Contents

1 Introduction

Robotics developed so far has achieved great success in the world of industrial manufacturing and scientific exploring. One of the most famous and common application is Robot Arms, or *manipulators* which can move with high speed and great accuracy to perform certain tasks such as spot welding and painting. Its great accuracy and precision have helped the electronics industry to place chips or other tiny self-mounted components, making the advanced electronics devices more portable and convenient to carry with [7].

However, these robots have a limited range of motion and can only perform tasks within this range as it is bolted down to a certain position. This is when mobile robots come in. Mobile robots are expected to move freely throughout the manufacturing plant and apply its applications to finish some tasks wherever its needed and required, significantly improving the manufacturing efficiency. To move safely, mobile robots are required to recognize landmarks and objects and avoid any obstacle that may causes troubles to the robots. One possible approach to recognize landmarks and objects is memory-based object recognition algorithm that recognizes objects with non-linear features, which is developed by Randal C. Nelson [4]. This algorithm has a time-consuming process where precise circular views of each object must be measured and stored in a database which is also obviously hard to implement in a relatively complex environment.

This report aims to provide an implementation of a mobile robot capable of sensing the environment, navigating it and performing actions based on the captured and processed. The final assignment is the *Grid World* where the robot ought to be capable of moving along the grip and finding an optimal route from the starting node to the goal node. The robot used in this project is a prebuilt robot with modular components based on Lego NXT platform [2].

2 Methodology

This section of the report will detail the exact methodology employed to complete the *Grid World* challenge. At first, the process of preparing the hardware and software required for the project will be described, and subsequently the partial components necessary to build the final model. Finally, the complete implementation of a robot capable of tangible behaviour within the Gird World environment that was developed in this project will be outlined.

2.1 Setup

The robots selected for this project were based on the Lego NXT platform [2]. Use of this hardware enabled component modularity and a use of well-documented Java API to program its behaviour - leJOS. The robot components include:

- **Body**. The main component of the robot that includes the processing unit, a display screen, components ports, buttons for basic control of the robot, and a USB port that enables to transfer compiled code from a computer to the robot for execution.
- **Motors**. Two motors, each controlling one wheel of the robot, connected to the body using an output port. Among other features, the programmer can set the speed and direction of each motor.
- **Sonar**. One sonar sensor connected to the body using an input port. The main functionality of this component is measuring the distance to the closest object detectable by the sonar.
- **Light sensor**. Two light sensors connected to the body using input ports. Each of them is capable of returning the light intensity. To increase measurement accuracy, sensors have a built-in LED (flood light) which emits additional light which reflected luminosity can be sampled.

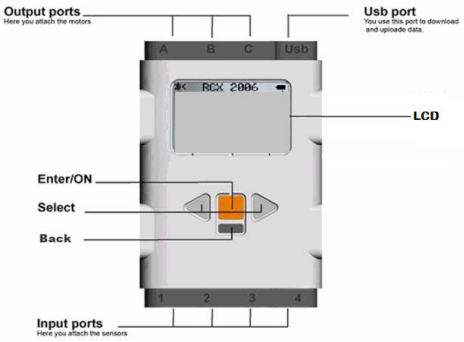

Figure 1: Lego NXT body (also know as the "brick") which is used to connect the robot's components together and provide a link through USB with the computer [6].

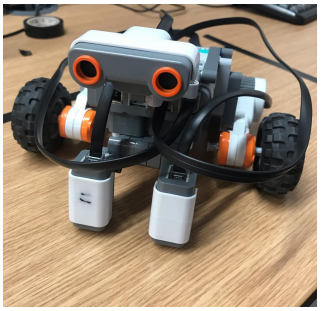

Figure 2: Assembled Lego NXT robot with parts described in section 2.1 (no cables included to improve clarity of the image) [1].

All three sensors are modular, hence, can be replaced, attached in different locations on the robot's body and rotated to a variety of positions. All components may also be connected to any compatible ports, i.e. a motor to any output port, but not to an input port (see Figure 1). As a result, the correctness of functionality is dependent on an appropriate port to the component assignment in the program's code. Moreover, some leJOS classes also require prior measurement, most notably, the wheel dimensions and track width (distance between the centre of tires) for motor control, which were determined using a measuring tape.

