Arduino Robot Line Maze Solver

Carson Kirkpatrick

Jose Fernando Gonzales Del Valle Olazabal

James Beans

12/1/2014

CEN 4065 Software Architecture & Design

Instructor: Dr. Janusz Zalewski

Software Engineering Program

Florida Gulf Coast University

Ft. Myers, FL 33965

1. Introduction

The objective of this project is to create an Arduino robot car built from parts that will solve a line maze. Similar to a normal maze, a line maze is usually a set of connecting black lines on a white background, or vice-versa.

This project is challenging because much work is done at the intersection of hardware and software levels and it requires understanding of the relationship between the two. It also delves into many aspects of robotics programming and artificial intelligence. As embedded systems programming and cyber-physical systems become more and more relevant in today's world, this project becomes more important because it teaches much of the basic knowledge needed to do low-level programming, combine it with practical application, and venture into cognitive processes.

This project was originally going to expand the functionalities of a Boe-Bot Robot Kit already constructed by Austin Hughes [1]. However, for the purpose of learning robotics, Boe gave little-to-no room to experiment with new parts because the kit had been made with specific parts which could not be replaced. This makes it very hard, if not impossible, to add custom expansions for the robot outside of the add-on kits offered by Parallax. For this reason, and for academic reasons, it was decided that a new robot should be built without the use of a kit in order to have parts that can be used interchangeably, and to better learn the ins-and-outs of electronics involved in the robot-making process.

2. Project General Description

The robot will be controlled by an Arduino Uno board [2], pictured in Figure 1. The robot will move using a pair of 42mm x 19mm wheels that are powered by 75:1 Micro Metal Gear Motors [3], shown in Figure 2.

Figure 1 - Arduino Uno Board

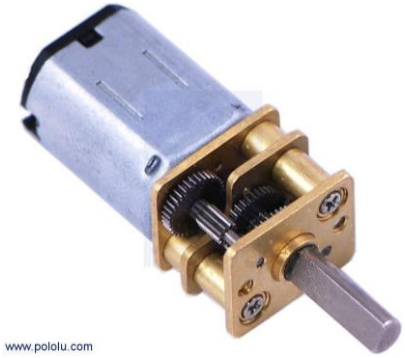

Figure 2 - 75:1 Micro Metal Gear Motor

The wheels will also have 42mm x 19mm wheel encoders, as shown in Figure 3, which will easily allow the robot to count the movements that it has made and feed this data back to the Arduino board.

Figure 3 - 75:1 Micro Metal Motors with 42mm x 19mm Wheel Encoders and 42mm x 19mm Wheels

The DC Motors will be fed power by the Qik 2s9v1 Dual Serial Motor Controller [4], shown in Figure 4, which will receive the driving instructions from the Arduino board and relay them to the motors.

Figure 4 - Qik 2s9v1 Dual Serial Motor Controller

The Arduino board will receive the data from the sensor and encoders in order to let the motor controller know which way to steer the wheels. Lastly, there will be a ⅜ inch metal ball encased on ball caster, shown in Figure 5, which will balance out the robot.

Figure 5 - ⅜ inch metal ball encased on ball caster

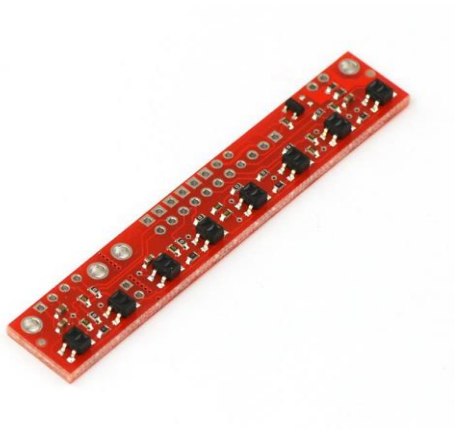

Figure 6 - QTR-8A Reflectance Sensor Array

Information about the maze will be acquired by a QTR-8A Reflectance Sensor Array [5], shown in Figure 6. The sensor has eight LED lights and eight reflectivity sensors. When the sensor has power, the LED lights are powered on, and the reflectivity sensors begin collecting data. For this project, the sensor will be facing downwards, looking directly at the line maze. The data from the sensor array will be used to make sure the robot stays on the lines of the line maze.

The robot will solve the maze by running a program uploaded to the Arduino board. The program uses the input from the reflectance sensors array to determine how the motors should run so that the robot stays on the line of the line maze. The main function of the program is to implement an algorithm that will have the robot solve any line maze.

