IEOR 140: Project 2 Milestone 4

Team 6: MoonSoo Choi & Sherman Siu

**Sherman**: experiment runner, coder, code commenter, report writer

**MoonSoo**: experiment runner, coder, code commenter, report reviser

**Time spent:** Approximately 12 hours each person

**Project Description:**

We programmed our robot to read the destination grid coordinate (through buttons), and reach the desired destination. While approaching the destination, a robot may have to establish its traveling directions and appropriate amount of travelling grids. After approaching the destination, robot can repeat this activity as many times as the user wishes to.

Unlike Project 2 Milestone 3, the robot navigates via a Shortest Path method. The robot treats each coordinate point on the grid as a node, and navigates towards certain nodes using Dijkstra’s Algorithm. The robot is also responsible for detecting obstacle blocks using its ultrasonic sensor. If the robot encounters a node with a block obstacle on it, the robot will recalculate a new shortest path to reach its destination, taking into account that there is an obstacle at that node.

Robot user can use buttons of a robot to enter coordinates of a destination; in our project, ButtonCounter class handles this activity. Through this class, the robot will be able to display X and Y values of destination, increase/decrease those values, and save the data for later uses.

**Experimental work:**

Our experimental work was to indicate the measured distance to a block node at one and two intersections away. Using the robot to read the getDistance() values from the ultrasonic sensor, we determined the following values:

**LOCATION getDistance() VALUES**

ONE intersection away 23

TWO intersections away 46

With these values, we can determine when the robot should recalculate Dijkstra’s Algorithm when it detects an obstacle block from one or two nodes away. If the robot’s next node in its shortest algorithm is blocked, then the robot can detect this block so it will not run into the block. Instead, once the robot detects this blocked node from one or two intersections away, the robot will then stop navigating and recalculate Dijkstra’s Algorithm with new information that the certain node is blocked. In our GridNavigator class

**Task Analysis:**

**GridNavigator.java**

* **Coordinate\_adj: (int direction)**
  + Above method adjusts the new location of the robot, after the pilot travels 1 blocks in a certain direction. It adds/subtracts appropriate x- and y- values according to the current heading of the pilot.
* **Navigate: (int direction, Node \_currentnode)**
  + Navigate method takes in current heading direction and current location as its two parameters, and travel forward if its respective neighboring nodes are not null, and either turn to the left or to the right, whichever node is closer to the destination (in other words, whichever node has smaller getDistance() value than the current node)
* **ShortestPath (int x, int y)**
  + Two parameters x and y of shortestpath method represents x- and y- coordinate of the designated destination node. The main responsibility of this method is to recalculate the shortest path, and block any nodes that have obstacles detected by the ultrasonic sensor, and recalculate accordingly to such blocked and unblocked nodes.

**Tracker.java**

The Tracker class is responsible for the robot steering along the blue tape and detecting a black marker. The robot uses the light sensors to navigate the robot along the blue line, only referring to GridNavigator for the next step when it encounters black tape (or when the light values are less than 0). Tracker is also responsible for the calibration of the robot.

**Grid.java**

The Grid class is responsible for calculating the distances of each node on the coordinate grid from your desired destination. Throughout the grid navigation, the Grid class will maintain an internal “map” of the node distances and will keep track of which nodes are blocked (as determined by the robot as it navigates). The robot will navigate based off of the smallest distance value determined by Grid.java for each neighboring node.

**Node.java**

The Node class contains all the information regarding the individual nodes in Grid.java. For each node, using the methods in Node.java one can determine whether the node is blocked or not, the x-y coordinates of the node, and the distance value of the nodes. The Grid class uses the Node class to keep track of each individual node information in the coordinate grid.

**ButtonCounter.java**

The ButtonCounter class allows the user to input coordinates into the robot so the robot knows where to navigate. This class has not changed since Milestone 3.

**Most interesting/challenging/difficult part of the project:**

The most difficult part of this project definitely involves coding shortest path and integrating shortest path into the robot’s grid navigation class. To code shortest path we first needed to develop a good understanding of Dijkstra’s Algorithm and how to integrate this algorithm in a (x,y) coordinate grid. This method involved calculating a grid of distances from a certain destination, using an array list to sort all the nodes/points. Once that was figured out, the next challenge was teaching the robot how to navigate the grid using shortest path. This involved the robot having to determine the smallest distance value among its neighboring nodes and then moving to that neighboring node. Once arrived, the robot will again have to determine the best distance value among its new neighbors and navigate there; the robot repeats this process until it arrives at its desired destination. Of course, the difficulty also involved recalculating shortest path once it detects an interfering blocked node, and then circumventing that node to reach the desired location.

**Programs:**

GridNavigator.java

Grid.java

Node.java

Tracker.java (has not changed since Milestone 3)

ButtonCounter.java (has not changed since Milestone 3)

Javadocs are in the doc folder.