The software leJOS software utilised in this project allows for designing Java programs for modelling robot's behaviour. The source code is developed in Atom IDE with *Atom Build* package installed. The package enables automatic code deployment with a designated bash script that is required for correct compilation and data transfer between the computer and the robot.

2.2 Ultrasonic sensors

Figure 3: A single Lego NXT ultrasonic sensor that allows the robot to detect its distance to other objects in sensor's range. Image from [6].

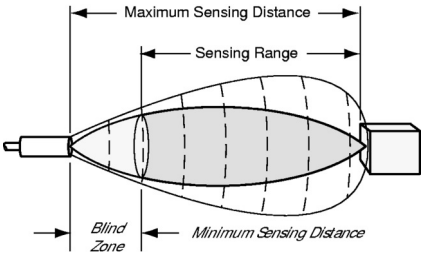

Figure 4: A sonar cone generated by an ultrasonic scanner. The effective range of the sensor is the region of a 3D space between the maximum range and the blind zone [5]. The accurate range of Lego NXT ultrasonic sensor is between 6 and 180 cm [6].

An interface with the sensor is provided by the `UltrasonicSensor` class. In this project the sonar was used only in the default mode (for different modes see [3]), in which the `getDistance()` method returns the distance to the closest object in sensor's range (up to 255 cm), however, the sonar is capable of detecting up to 8 objects [3].

2.3 Light sensors

The Lego NXT light sensors measure the light intensity (also known as luminosity). The sensor can register visible as well as invisible to human eye light (e.g. infrared). Moreover, it is also possible to enable a red LED which is used as an active (environment) method for producing light which reflection can be measured.

An interface with the sensor is provided by the `LightSensor` class. The most important method of the class the allows to measure the light intensity is `getLightValue()`. The function returns a percentage corresponding to the brightness of detected light, 0% for darkness and 100% for intense light.

Figure 5: A single Lego NXT light sensor that measures light luminosity. Image from [6]

In order for the robot to trace a black line, the `DifferentialPilot` class (see section 2.4.1) was utilised for movement control, and two states of the robot had to be modelled - on the line and off the line. Assuming the starting position to be "on the line", the robot remains in that state as long as both sensors read high light intensity values. These initial light values are registered on start and are compared against in later execution. Once a difference between the current and the stored light values exceeds the measurement error margin (5% of the initial value) then the robot is deemed to be "off the line". If the decrease in luminosity was detected then the robot was stopped and three possibilities were considered:

1. If right value indicated line, then the robot was rotated right by a small angle (e.g. -5 degrees)
2. If left value indicated line, then the robot was rotated left by a small angle (e.g. 5 degrees)
3. If both values indicated line, then the robot was at a junction and would first move forward by a small step (e.g. `travel(3)`) to position itself directly above the junction, before choosing a random direction of rotation and turn until a line was detected.

All four cases (including the "on the line" case) were described as `if` within an infinite loop. Such implementation resulted in an endless random walk of the robot along black lines. The solution was limited in the sense that many constants were used, which led to erroneous behaviour in unpredictable environment configurations (e.g. thicker lines).

2.4 Movement

The subsequent subsections will outline some of the possible methods for controlling robot's movement by managing its motors (see Figure 6). Not only is each motor able to rotate through gear reduction, but also provide information on the degrees turned by feedback sensing [6].

Figure 6: A single Lego NXT motor that allows the robot to travel in the environment. Image from [6]

2.4.1 Differential Pilot

The most common approach for motor control involves the application `DifferentialPilot` class implemented by the leJOS library. An instance of this class is responsible for steering both motors, therefore, object's constructor requires instances of both motors (using a static method of the

corresponding port in Motor class), as well as tire diameter (equivalent for both wheels) and the distance between both wheels. The `DifferentialPilot` interface allows the programmer to model robot's movement without directly handling motor synchronisation, for example, pre-implemented `rotate()` function instead of explicitly turning wheels in opposite directions.

Other important methods are:

- `travel()` - a blocking method (no further execution until a given function exits) for moving motors by a given distance.
- `forward()` - a non-blocking method (further execution of the program, while the method is running "in the background") for initiating a forward movement. The method can be interrupted using the `stop()` function.
- `isMoving()` - a method that returns true if the robot's motors are in motion, false otherwise.
- `rotate()` - a function that turns the robot in place by a specified angle, positive for anti-clockwise and negative for clockwise.
- `setTravelSpeed()` and `setRotateSpeed()` - methods for modifying the default speed of robot's movement and rotation respectively.

