Maze Solving Robot Assignment

ME30295: Electronics, Signals and Drives

Harry Mills: **12245**

Summary

This report details the process of writing and testing code to drive a maze solving robot. A Lego Mindstorms NXT robot programmed with RobotC code was used in this assignment. Light sensors pointing at the floor allowed the robot to follow a black line to not hit the walls of the maze, while ultrasonic distance sensors on the front and left of the robot detected the presence (or lack of) walls in the maze at certain points.

A program was successfully written that used the left-hand to autonomously drive the robot from the start to the finish of the maze. Furthermore, using the recorded path, the shortest return route to the start of the maze has been calculated. This has been implemented in pseudo-code as an example of how this would be completed. The shortest return route is calculated by using an algorithm to simplify the recorded route until it can no longer be simplified; at which point the shortest return route has been calculated.

Contents

1. Introduction

This report details the writing and testing of code to solve a maze using a Lego Mindstorms NXT robot. Any 2D maze can be solved using the left-hand rule and it is very simple algorithm to implement; if you can turn left, turn left.

Also described in this report is the method for calculating the shortest return path through the maze after reaching the finish via the left-hand rule. This was done using an iterative algorithm which removes redundancies in the recorded robot path until it can be simplified no further.

Figure 1 shows the NXT robot and the construction of the maze. Black tape in a grid creates crossroads, or nodes, that the robot can detect with the light sensors as it drives through the maze following the tape. Figure 2 shows the physical offset of sensors on the robot which must be accounted for when calculating turning and sensing walls.

