CS 4710 Artificial Intelligence — Sydney Johnson (slj8au)
Homework 2: Path Finding — Mamadou Diallo (mdd4vn)

Important Notes:

- Everything in this report is a reflection on our path finding solution, where we weigh the pros and cons of various optimizations and modifications. We do not believe there is a perfect solution to this assignment, therefore don't make any such claims.
- Occasionally, when running the uncertain algorithm, the program may reach an infinite loop or terminate. If this happens, this is the result our loop catcher, which we describe in more detail below. If this occurs, please restart the program.
- Please use the Test.java file when running our program. There are changes (boolean uncertainity, myRobot.passInfo(), as well as others) that are made to ensure our algorithm runs as effectively as possible.
- Uncertain and certain Worlds are both handled in one file, MyRobotClass.java. To set uncertainty and input files, please refer to **line 15 of Test.java**.

Data Structures:

Node: Represents an position in the world. Holds the position's coordinate, a pointer to the parent node, f g and h values, as well as the pinged value of that position.
G = the cost to reach this node from the starting node
H = the estimated/heuristic cost to reach the destination from here
Originally used the Manhattan distance as the heuristic, but we soon changed this to Euclidean distance. We will discuss this change, as well as others, later in this write-up.
F = G + H (total cost of the node)

NodeList: Essentially an ArrayList of Nodes, with a few additional helper methods.

Original Algorithm Description (assuming *uncertainty* is false):

Our basic path finding algorithm is A* (A-Star) and it is implemented using Nodes (Nodes.java) and NodeList (NodeList.java), which are both user implemented classes. We initialize two NodeLists, *open* and *closed*. The closed list stores all paths that are either worse or no better than the current path at which we are currently looking. While we have not reached our goal, our algorithm considers the best node in the open list (having the lowest f value). This node's neighbors are created and have their respective f,g, and h values calculated. For each neighbor, we check if it already exists in both the open and closed lists. If so, and if new recalculated node costs are better, we replace them in their respective

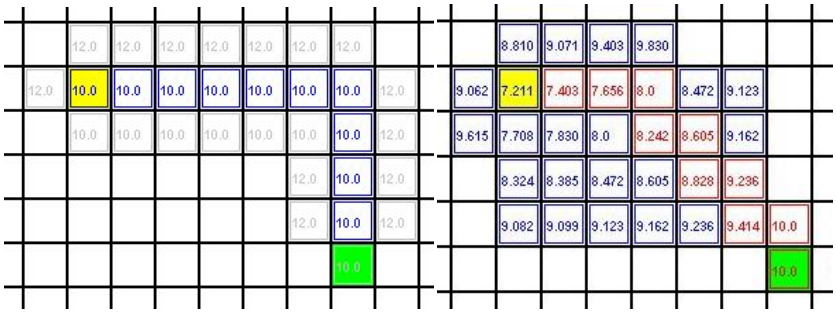
list. Otherwise, we set the node values and add the unvisited node to the open list. We repeat this process until we reach the "F" node. Once this happens, we simply traverse the parents nodes of the closed list nodes beginning with the last element to build the path. Finally, we pass this path to the robot's move method and the program will terminate with statistics of the path finding algorithm.

Optimizations:

1. Change Heuristic

We noticed fairly quickly that our algorithm wasn't always taking the optimal path. This was the result of our original heuristic choice, the Manhattan distance. Using the Manhattan distance as a heuristic caused our pathfinder to avoid diagonal paths which, for certain maps, wasted a significant amount of moves. We instead opted to use Euclidean distance (Java's Point class's distance() method uses this by default) and reduced our number of moves considerably. However, this did not come without tradeoffs. This reduced our number of moves, but at the cost of using more pings. We've illustrated this cost below:

Manhattan vs. Euclidean



images from: wiki.gamegardens.com/Path_Finding_Tutorial

Due to the increase of pings caused by changing our heuristic, we looked for ways to optimized our pings.

2. Null Pings:

After printing out debug code to see exactly where our algorithm was pinging, we noticed that we were wasting pings on locations that were outside the world's dimensions.

Originally, since the ping method was returning "null", we believed these pings wouldn't count against us but found otherwise after further inspection. To avoid this, we passed additional (and according to Professor Floryan, allowed) parameters to the MyRobotClass to help choose pings more wisely. These include *myWorld.numRows(), myWorld.numCols()*, as well as *myWorld.getEndPos()* and a Boolean variable to represent *uncertainty*.