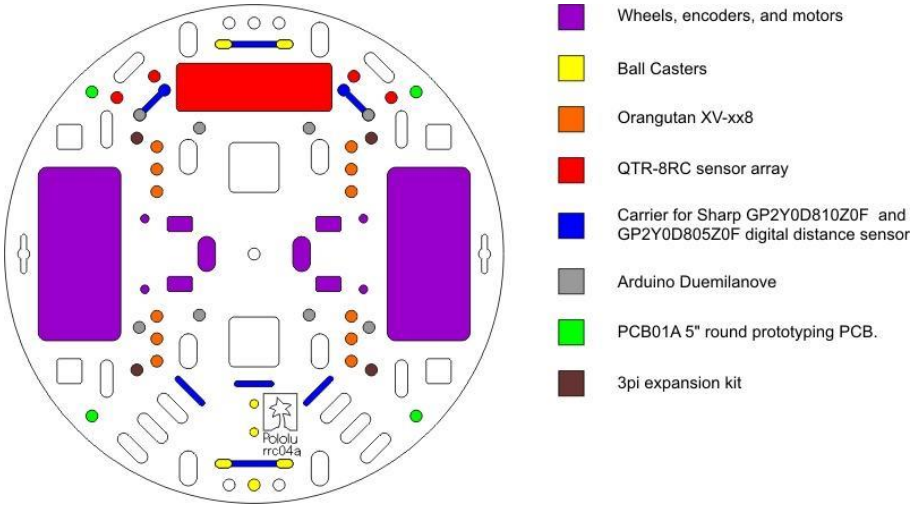

Figure 7 - 5” Robot Chassis RRC04A Transparent Clear

All the previously mentioned parts will be mounted on a 5” Robot Chassis RRC04A Transparent Clear [6] (Figure 7) that will act as the body of the car. Because of the size and number of the parts, two chasses will be used; one will be mounted on top of the other so that there will be two levels to the car.

3. Problem Definition

Our objective is to create a robot that can solve a line maze by itself. Similar to a normal maze, a line maze is usually a set of connecting black lines on a white background, as shown in Figure 8. Each line has a starting point and a finishing point. The robot is expected to follow the lines and its way from the start to the finish. However, the robot cannot traverse the maze without first taking a number of wrong turns. Line maze solver robots are usually part of competitive programming challenges with varying sets of rules.[7][8] The robot will be a wheeled robot with an Arduino board attached to it in order to give it simplistic intelligence.

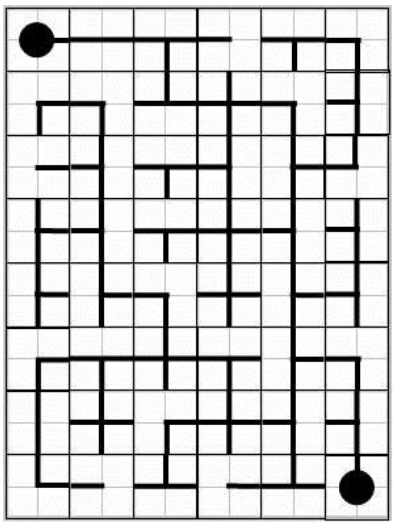

Figure 8 - Line Maze Example

The steps that are planned to complete the project are as follows:

1. Connect the motor controller to the Arduino board and test it to ensure that it is working properly.
2. Connect the motors to the motor controller and test them to ensure they are working properly.
3. Feed the battery to the motor controller in order to power the DC motors.
4. Connect the reflectance sensor array to the Arduino board and test to ensure it is working properly.
5. Connect the wheel encoders to the Arduino Board in order to track the number of times the wheel has rotated.
6. Mount all parts to the chassis so the car can move independently.

7. Develop the program that uses the data from the reflectance array sensor to have the robot solve a line maze by moving from the start point to the end point without getting stuck.
8. Test the program thoroughly with many different mazes of varying complexity to ensure that the robot can solve any maze imaginable.

The Physical Diagram of the project is shown in Figure 9.