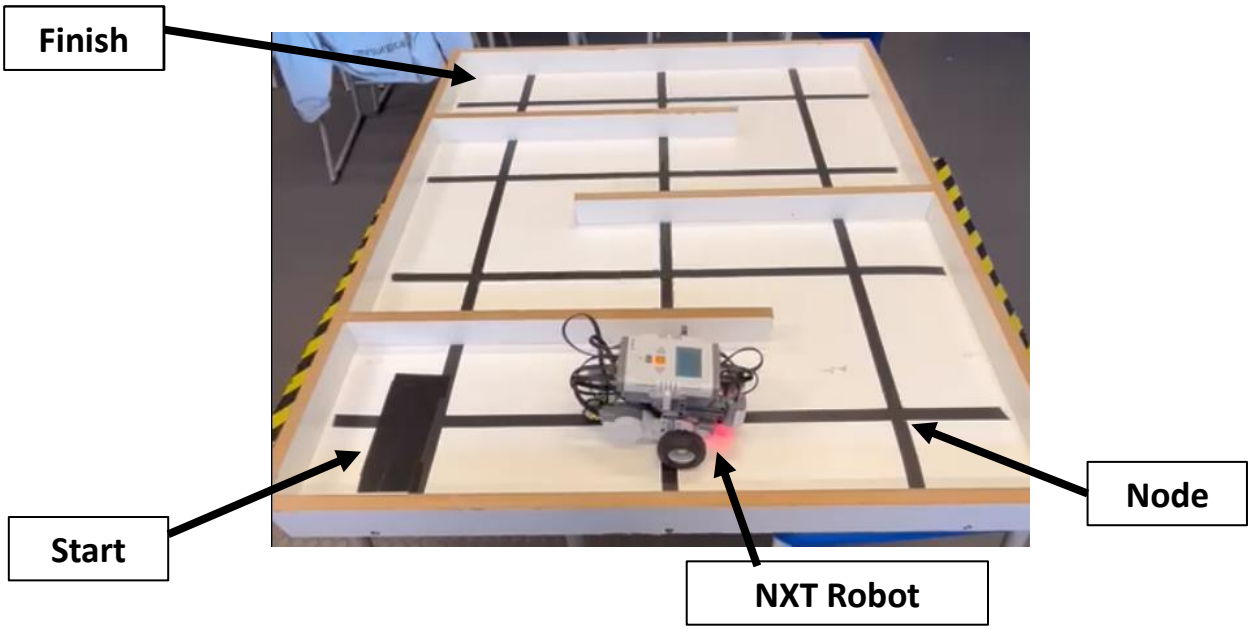

Figure 1 Construction of the maze solved in this task

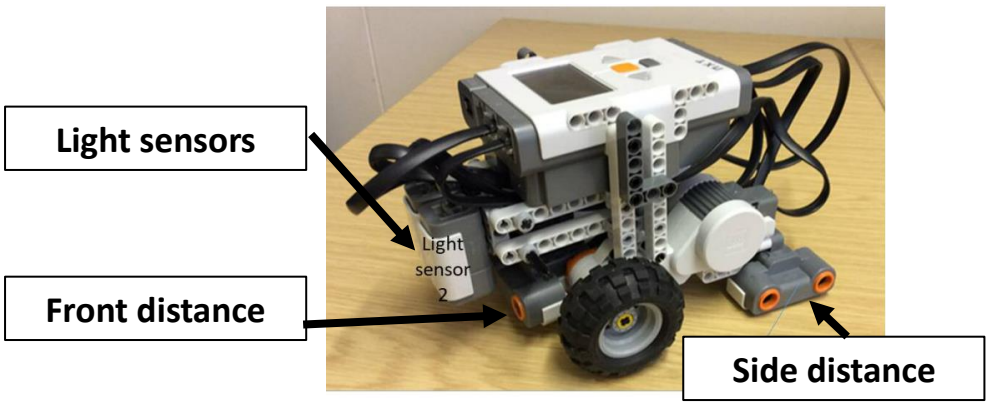

Figure 2 Relative position of sensors on the NXT robot. Note how the left distance sensor is located at the back of the robot

2. Maze Solving Task

Figure 3 shows the flow diagram logic to solve the maze using the left-hand rule. This is very simple in operation and follows the process: drive to a node, turn left, if not possible go forward, if left and forward are not possible then turn right. The robot will turn right until there is not wall in front of it – allowing it to u-turn out of dead-ends inside the maze.