Correct configuration of the parameters described in section 2.1 posed a challenge, as it was crucial to determine precise turning angle. The robot interleaved movement by a distance unit with rotation by 60 degrees, hence, any inaccuracy in angle resulted in erroneous final position. This is due to not exact mapping of the argument passed to the `rotate()` to the robot's heading change in degrees.

2.4.2 Regulated Motor

Another technique is controlling the motors directly with the instance of `NXTRegulatedMotor` class. This allows for better robustness and more freedom in terms of movement behaviour, however, introduces problems such as exact distance measuring or synchronisation. Thanks to less constricted movement options that the `NXTRegulatedMotor` class accommodates, a Braitenburg vehicle could be designed. The light sensors were pointed in the direction of movement and the luminosity measurements (see subsection 2.3) were normalised in order for the result to be in a range of reasonable rotation speed, e.g. half of the maximal speed (returned by `getMaxSpeed()`) of the slower motor. The results were used as an argument to the `setSpeed()` method of the motor on the same side of the device. Such a configuration produced a "fear-of-light" model, i.e. the robot moves away from light sources. To reverse the behaviour, each normalised light value can be supplied to the motor on the opposite side, resulting in a robot "chasing" light sources.

2.4.3 Navigator

The final movement method adopted in this project used the `Navigator` class. This interface provides even further abstraction to the motor control problem than the `DifferentialPilot` class. The 3-dimensional state space the robot moves in is projected onto a 2-dimensional Cartesian coordinate system, i.e. every legal position of the robot can be described by its X-coordinate and Y-coordinate. Such locations can be instantiated by constructing a `Waypoint` object, a sequence of which forms a `Path`. Motors' rotation is handled by one of the `Pilot` classes (e.g. `DifferentialPilot`) which is an argument to the `Navigator` class constructor. Each `Waypoint` was stored in a user-defined `Node` class which together formed a graph structure. Objects of that class contained not only point's coordinates and its label, but also a `visited` flag and a list of its neighbouring `Node` objects. The robot was meant to explore the resulting graph by recursively selecting an unexplored neighbours of consecutive `Node` objects, until the goal was reached or no unexplored nodes remained. Nodes included in the path to the goal point were stored on a stack and used to create a `Path` object for the robot to follow.

3 Grid World

The problem faced by the robot is called "Grid World". Approaches and solutions previously described in this report were utilised to achieve the aim of this project. The goal was to explore a grid consisting of black square cells by transitioning the robot along the lines (see Figure 7). The device had to account for any obstacles and variances in terms of dimensions (different length of square sides or number of cells). The assumption of a static world was adopted, hence, no modifications to the gird and obstacles could occur throughout the state space exploration. Finally, the shortest path from the starting state in the bottom right corner to the goal state in the top right corner had to be found. The software developed to complete the task was divided into four classes:

- **MazeRunner**. The main class that implements the DFS, instantiates all classes with sampled prior parameters and handles their synchronised execution.
- **Grid**. The class that handles the set of explored nodes, where each node is a unique junction (or corner) of the grid. It also implements the shortest path finding algorithm between any two explored nodes and it stores a set detected obstacles.
- **Robot**. The class which manages any sensory measurements (obstacles and light values), as well as the robot's movement.
- **Node**. The class that is used to represent a single node of the grid. The main function of the object is to provide a list of neighbours (can be visited or unvisited) of that junction.

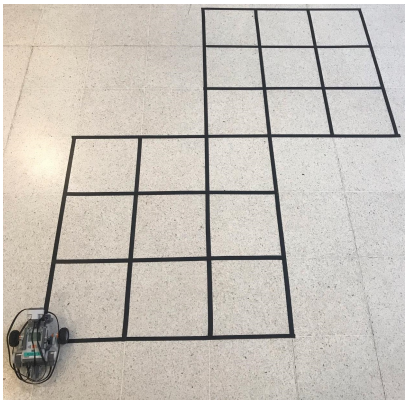

Figure 7: Example of the Grid World environment with no obstacles. The starting position (where the robot is situated on the picture) is the bottom left corner and the goal position is in the top right corner of the grid.