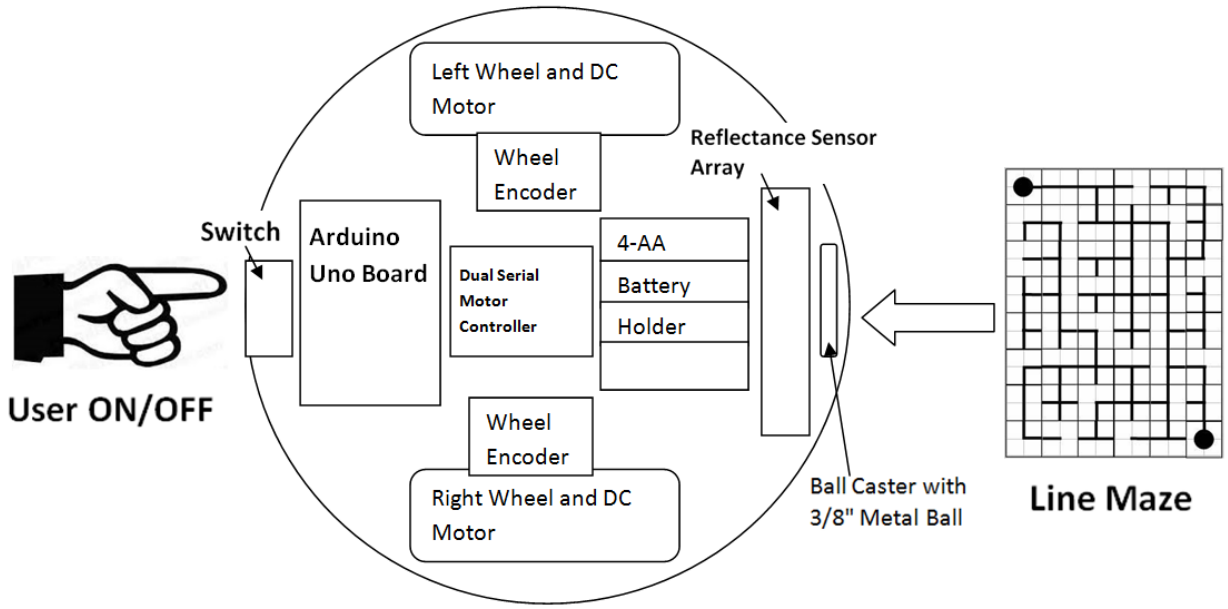

Figure 9 - Physical Diagram for the Arduino Maze Solver

In the simplest terms, the user starts the Arduino Maze Solver Robot, by turning on the switch shown in Figure 9. Once started, the robot shall begin receiving the reflectivity data from the colors of the lines in the line maze. It will gather the color differences in order to stay in the line when solving the maze.

The Context Diagram for the Software Controller of the Arduino Maze Solver Robot is shown in Figure 10.

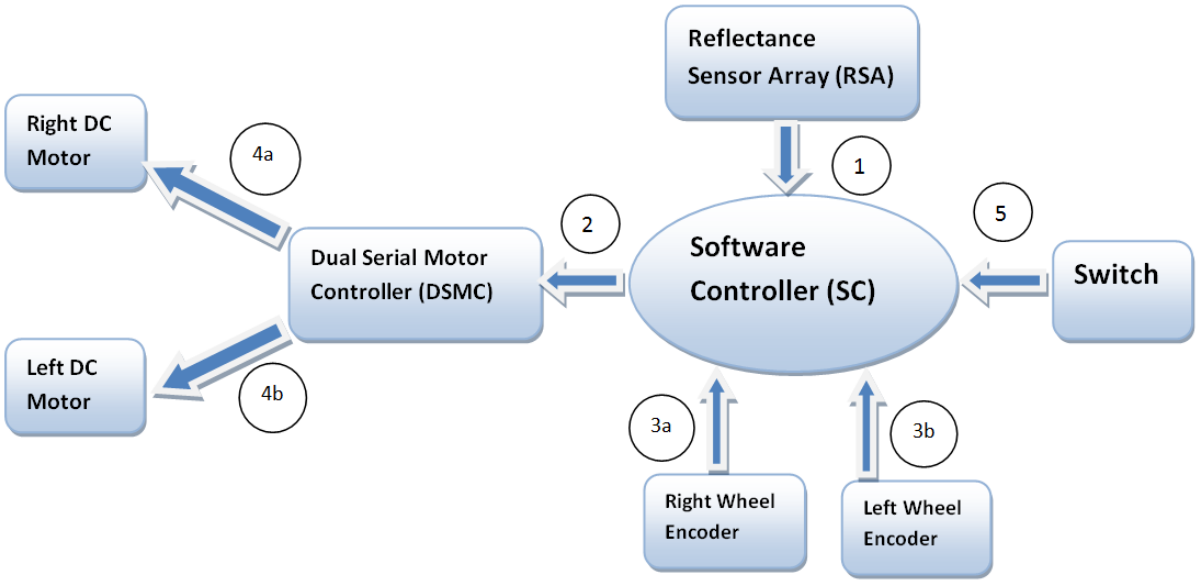

Figure 10 - Context for the Software Controller

Connections shown in Figure 10 reflect the following data transfers:

1. The Reflectance Sensor Array sends six analog streams of data to the Software Controller.
2. The Software Controller sends instructions to the Dual Serial Motor Controller in order to move the motors to keep the robot in the line.
3. The Dual Serial Motor Controller starts the right or left DC motor in order to move the robot.
4. The right and left wheel encoders tell the Software Controller the amount of times the right and left wheel has spun, respectively.
5. The Switch tells the Software Controller to start the motors and begin running a maze.

The general functionality of the device can be described as follows. The Reflectance Sensor Array (RSA) shall send the program six streams of reflectivity data. The data will be in the form of six bits, 000000 to 111111. Each bit represents the reading of one of the reflectivity sensors: 0 means the sensor reading is below the preset threshold, indicating a light-colored area, and 1 means the sensor reading is above the preset threshold, indicating a dark-colored area. The Software Controller (SC) shall process these data and tell the Dual Serial Motor Controller

(DSMC) which motor to start and which direction to go. The right and left wheel encoders shall tell the SC how far the robot has advanced. Lastly, the DSMC will signal the motors to start or stop.

More precisely, the input/output requirements for the Software Controller are defined as follows:

1. The SC shall handle six analog streams of data from the RSA, each in the range from 0 to 1000 depending on the reflectivity
2. The SC shall read the streams of data from the RSA at least once every 50 ms.
3. The SC shall signal the Dual Serial Motor Controller which motor to left or right to power upon receiving data from the RSA.
   *Note. The algorithm to do this is a part of the design.

3a. The SC shall interpret the data from the right wheel encoder, which tells it how far the right wheel has moved in terms of the angle.

3b. The SC shall interpret the data from the left wheel encoder, which tells it how far the left wheel has moved in terms of the angle.

4. The SC shall start the motors to begin running the maze when a designated button is pressed.
5. The SC shall stop the motors when it has determined that it has arrived at the end of the maze.
6. The SC shall stop the motors if it encounters an unexpected obstacle.

Safety is a very critical feature in the software design phase, our Arduino Line Maze Solver Robot will meet certain safety and security requirements.
The robot will be able to follow the line in order to not steer off the prepared path, this will make sure that it does not collide with any outside objects or fall off the given path. The robot will also maintain a slow but steady speed, in order to not harm any observer, should the robot lose control. These measures will insure, the robot has a high degree of safety.
Next, the robot will also meet security requirements in order to ensure that the system is secured from any threats. In terms of security, the robot is in a privileged position, for it is an isolated system (no internet access), and receives no use input. The only way an attacker, could commit an assault to the system would be for him to physically, connect to the system itself via USB and

override the program. Luckily, the project team members will maintain a vigilant eye on the robot's wellbeing. They have been especially trained to handle any type of saboteurs should the situation call for it.

Lastly, the robot must have a high degree of reliability on both the system as a whole as well as its individual components. The individual pieces must work at all times in order for the robot to work properly, if any one of the pieces mentioned in the general description were to fail, the robot will begin to malfunction and render itself unable to perform the task for which it was designed for. Lastly, all individual pieces must be reliably able to work together in order for the robot to have coordinated movement. Otherwise, the robot would not be able to stay in the line, since the data would not accurately represent its, real-time environment.