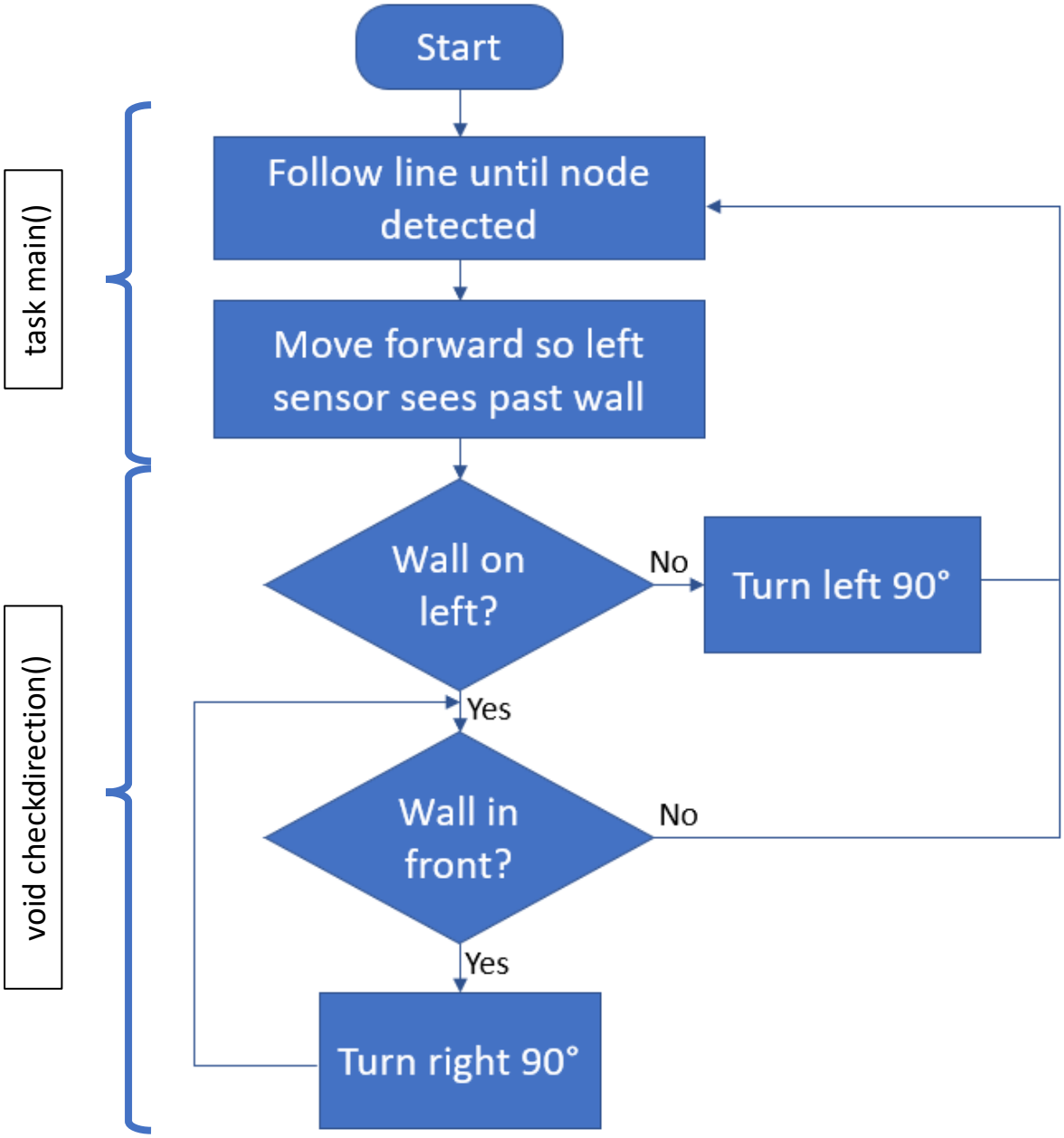

Figure 3 Flow diagram for solving a maze using the left-hand rule

1.1. Line Following

The first task was to make the robot follow the lines on the floor of the maze by keeping the line between the two light sensors on the front of the robot. This was achieved using the 3 lines of code shown in FIG.

```
readsensors();
motor(leftmotor) = (leftlight - rightlight/2 - 2);
motor(rightmotor) = (rightlight - leftlight/2 - 2);
```

Figure 4 Line following code

Line following is achieved by calculating the difference between the readings of the left and right light sensors. If the left light is over the line, the robot is too far to the right and the left light sensor will read lower than the right sensor. Vice versa for the right light.

Figure 5 shows visually how this proportional difference correction operates. The difference between the values of the light sensors is used to drive the motors. If the right light value is lower (bottom case Fig 5) then the left motor will be driven forwards faster than the right motor in an attempt to reach equilibrium. The larger the difference between the light values, the faster the robot will correct to the line.

Practically, it was useful to divide the light value being subtracted in Figure 4 as this prevented any negative speeds being sent to the wheel motors. This would have made the robot very jittery and slow to move around the maze. This effectively tunes the proportional control gain of the line following so it is always moving forward with less severe corrections when it deviates by a small amount.

1.2. Light Threshold

In order for the line following described to stop at each node and check the sensors to detect walls (Figure 3) a threshold between black and white on the maze floor is required.

The readings between the 2 light sensors were offset slightly as one always read slightly lower on the same surface. Reading light values from the NXT display showed that when over white the light sensors outputted consistently around 50, and 30 when over black.

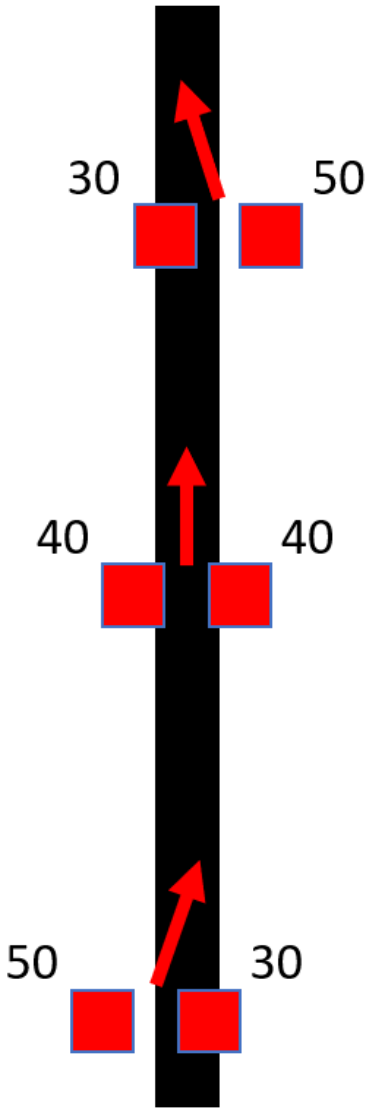

Figure 5 Line following logic tries to keep both light sensors at the same level

```
if ((leftlight < lightthreshold)&&(rightlight < lightthreshold))
            // If the Light Sensor reads a value less than 36:
{
      wait1Msec(1000);
      motor(leftmotor) = 0;
      motor(rightmotor) = 0;
      wait1Msec(500);
      checkdirection();
}
```

Figure 6 Node detection using light thresholds stops the line following code whenever the robot crosses a horizontal line

This threshold is used to break the loop of line following in task main() by running the code shown in Figure 6 whenever both light sensors cross the threshold – when both light sensors cross a line at the same time. This occurs whenever the robot reaches a node.

When the robot crosses a node, due to the physical location of the left sensor, the robot must drive forward a bit more until sensor passes the end of the wall. This happens because the left sensor is offset behind the light sensors – see Figure 7.