3. PingMap:

    Another place we initially wasted pings was repeated pinging. To solve this, we used Java's HashMap to create a *PingMap.* This mapped points in our world to their respective ping value once we ping it the first time. Any other time this point's value is needed, we simply check the map for the point and avoid the additional ping. This essentially put the world's value in memory, and reduced our ping amount drastically.

Uncertain Algorithm Description (assuming *uncertainty* is true):

Fundamentally, our uncertain algorithm is almost identical to the certain one with a few modifications to deal with the probability factor. Before we were pinging the world, deciding the best path, and passing the information back to the robot so it could travel safely and confidently. Now, since further pings are likely false, we are forced to move throughout the path finding algorithm in order to obtain the most valid pings possible. The issues that caused the most trouble while we were implementing this algorithm were incorrect pings and infinite loops (which were likely due to incorrect pings). We made 4 major changes to our original algorithm to account for this new uncertainty, and the problems that came alongside it.

1. pingLimit and move()

When you run this program with uncertainty set to true, you will always be prompted to enter an integer value before it starts. This value, pingLimit, will represent the maximum number of pings the robot is allowed to perform before it must move along the current path is had found. The higher the pingLimit, the higher the probability that you will get incorrect pings and have the program terminate before it completes. The lower the pingLimit, the lower the probability that you will get incorrect pings, however this comes at the cost of an almost completely greedy algorithm as the pingLimit gets closer to 0. We've found a reasonable amount of luck with around 40 pings, and recommend that when the program is initialized.

2. Penta-pinging

Each time we ping a location in the world there is a probability that it could return the wrong value. This is a major program, as an incorrect ping could cause the program to run infinitely or crash. At the cost of 5 times as many pings, we've implement a function in our pinging process that pings a location a total of 5 times and uses the value that is pinged at least 3 times as the value that is added to the Ping Map. Since our algorithm has a variable pingLimit, if the pingLimit is low (meaning our algorithm only pings locally) then our algorithm ensures a higher probability of correct pings. However, adversely, if the pingLimit is set higher than it could likely result in 5 times as many incorrect pings. As mentioned earlier, we've found that 40 works as a good pingLimit. The reason 40 has been more favorable is because the algorithm pings its maximum of 8 neighbors 5 times (8 x 5 = 40) which results in at least a single move before a complete ping cycle, thus reducing the frequency of incorrect pings.

3. Clearing Ping Map

Even with 5 pings on a single location before adding it to the Ping Map, we found that our Ping Map still contained incorrect pings while debugging. Since incorrect pings are the most

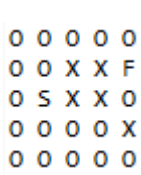
detrimental factor working with this algorithm, we decided it would be best to clear the Ping Map after each ping cycle (when pingcount > pingLimit) at the cost of a significant amount of added pings and moves. This reduced the number of incorrect pings, and consequently the number of infinite loops and program crashes. While this change improves path finding for most worlds, we found that it adversely affected a few others. For grading, if the uncertain algorithm loops or crashes, please try removing the clear and running the program again. This clear is located at **line 53 of MyRobotClass.java**, and can be easily commented out to stop this clearing.

4. Kill process in case of infinite loop

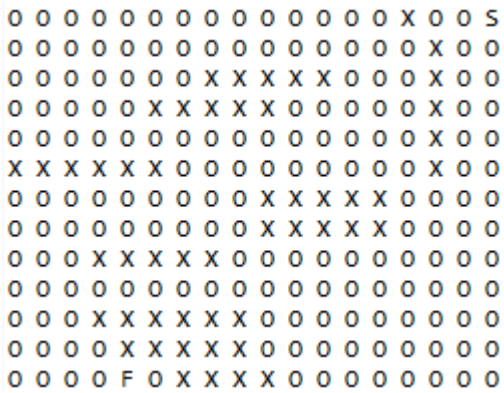
With all the changes above, our uncertain algorithm would still crash or loop infinite given certain paths, based on the pingLimit. To prevent this from destroying our computer's CPU, as well as the TAs, we've implemented a loop catcher that will catch most if not all loops and terminate the program with a "Program was terminated to avoid infinite loop. Please restart program." message. In this case, restarting the program typically fixes the issues but sometimes it helps to change the pingLimit and try again.

