Check: this is the associated probability: " + mazeRunner.cellOfHighestProb.getImminent
                        mazeRunner.updateHeur(mazeRunner.cellOfHighestProb);
                        plannedPath = mazeRunner.plan(currCell, mazeRunner.cellOfHighestProb);

                        if (plannedPath == null) { // WE WEREN'T ABLE TO REACH THE CELL WITH THE HIGHEST PROBABILITY
                                mazeRunner.collision(currCell, mazeRunner.cellOfHighestProb, false);
                        } else if (plannedPath.size() > 1) { // WE KNOW WE'RE NOT PLANNING TO ARRIVE WHERE WE ALREADY ARE
                                plannedPath.poll();
                        }

                        stuck++;

                        /* try {
                                Thread.sleep(250);
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        } */

                } while (plannedPath == null);

                /*System.out.println("The new planned path is as follows: ");
                for (CellInfo step: plannedPath) {
                        System.out.print(step.getPos().getX() + "," + step.getPos().getY() + "; ");
                }
                System.out.println();*/

        }

        // OUR PLANNED PATH IS STILL OKAY AS FAR AS WE KNOW IF WE'RE HERE
        // ATTEMPT TO EXECUTE EXACTLY ONE CELL MOVEMENT
        while (!mazeRunner.canMove(currCell, plannedPath)) {
                // WE'VE FOUND / HIT A BLOCK
                mazeRunner.collision(currCell, plannedPath.peekFirst(), true);

                // DEBUGGING STATEMENT
                // System.out.println("We've hit a block at coordinate " + plannedPath.peekFirst().getPos().toString());
                int stuck = 0;
                // AND WE NEED TO REPLAN AS WELL
                do {
```

```
					if (stuck > 10) {
						System.out.println("The agent has gotten stuck somehow. Try again.");
						return 'F';
					}
					/*System.out.println("Our next planned destination is " +
							mazeRunner.cellOfHighestProb.getPos().getX() + "," +  mazeRunner.cellOfHighestProb.getPos(
					mazeRunner.updateHeur(mazeRunner.cellOfHighestProb);
					plannedPath = mazeRunner.plan(currCell, mazeRunner.cellOfHighestProb);

					if (plannedPath == null) { // WE WEREN'T ABLE TO REACH THE CELL WITH THE HIGHEST PROBABILITY
						mazeRunner.collision(currCell, mazeRunner.cellOfHighestProb, false);
					} else if (plannedPath.size() > 1) { // WE KNOW WE'RE NOT PLANNING TO ARRIVE WHERE WE ALREADY ARE
						plannedPath.poll();
					}

					/* try {
						Thread.sleep(250);
					} catch (InterruptedException e) {
						// TODO Auto-generated catch block
						e.printStackTrace();
					} */
					stuck++;

				} while (plannedPath == null);

				/*System.out.println("The new planned path is as follows: ");
				for (CellInfo step: plannedPath) {
					System.out.print(step.getPos().getX() + "," + step.getPos().getY() + "; ");
				}
				System.out.println();*/

				mazeRunner.nextTimeUnit(currCell);
				mazeRunner.maze.moveTarget();

			}

			// REFRESH FOR THE NEW ITERATION
			mazeRunner.trajectoryLength++;
			mazeRunner.nextTimeUnit(currCell);
			mazeRunner.maze.moveTarget();
			/* try {
				Thread.sleep(50);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} */
		}

		mazeRunner.errorCheck();

		// 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;

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

		return 'S';
	}

	/**
	 * Main method. Program takes number of rows, number of columns and number of successful trials wanted as arguments.
	 * Density of blocks within the maze is fixed at 0.3
	 * @param args			Command line arguments (refer to method description)
	 * @see				Maze.java
	 */
	public static void main(String args[]) {

		// ROWS, COLUMNS, 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]);
		int successfulTrials = Integer.parseInt(args[2]);

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

		return;

	}
}
```