4. Design Description

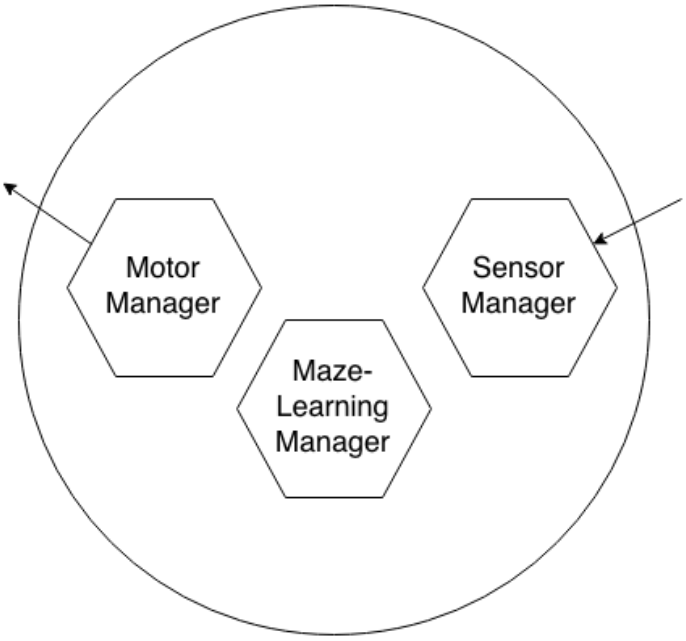

Figure 11 - Architecture of the Software Controller

The Architecture of the Software Controller illustrated follows the I/O transfer on the Context Diagram (Fig. 10) and is shown in Figure 11. The Motor Manager outputs the commands to control the motors' speed and the car's direction. The Maze-Learning Manager learns the mazes that robot runs so that when it runs the same maze again, it will use the correct path to complete the maze. The Maze-Learning Manager unit also acts as the heart of the software organizing its operation and controlling other components: the Motor Manager, Sensor Manager, and Maze-Learning Manager. The Sensor Manager handles the input from the reflectance sensor array, the motor encoders.