Results:

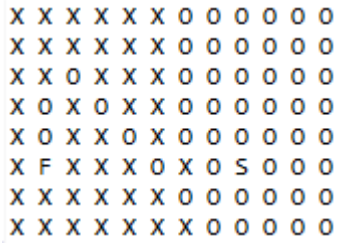
All test files in the report are included in the HW2 folder.



myInputFile1.txt



myInputFile2.txt



myInputFile3.txt

All tests where uncertainty is true, the pingLimit is set to 40m the PingMap is cleared after moves, and the best statistics of 3 trials is displayed below.

	Uncertainty = false	Uncertainty = true
myInput File1.txt	**Moves = 5 / Pings = 22** <terminated> Test (1) [Java Application] C:\Program Files\Java\jdk1.7.0_11\bi Loading world from file: myInputFile1.txt Validating the World: Checking for errors... World loaded! Everything seems ok! Uncertainity is set to false. You reached the destination! Total number of moves: 5 Total number of pings: 22	**Moves = 5 / Pings = 110** <terminated> Test (1) [Java Application] C:\Program Files\Java\jdk1.7.0_ Loading world from file: myInputFile1.txt Validating the World: Checking for errors... World loaded! Everything seems ok! Uncertainity is set to true. Please input an an integer value to set pingLimit. If 40 You reached the destination! Total number of moves: 5 Total number of pings: 110
myInput File2.txt	**Moves = 20 / Pings = 110** <terminated> Test (1) [Java Application] C:\Program Files\Java\jdk1.7. Loading world from file: myInputFile2.txt Validating the World: Checking for errors... World loaded! Everything seems ok! Uncertainity is set to false. You reached the destination! Total number of moves: 20 Total number of pings: 110	**Moves = 21 / Pings = 585** <terminated> Test (1) [Java Application] C:\Program Files\Java\jdk1.7.0_ Loading world from file: myInputFile2.txt Validating the World: Checking for errors... World loaded! Everything seems ok! Uncertainity is set to true. Please input an an integer value to set pingLimit. If 40 You reached the destination! Total number of moves: 21 Total number of pings: 585
myInput File3.txt	**Moves = 10 / Pings = 76** <terminated> Test (1) [Java Application] C:\Program Files\Java\jdk1.7.0_ Loading world from file: myInputFile3.txt Validating the World: Checking for errors... World loaded! Everything seems ok! Uncertainity is set to false. You reached the destination! Total number of moves: 10 Total number of pings: 76	**Moves = 12 / Pings = 550** <terminated> Test (1) [Java Application] C:\Program Files\Java\jdk1.7.0 Loading world from file: myInputFile3.txt Validating the World: Checking for errors... World loaded! Everything seems ok! Uncertainity is set to true. Please input an an integer value to set pingLimit. If 40 You reached the destination! Total number of moves: 12 Total number of pings: 550

Conclusions:

While both algorithms performed about the same in regards the moves, we see that the uncertain algorithm's Penta-ping increase the number of pings by a multiple of 5, or more in the case of myInputFile3. Also, while this was not apparent in the results above, the range of solutions for the uncertain algorithm is very wide. The computed paths could range from anything between optimal to highly inefficient. This is the result of incorrect pings, that lead to the robot moving along points it has already traveled on until the PingMap is cleared, and contains the correct ping.

Regardless of whether the algorithm was certain or not, there was also a continuous struggle to balance tradeoffs between pings and moves which could not have been avoided. Ultimately, we decided to focus more on minimizing moves, and thus in some cases the number of pings can grow

rapidly. While it is not documented above, when given a less efficient data set both algorithms infinitely loop, or terminate prematurely. A less efficient data set would be described as one with no solution- meaning the 'F' is surrounded by 'X's or if there is no 'F' at all. This is the only way that the robot would never be able to reach the finish mark.

In retrospect, this was a definitely a time consuming but highly rewarding assignment. We realized quickly that no change to our certain algorithm could possibly make our uncertain algorithm, bullet proof, however all the optimizations and modification helped significantly improve the path solution and reduce the frequency of infinite loops/crashes. With a little more time, and research, we would have liked to implemented a multithreaded uncertain algorithm which would start another instance of the application whenever an infinitely loop was detected and killed. This would have solved the issue of infinitely loop paths, for solvable worlds, and helped enhance the reliability of our uncertain algorithm.