3.1 MazeRunner class

The **MazeRunner** class consists of a `main()` and `explore()` functions. The first task of the main function is to initialise required objects: (1) start node and current node as a `Node` object with coordinates $(0, 0)$, new `Robot` object, new `Grid` object and an integer variable representing current heading as north. Finally, the method begins the exploration of the grid by calling `explore` with the starting node as the argument. The `explore` function is then recursively called to explore the grid using an online DFS approach. Online in this context entails that the robot must actually follow the exploration, as it has to use sensor measurements to determine junction shape or detect an obstacle. Due to this, in this project, it is also assumed that all legal neighbours (with positive coordinates) for every node exist and are later pruned if no path is found. The `explore` method first marks the argument `Node` as explored and adds it to the grid. Then for all neighbours we execute the main algorithm as shown in appendix A.1.

3.2 Robot class

The next discussed class is the **Robot**, which most complex component that was the `moveTo()` transition function from one node to another. The constructor creates objects for all the sensors and records their nominal values (separate for each). The first method is the line following algorithm - `followLine()`. The enhanced version included obstacle detection case, so if the `UltrasonicSensor` detected an obstacle, then the robot would return to the source node by executing the following code:

```
if(sonar.getDistance() < OBSTACLE_DISTANCE) {
    pilot.stop();
    result = OBSTACLE;                // set the result to indicate obstacle detection
    pilot.rotate(BASE_ANGLE * 2);
    pilot.forward();
}
```

The next helper function was designed to find the correct turning angle. The implemented method took as input the source `Node` object, target `Node` object and the current heading. The produced output was a number of degrees to turn represented as a double:

```
private static double getRotation(Node from, Node to, int from_dir) {
    // find relative position of the target node
    int to_dir;
    if (from.getX()>to.getX()) to_dir = H_WEST;
    else if (from.getX()<to.getX()) to_dir = H_EAST;
    else if (from.getY()>to.getY()) to_dir = H_SOUTH;
    else to_dir = H_NORTH;

    // check if a 180 turn
    if (Math.abs(from_dir-to_dir)==2) return BASE_ANGLE * 2;

    // return the turning angle
    return (from_dir-to_dir)%2 * -BASE_ANGLE;
}
```

The next sophisticated function was created handle robot's rotation and path that connects the node detection. In the employed approach, the robot turned the majority of the required angle at a high speed and subsequently slow down for the remainder until either a limit was reach or the line was found(See appendix A.2).

All above functions are combined in the `moveTo()` method. It handles the entire procedure of transitioning between two nodes and accounts for obstacles. Moreover, it ensures shortest angle rotation and detects whether the desired target node is reachable at all (if the path connects the nodes). Implementation of `moveTo()` can be found in appendix A.3.

3.3 Node class

The constructor of the `Node` class creates all valid neighbours, i.e. has positive coordinates and there is no obstacle between them. One improvement is a method that allows returning either all explored or all unexplored neighbours of that node:

```
public ArrayList<Node> getNeighbours(boolean exp) {
    ArrayList<Node> result = new ArrayList<>();
    for(Node neighbour : this.neighbours) {
        if(neighbour.getExplored() == exp) result.add(neighbour);
    }
    return result;
}
```

3.4 Grid class

The final class used in this assignment is the `Grid` class, which provides a snapshot of the explored environment. It stores visited nodes, maintains a symmetric map of obstacles between all nodes and, most importantly, allows to determine the quickest explored way between any two points.

The solution for finding the shortest path from a starting node to a destination node is to conduct a Depth-First Traversal of a given directed graph between these two nodes and store all possible paths in a set and then find the path with the shortest length. Starting from the source node, the algorithm keeps storing the visited node in an array - `path[]`. Once the destination node is found, add the path to the set. One important thing to be noticed is that current vertices in `path[]` need to be marked as visited also so that the traversal will not end up having a loop or circle. Then another `for` loop is used to find the shortest path which should have a minimal number of vertices or nodes in it. This algorithm for finding the shortest path is stable(preserves the order of paths with same distance) so the first path it finds with the shortest distance will be returned.