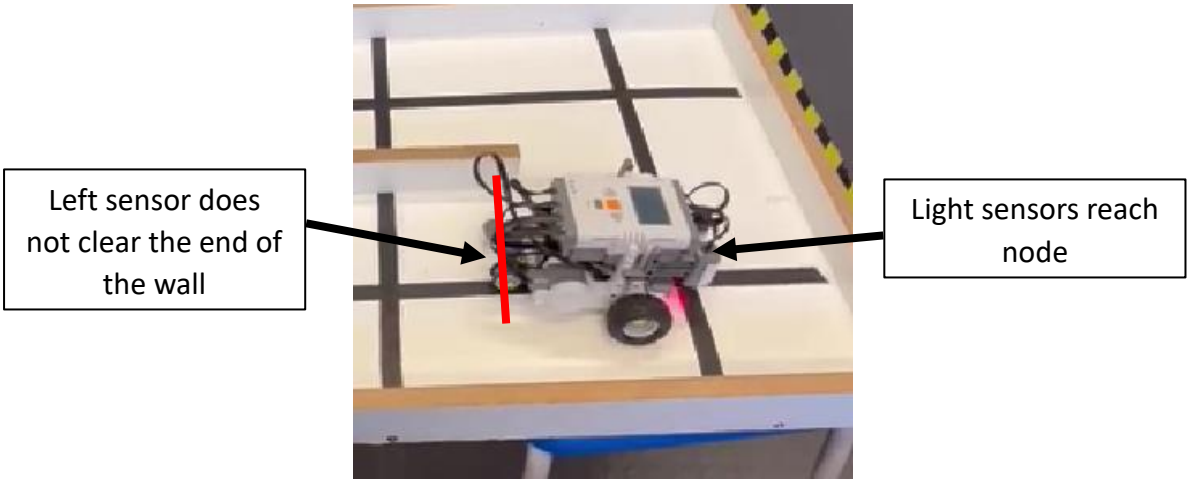

Figure 7 The left distance sensor does not clear walls that end when the light sensors reach the corresponding node

1.3. checkdirection()

When the robot crosses a node, the function checkdirection() is called. This is the second half of Figure 3 and both checks the sensors for walls around the robot, and then executes turns based on this.

```
void checkdirection()                                   // Checking whether there are walls to the side and to the front
{
     readsensors();

     if (sidedistance > sidethresh && sidedistance != 255)          //if there is no wall to the left (hardcoded for 255 bug)
     {
          motor(leftmotor) = -7;                        // turn left
          motor(rightmotor) = 7;
          wait1Msec(2500);
          while (rightlight > lightthreshold) // while right is seeing white
          {
               readsensors();
               motor(leftmotor) = -7;                   // turn left
               motor(rightmotor) = 7;
          }
          motor(leftmotor) = 0;
          motor(rightmotor) = 0;
          wait1Msec(500);
     }
```

Checks for wall on left – turns left if possible

Figure 8 The first part of checkdirection() checks the left distance sensor and turns left if possible

As per the flowchart in Figure 3, the first step in the left-hand rule algorithm is to check if there is a wall on the left, and then turn left if possible. Figure 8 shows this is done by firstly checking if the detected side distance is over the threshold (there is no wall to the left), so the robot can turn left. A catch included is that if the sensor output is 255 (the sensor maximum) then there is also a wall on the left. This occurs because the ultrasonic receiver is probably not receiving the sound waves sent out. The most common cause of this was because the sensor was too close to the wall.

The turning action code in Figure 8 turns the robot left (anti-clockwise) until the right light sensor reaches a line. The ensures that the line is between the light sensors so the robot can successfully continue following the line.

```
else if ( frontdistance < frontthresh || frontdistance == 255)       // if there is a wall in front (hardcoded for 255 bug)
{

     //motor(leftmotor) = -7;                                        // reverse for turning circle
     //motor(rightmotor) = -7;
     //wait1Msec(1000);

     //motor(leftmotor) = 0;                                         //pause
     //motor(rightmotor) = 0;
     //wait1Msec(500);

     motor(leftmotor) = 7;                                         // turn right
     motor(rightmotor) = -7;
     wait1Msec(2500);
     while (rightlight > lightthreshold) // while left is seeing white
     {
          readsensors();
          motor(leftmotor) = 7;                                    // turn right
          motor(rightmotor) = -7;
     }
     motor(leftmotor) = 0;
     motor(rightmotor) = 0;
     wait1Msec(500);

     checkdirection();

}
```

Reverse to avoid bumping walls while turning

Right turn then check to see if another right turn is needed

Figure 9 Forward sensor check in checkdirection()

Figure 9 shows the front sensor check that is run if there is a wall detected on the left. In this case it checks if there is a wall in front by comparing the front distance to the front threshold distance. Again, a catch for erroneous 255 readings due to sensor malfunction is included.