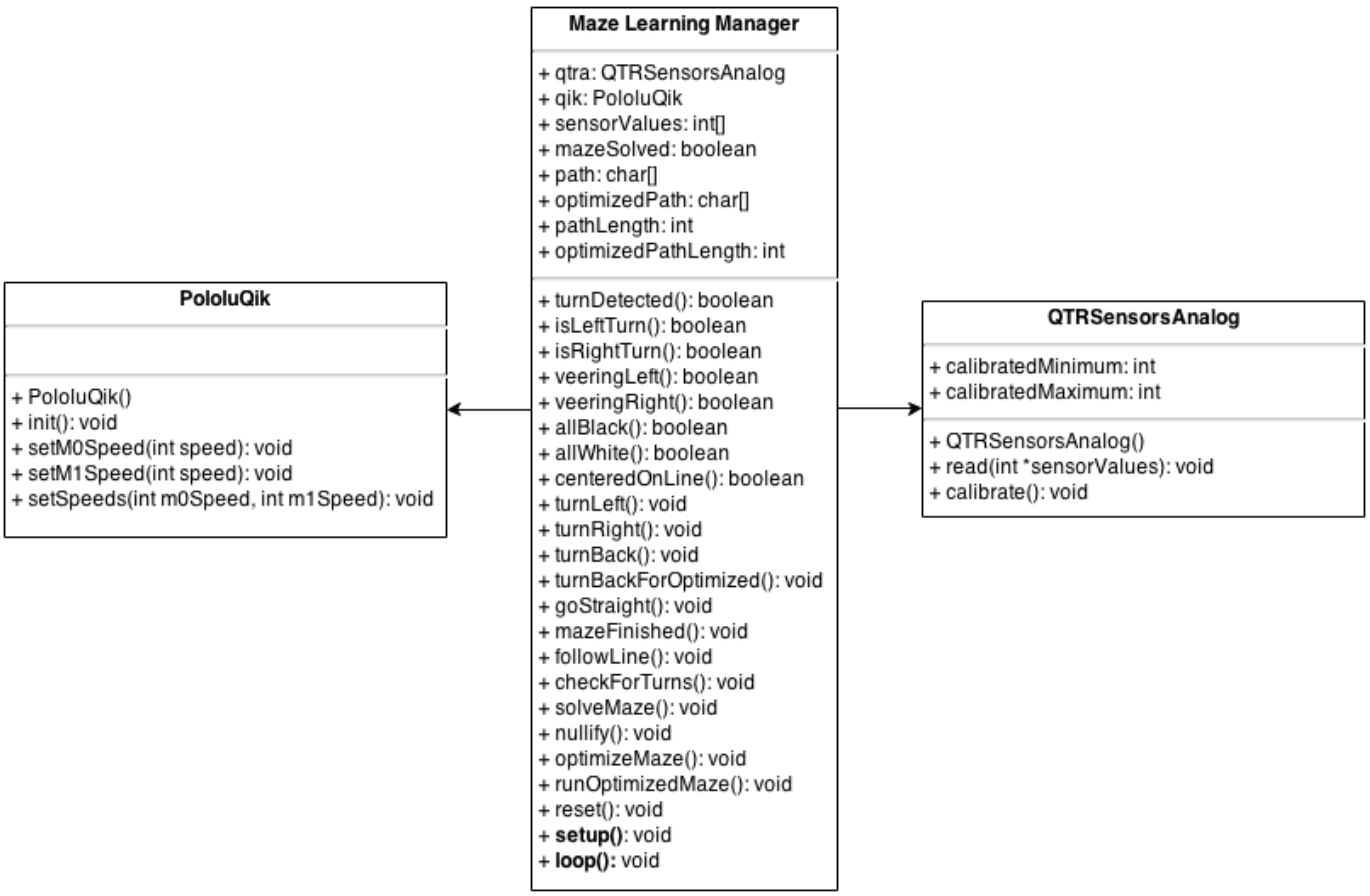

Figure 12 - Detailed Design of the Software Controller

The detailed design is composed of a main class and two classes that help control the motors and get data from the reflectance sensor. `PololuQik` is a class that uses the Qik2s9v1 Dual Serial Motor Controller to control the speed and direction of the motors. `QTRSensorsAnalog` is a class that reads the data from the QTR-8A Reflectance Sensor Array. Both of these classes are provided as libraries by Pololu Inc. [9].

The Maze Learning Manager has the robot solve the maze and then optimize the path that it took to solve the maze in the most efficient way possible. This class has many functions: the functions that deal with the reflectance sensor using the qtra object will be referred to as the Sensor Manager Code and the function that deal with controlling the motors using the qik object will be referred to as the Motor Manager Code. Everything else is part of the Maze Learning Code.

5. Implementation and Testing

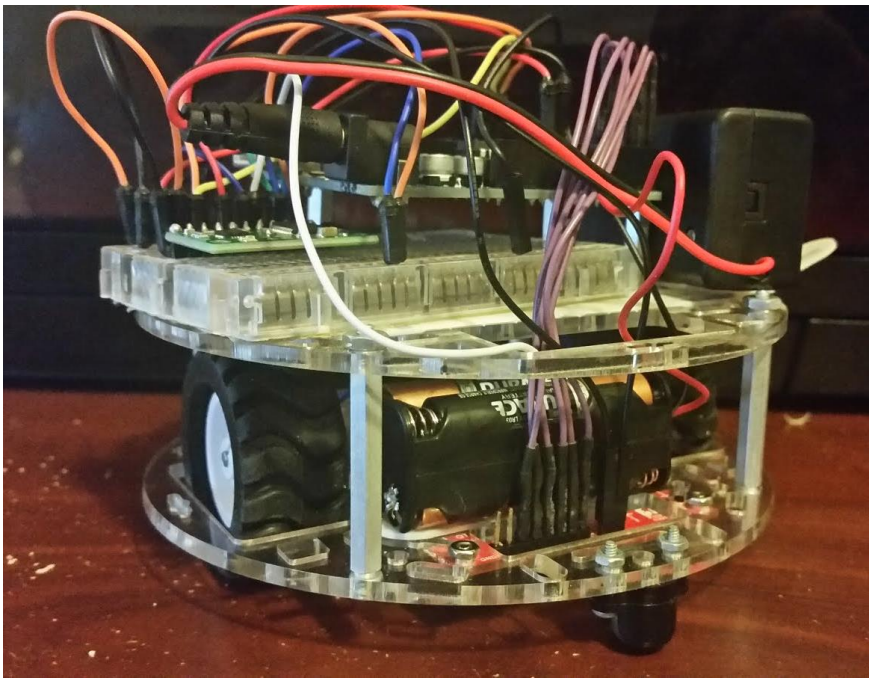

Figure 13 - Arduino Line Maze Solver Version 2.

Before any programming can be done, in the tracking, solving and learning algorithms, the robot must be in a state where it can move without the assistance of a computer connection. Figure 13, illustrates the second version of the robot. In this version, the robot now moves as a singular unit and implementation of the classes shown in Figure 12 can begin.