4 Results

The robot in its final configuration (as presented in section 3) had successfully explored the grid maze and was able to store the representation of the environment within the `Grid` object which prints it in human-readable form to robot's LCD screen. The ratio of correct full runs is 4:2, however, due to the lack of time for thorough testing, insufficient data was gathered for the measure to be representative.

The robot was not found to ever throw an exception to exit without finishing the program's execution. Despite this, the device does tend to corners. After further investigation, it was determined that the right corners posed a greater challenge to the robot and it was likely to re-calibrate instead of detecting a new node.

Figure 8: An example of a position that could result in incorrect behaviour with higher probability.

The robot is invariable to error propagation, thanks to re-calibration during line following and dynamic rotation algorithm. The approaches remove as much prior knowledge (e.g. hard-coded maximal light intensity) and constants as possible, hence, making the model more general.

The shortest path finding algorithm has always found the optimal route and successfully avoided stored obstacles. However, previously mentioned node classification issues could result in incorrect state space representation and hence erroneous path following. Apart from these isolated situations, the backpropagation and transition algorithms (rotation, line detection and following) proved properly functional.

5 Discussion

During the development of the project, we have encountered several issues and problems that did affect the performance of the mobile robot and relatively hindered our developing process.

We first had troubles in getting and setting initial parameters. Due to the inaccuracy of the movement and rotation of the robot, we had to use a naive try and error to get the initial parameters such as angle required for rotating 90 degrees or distance traveled for centering the robot in the junctions. For the `followline()` function mentioned in the previous section, initially, a naive approach was created, which consisted of traveling until either sensor detects a line, i.e., light intensity decreases. This method was deemed very sensitive to any variations and not sufficiently robust. Moreover, when the robot reaches a junction, the initial approach is to turn to the exact position and later rotate a small degree to check whether a line is found. However, this method was prone to error propagation, hence, inconsistent even with an exhaustive tuning of parameters. To improve the model a more intelligent approach was used in which the robot turned the majority of the required angle at high speed and subsequently slow down for the remainder until either a limit was reach or the line was found.

Light intensity, the only variables used to for robot's vision and the measurement for checking whether the robot is on the line or junctions is of vital importance to the performance of the robot navigation process. The performance of the light sensors is one of the significant limitations discovered. A critical assumption is made, which is assuming that the grid route is consistently dark and the ground area is consistently bright, and there is a clear boundary between the darkness and brightness. However, the surface materials can have a profound effect on the light value detected by the sensor where a smooth and reflective surface could return more accurate light intensity value while a rough surface would cause light diffusion, resulting in lower light intensity. Besides, we also assume the background light intensity does not change over time. Changing the background light intensity may confuse brightness and darkness, making the robot difficult to distinguish from the line from the ground.

6 Conclusion

We have aimed to provide an implementation of the mobile robots that could navigate based on the information collected. In other words, our purpose is to develop an efficient algorithm for robot navigation. Although the algorithm developed has successfully made the robot to move and find the optimal route as expected, several major problems in robot navigation such as robot localization, goal recognition or sense fusion remain unsolved, and the algorithm is not robust enough to be applied in the real scenario or more complex environments.

The complexity of the robot navigation can be increased due to more uncertainties added to the environment, or less environment information are collected. Some common problems or uncertainties are: (1) lack of knowledge about the environment and (2) dynamic moving obstacles, (3) failures occur when sensing the environment and (4) the uncertainties and unreliability of the robot equipment. Our current approach does not provide an applicable solution to address any of the issues and the robot would fail to finish the last task in a constraint-free environment with the presence of the above factors.

7 Future work

The major task we have accomplished so far is the algorithm developed for the mobile robot to find its way out and provide an optimal route in the *Grid World*. There are many potential improvements we can make to further improve the performance. One of the possible improvements is to have more sensors for more accurate environment sensing or have sensors that are more precise and accurate or improve the structure of the wheel motors so we can have a more precise rotation. Also, further research on building a gird world with dynamic moving obstacles and running experiments on the performance of current robot navigation algorithm can be done and a more robust robot navigation algorithm can be developed from the results of the experiments.

References

[1] Bogatech. Teachers Introduction Course to LEGO® Mindstorms NXT & EV3. http://www.bogatech.org/cursos/Curs%20Introduccio%20Lego%20Mindstorms%20NXT/UNIT%209.htm, 2010.