If the reading is below the threshold then there is a wall in front and the robot executes the code in the else if statement. Firstly, the robot reverses slightly. This is to prevent the left light sensor from hitting walls (note its position in Figure 2). Then the robot turns right (clockwise) until the left light sensor reaches a line – so that the line ends up between the light sensors when the turn is complete as with the left turn.

To finally complete the logic represented by the flow chart in Figure 3, this check must loop to allow the robot to perform a u-turn when at a dead-end. Calling checkdirection() at the end of this block immediately checks again and will perform another right turn if required.

Complete maze solving code is attached in the appendix.

3. Shortest Return Path

One the maze has been solved using the left-hand rule, an algorithm can be run on the stored movements to calculate the shortest return path to the start of the maze. This is done by removing redundant dead-ends in the maze path (left = L, right = R, forward = F):

1. Any u-turns in the path can be replaced with RR. (FRRF, RRRL or LRRR)
2. If consecutive LR or RL appear they can be deleted as they cancel out

A potential method of implementing this is represented by Figure 10 below. This uses a checking index to iterate through the sequence of moves found by solving the maze with the left-hand rule. If the next 2 moves are redundant (rule 2 above), they are skipped. If the next 4 moves constitute a u-turn they are represented by RR in the shortest path code, and the checking index skips these. Otherwise, the index of maze path being checked is added to the shortest path sequence until the checking index reaches the end of the left-hand rule maze path.

The algorithm shown below completes one iteration going through each move in the maze path. This would need to be run multiple times until no more simplifications can be made.

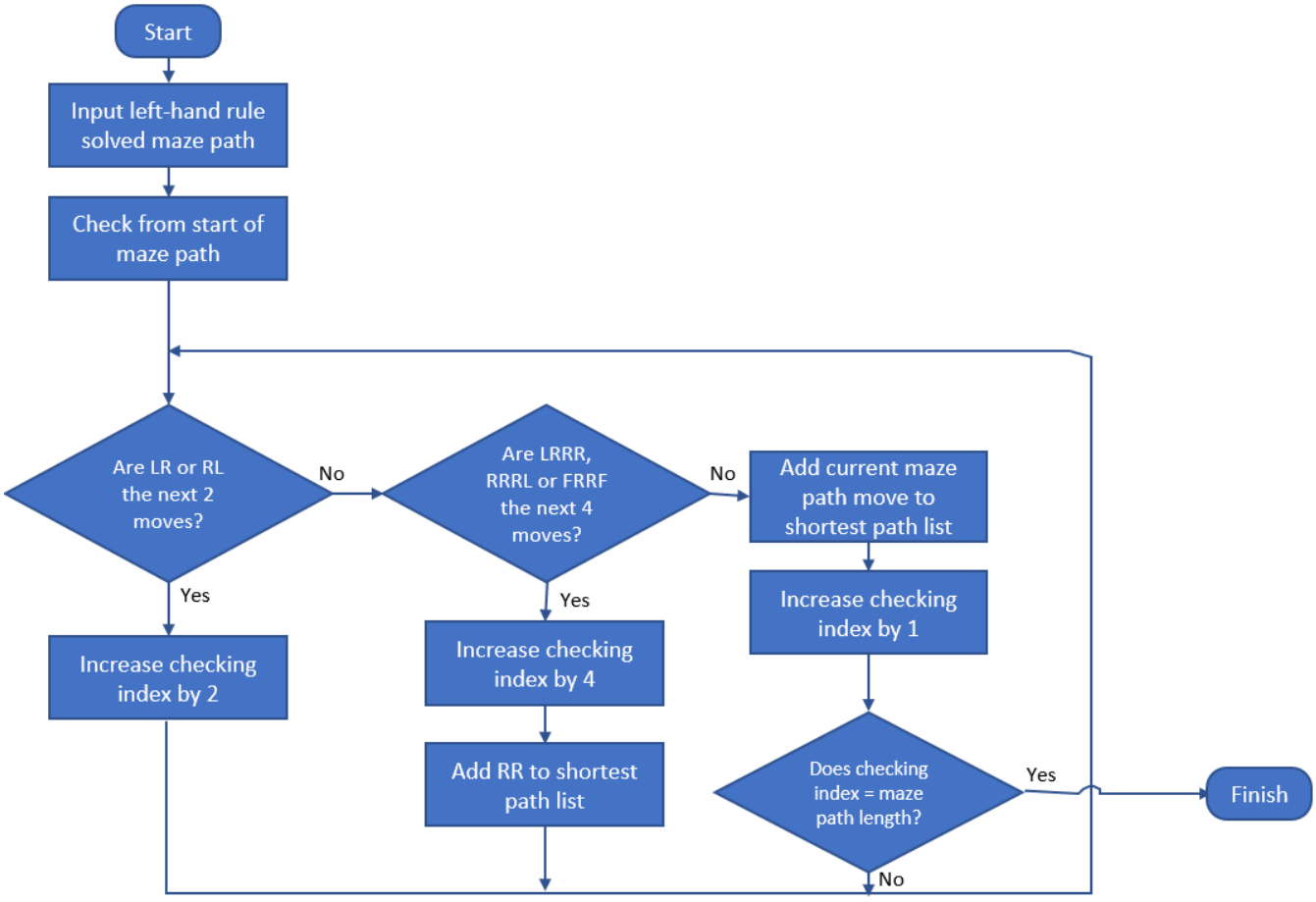

Figure 10 Method of iterating through the solved maze path using an algorithm to remove redundant movements to get the shortest path through the maze