[2] Lego. Lego Mindstorm: Build a Robot. https://www.lego.com/en-us/mindstorms/build-a-robot, 2018.

[3] leJOS Team. Overview - leJOS NXJ API documentation. http://www.lejos.org/nxt/nxj/api/, 2018.

[4] Randal C. Nelson. *Memory-Based Recognition for 3-D Ob jects*. University of Rochester, 1996.

[5] PhysComp. Sonar Sensor. https://digitalmedia.risd.edu/pbadger/PhysComp/index.php?n=Devices.DistanceSensor, 2018.

[6] MathCS Robotics. The NXT Toolkit. https://mathcs.org/robotics/nxt-java/building/nxt_intro.html, 2013.

[7] R. Siegwart and I. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. Massachusetts Institute of Technology, 2004.

Appendices

A Implementation of the algorithm for *Grid World*

A.1 Body of the DFS while-loop

```
// backtrack if node to explore is not the node robot is currently located at
if (n != current_node) {
    ArrayList<Node> path = grid.getShortestPath(current_node, n);
    heading = robot.followPath(path, heading);
    current_node = n;
}

// select a random unexplored neighbour and try to move to that node
Node selectedNode = n.getRandNeigbour();
selectedNode.makeNeighbours();
int moveResult = robot.moveTo(n, selectedNode, heading);

// if obstacle found, add it to obstacle list, explore other neighbours
// reverse the heading (see followLine method)
if (moveResult==Robot.OBSTACLE) {
    heading = Robot.getHeading(selectedNode, n);
    grid.addObstacle(n, selectedNode);
    n.removeNeighbour(selectedNode);
}

// if no path, then prune the neighbour and explore others
else if (moveResult==Robot.NO_PATH) {
    n.removeNeighbour(selectedNode);
}

// if new neighbour found, then recursively explore it
else {
    heading = Robot.getHeading(n, selectedNode);
    current_node = selectedNode;
    explore(selectedNode);
}
```

A.2 Rotation with path detection on junction

```
 public boolean rotateRobot(double init_angle, double scan_angle) {
    // if chose to move forward, check if path exists in range [-20,20]
    if (init_angle==0) {
      boolean pathFound1 = rotateRobot(0, 20);
      boolean pathFound2 = rotateRobot(-20, -20);
      return pathFound1 || pathFound2;
    }

    // otherwise begin by rotating initial angle in high speed
    pilot.rotate(init_angle);
    pilot.setRotateSpeed(ROTATE_SLOW);

    // choose correct (outer) light sensor to check for line detection
    LightSensor outerLS = (scan_angle<0) ? lightSensorR : lightSensorL;
    int default_lv = (scan_angle<0) ? LS_LIGHT_R : LS_LIGHT_L;

    // initialise non-blocking slow rotation
```

```
    pilot.rotate(scan_angle, true);
    boolean isPath = false;

    // as the robot rotates, check for line detection, if found set flag and stop
    while(pilot.isMoving()) {
      if (default_lv - outerLS.getLightValue() > LS_MARGIN) {
        isPath = true;
        pilot.stop();
      }
    }
    pilot.setRotateSpeed(ROTATE_NORMAL);

    // if no path found, then rotate back
    if(!isPath) pilot.rotate(-init_angle-scan_angle);

    // return the found flag to caller method
    return isPath;
}
```

A.3 State transition function

```
public int moveTo(Node from, Node to, int heading) {

    // backpropagation check: move only to adjacent nodes (should never be called)
    int manhattan = Math.abs((int) (from.getX()-to.getX())) + Math.abs((int)
        (from.getY()-to.getY()));
    if(manhattan > 1) return ERROR;

    // if same coordinates, then the robot does not move
    if(from.getX()==to.getX() && from.getY()==to.getY()) return NO_MOVE;

    // calculate angle and rotate in the direction of target node
    double angle = Robot.getRotation(from, to, heading);
    boolean pathExists = rotateRobot(0.5*angle, 0.7*angle);

    // obstacle detected right after turning, or no path found
    if(MazeRunner.grid.isObstacle(from, to) || !pathExists) {
        return NO_PATH;
    }

    // otherwise, begin following the line with initial result as success
    int result = SUCCESS;
    result = followLine(result);

    // once followLine exits, stop the robot and return its final result
    pilot.stop();
    return result;
 }
```