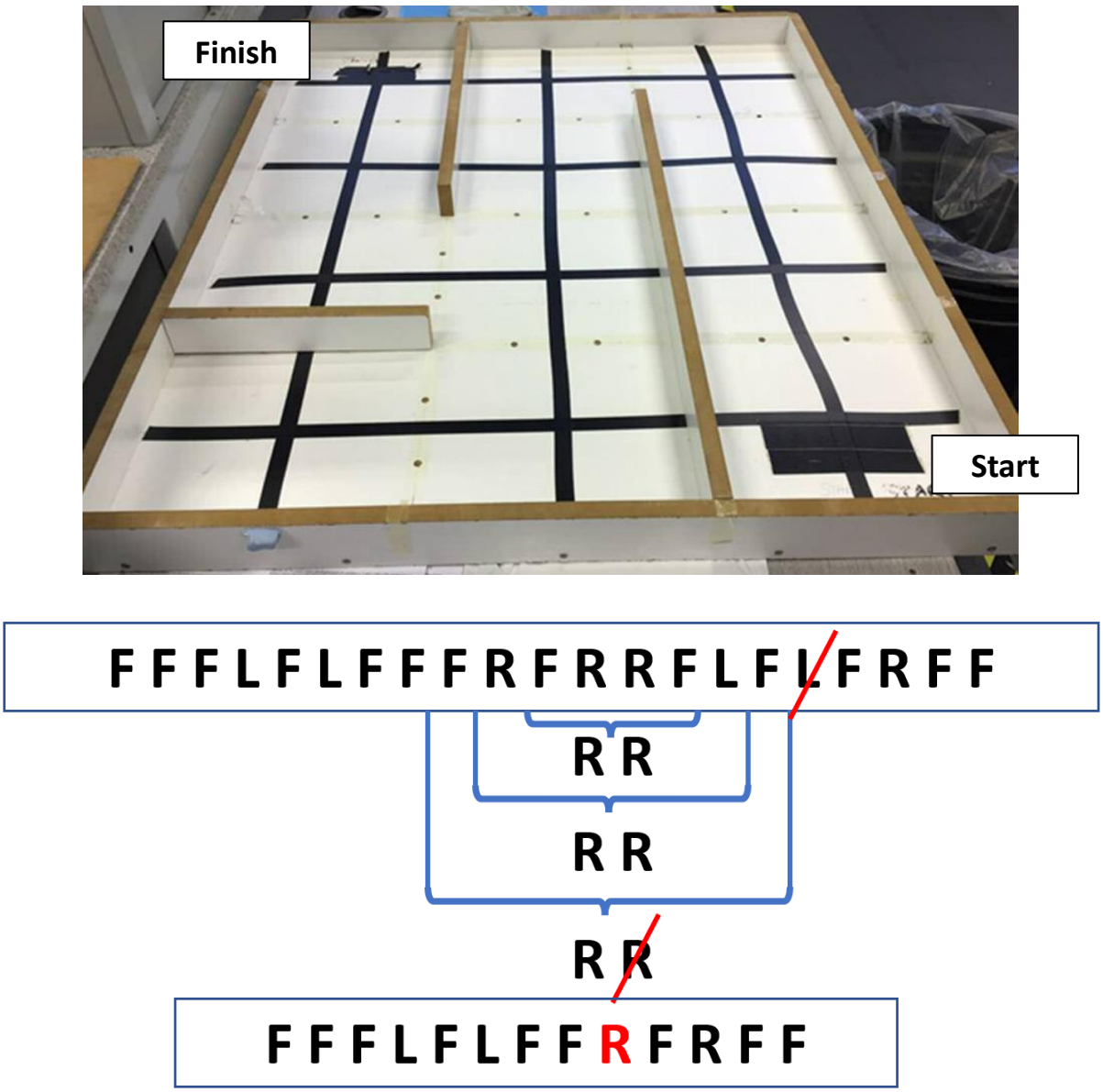

Figure 11 Top: Example maze. Bottom: Visual representation of how the algorithm described in Figure 10 works in practice on the maze shown.

Figure 11 demonstrates how the algorithm iteratively removes redundancies in the path until it can be simplified no more. To calculate the shortest <u>return</u> path the order is simply reversed, and L and R changed to their opposites:

F F L F L F F R F R F F F

Figure 12 Shortest return past for the maze shown in Figure 11

4. Discussion and Improvements

Implementation of the logic for the left-hand rule to solve was relatively simple. A straightforward algorithm (Figure 3) to check left, then forwards was all of the logic needed to drive the robot from the start to finish of the maze. However, practically implementing this so that the robot reliably followed the line, successfully turned and would continue following the line was more difficult to tune. Using the difference of the light sensor values the left and right wheels of the robot were driven proportionally to ensure the line was followed and any deviations quickly corrected. In this instance reliability was preferred over speed, although the proportional motor speeds could be tuned to achieve a faster maze solving speed,

Practically, the biggest issue was implementing the turning function of the robot, especially when turning right. The positioning of the left-distance (Figure 2) sensor meant it would jam against the walls of the maze when turning right – sometimes triggering the light sensors to stop turning prematurely. This was mitigated by adding a short reversing movement before any right turns to allow more space for the robot to turn.
This could also be fixed by moving the left-distance sensor further into the body of the robot, so it does not protrude.

Another issue found during testing was that shadows over the maze can strongly affect the light sensor readings. This can affect the line following accuracy or turning reliability. This could be solved by adding a small light on the underside of the robot to remove any shadows cast from the surrounding environment.

Finally, to achieve full autonomy a method of recognising the start and finish point of the maze is also required. This could be implemented by placing lights embedded in the floor of the maze so that when the robot drives over them the light sensors pick them up as brighter than the surrounding areas. This would trigger the robot to do a u-turn, calculate the shortest return path (Section 3), and execute these steps to reach the start. For this to work the lights embedded in the floor would need to be significantly brighter than the white maze floor.

5. Conclusion

This report has outlined the successful writing, implementation and testing of RobotC code to drive a Lego Mindstorms NXT robot around a maze using the left-hand rule to reach the finish. Furthermore, using an iterative simplification algorithm, it has been shown that the shortest return path from the finish to the start can be found for any sequence of moves.

The physical placement of the left-distance sensor cause issues when performing right turns. This was rectified by adding a short reverse before performing any right turns. Improvements for other issues such as eliminating shadows from surrounding objects, or a method of detecting the start and finish location of the maze have also been suggested.

Appendix

```
#pragma config(Sensor, S1,     sideSensor,     sensorSONAR)
#pragma config(Sensor, S2,     frontSensor,    sensorSONAR)
#pragma config(Sensor, S3,     rightLightSensor, sensorLightActive)
#pragma config(Sensor, S4,     leftLightSensor, sensorLightActive)
#pragma config(Motor,  motorA,          leftmotor,     tmotorNXT, PIDControl, encoder)
#pragma config(Motor,  motorB,          rightmotor,    tmotorNXT, PIDControl, encoder)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

/*-----------------------------------------------------------------------------------------------------*\
|*                                                                                                     *|
|*  This program reads the light sensor and follow a line materialised by a black tape on a white floor   *|
|*                                                                                                     *|
|*                                       ROBOT CONFIGURATION                                           *|
|*                                                                                                     *|
|*    MOTORS & SENSORS:                                                                                *|
|*    [I/O Port]              [Name]              [Type]             [Description]                     *|
|*    Port A                  leftmotor             NXT                Left motor                      *|
|*    Port B                  rightmotor            NXT                Right motor                     *|
|*    Port 1                  sideSensor          Sonar Sensor         side facing                     *|
|*    Port 2                  frontSensor         Sonar Sensor         front facing                    *|
|*    Port 3                  rightLightSensor    Light Sensor         floor facing                    *|
|*    Port 4                  leftLightSensor     Light Sensor         floor facing                    *|
\*-----------------------------------------------------------------------------------------------------*/

// global variables
int sidedistance = 0;     // Will hold the values read in by the side Sonar Sensor.
int frontdistance = 0;     // Will hold the values read in by the front Sonar Sensor.
int leftlight = 0;     // Will hold the values read in by the left Light Sensor.
int rightlight = 0;     // Will hold the values read in by the right Light Sensor.
int lightthreshold = 36;          // Define the threshold between black and white colors (white ~ 50; black ~ 30)// (maze 1 = 36)
int sidethresh = 29;           // Define the threshold for detecting a wall
int frontthresh = 31;

void displaysensors() {
      displayCenteredTextLine(0, "side distance");       /* Display side Sonar Sensor values */
      displayCenteredTextLine(1, "%d", sidedistance);   /* to LCD screen using %d.     */
      displayCenteredTextLine(2, "front distance");       /* Display front Sonar Sensor values */
      displayCenteredTextLine(3, "%d", frontdistance);   /* to LCD screen using %d.     */
      displayCenteredTextLine(4, "left light");       /* Display left Light Sensor values */
      displayCenteredTextLine(5, "%d", leftlight);   /* to LCD screen using %d.     */
      displayCenteredTextLine(6, "right light");       /* Display right Light Sensor values */
      displayCenteredTextLine(7, "%d", rightlight);   /* to LCD screen using %d.     */
      //wait1Msec(100);                                       // Only update the screen every 100 milliseconds.
}

void readsensors()
{
      sidedistance =      SensorValue(sideSensor);           // Store side Sonar Sensor values in 'sonarValueSide' variable.
      frontdistance =     SensorValue(frontSensor);         // Store front Sonar Sensor values in 'sonarValueFront' variable.
      leftlight =               SensorValue(leftLightSensor);       // Store left Light Sensor values in 'lightValueLeft' variable.
      rightlight =        SensorValue(rightLightSensor) - 8;  // Store right Light Sensor values in 'lightValueRight' variable and offset
      displaysensors();
}

void checkdirection()                                                   // Checking whether there are walls to the side and to the front
{
      readsensors();

      if (sidedistance > sidethresh && sidedistance != 255)            //if there is no wall to the left (hardcoded for 255 bug)
      {
            motor(leftmotor) = -7;                                          // turn left
            motor(rightmotor) = 7;
            wait1Msec(2500);
            while (rightlight > lightthreshold) // while right is seeing white
            {
                  readsensors();
                  motor(leftmotor) = -7;                                    // turn left
                  motor(rightmotor) = 7;
            }
            motor(leftmotor) = 0;
            motor(rightmotor) = 0;
            wait1Msec(500);
      }
      else if ( frontdistance < frontthresh || frontdistance == 255)       // if there is a wall in front (hardcoded for 255 bug)
      {

            //motor(leftmotor) = -7;                                                // reverse for turning circle
            //motor(rightmotor) = -7;
            //wait1Msec(1000);

            //motor(leftmotor) = 0;                                                     //pause
            //motor(rightmotor) = 0;
            //wait1Msec(500);

            motor(leftmotor) = 7;                                                   // turn right
            motor(rightmotor) = -7;
            wait1Msec(2500);
            while (rightlight > lightthreshold) // while left is seeing white
            {
                  readsensors();
                  motor(leftmotor) = 7;                                             // turn right
                  motor(rightmotor) = -7;
            }
            motor(leftmotor) = 0;
            motor(rightmotor) = 0;
            wait1Msec(500);

            checkdirection();

      }
}

task main()               // Line following function
{
      while(true)       // (infinite loop, also represented by while(1)).
      {
            readsensors();
            motor(leftmotor) = (leftlight - rightlight/2 - 2);
            motor(rightmotor) = (rightlight - leftlight/2 - 2);

            if ((leftlight < lightthreshold)&&(rightlight < lightthreshold))
                        // If the Light Sensor reads a value less than 36:
            {
                  wait1Msec(1000);
                  motor(leftmotor) = 0;       // leftmotor is run at a 0 power level.
                  motor(rightmotor) = 0;      // rightmotor is run at a 0 power level.
                  wait1Msec(500);
                  checkdirection();
            }
      }
}
```

Figure 13 Complete RobotC code for driving the Lego Mindstorms NXT robot around the maze using the left-hand rule