5.1 Sensor Manager Code

To solve a maze, the robot must first be able to see the maze. The sensor manager code uses the `QTRSensors` library to get data from the QTR-8A Reflectance Sensor Array, which detects the reflectivity of the surface it is facing. It also has functions that are used to help solve a maze. The sensor manager code is shown below, in Figure 14.

```
#include <QTRSensors.h>

#define numberOfSensors 6
#define samplesPerSensor 4

//thresholds for determining colors
#define blackThreshold 400
#define whiteThreshold 100
```

```
//sets up the sensor array
QTRSensorsAnalog qtra((unsigned char[]) {0, 1, 2, 3, 4, 5},
   numberOfSensors, samplesPerSensor, QTR_NO_EMITTER_PIN);

//the array where the sensor values are stored
unsigned int sensorValues[numberOfSensors];


boolean turnDetected() {
 qtra.read(sensorValues);
 return sensorValues[0] > blackThreshold || sensorValues[5] >
blackThreshold;
}

boolean isLeftTurn() {
 qtra.read(sensorValues);
 return sensorValues[0] > blackThreshold;
}

boolean isRightTurn() {
 qtra.read(sensorValues);
 return sensorValues[5] > blackThreshold;
}

boolean veeringLeft() {
 qtra.read(sensorValues);
 return sensorValues[2] < blackThreshold && sensorValues[3] >
blackThreshold;
}

boolean veeringRight() {
 qtra.read(sensorValues);
 return sensorValues[3] < blackThreshold && sensorValues[2] >
blackThreshold;
}

boolean allBlack() {
 qtra.read(sensorValues);
 return sensorValues[0] > blackThreshold && sensorValues[1] >
blackThreshold &&
        sensorValues[2] > blackThreshold && sensorValues[3] >
blackThreshold &&
        sensorValues[4] > blackThreshold && sensorValues[5] >
blackThreshold;
}

boolean allWhite() {
 qtra.read(sensorValues);
```

```
 return sensorValues[0] < whiteThreshold && sensorValues[1] <
whiteThreshold &&
        sensorValues[2] < whiteThreshold && sensorValues[3] <
whiteThreshold &&
        sensorValues[4] < whiteThreshold && sensorValues[5] <
whiteThreshold;
}

boolean centeredOnLine() {
 qtra.read(sensorValues);
 return sensorValues[0] < whiteThreshold && sensorValues[1] <
whiteThreshold &&
        sensorValues[2] > blackThreshold && sensorValues[3] >
blackThreshold &&
        sensorValues[4] < whiteThreshold && sensorValues[5] <
whiteThreshold;
}
```

Figure 14 - Sensor Manager Code

The code starts out by defining some constants. `numberOfSensors` is the number of sensors from the sensor array that will be used. `samplesPerSensors` is the number of samples taken per sensor for each sensor reading. The `blackThreshold` and `whiteThreshold` are values that are used by the functions to determine whether a reading is “white” or “black”. When testing the sensor, it was found black electric tape reads a value of about 500, and that white poster board reads about 50, hence 400 is used for `blackThreshold` and 100 is used for `whiteThreshold`.

The `QTRSensorsAnalog` constructor sets up the `qtra` object, which has the `read()` function that is used to read the sensor values. It takes four arguments: an array of the analog pins that will be used, the number of sensors being used, the number of samples per second to be taken from the each sensor, and the emitter pin. The emitter pin is optional because it is generally used for testing the LEDs on the sensor; `QTR_NO_EMITTER_PIN` should be passed as the last parameter if the emitter pin is not being used.

`sensorValues` is an array to which value of each sensor is stored. The robot uses six sensors, where the leftmost sensor is `sensorValues[0]` and the rightmost sensor is `sensorValues[5]`. All of the functions in the sensor manager code use

`qtra.read(sensorValues)` before performing a Boolean comparison to ensure that the sensor values are up to date for the comparison; otherwise, unexpected results may occur.

The `turnDetected()` function checks if there is a left turn or a right turn and returns `true` if either a left turn or right turn is found, or `false` if neither is found. The `isLeftTurn()` function returns `true` if there is a left turn and `false` if there is not a left turn. The `isRightTurn()` function returns `true` if there is a right turn and `false` if there is not a right turn. The `veeringLeft()` function returns `true` if the robot is starting to veer off of the black line to the left and `false` if the robot is following the center of the black line. The `veeringRight()` function returns `true` if the robot is starting to veer off of the black line to the right and `false` if the robot is following the center of the black line. The `allBlack()` function returns `true` if all the sensors are reading black and `false` if one or more is not reading black. The `allWhite()` function returns `true` if all the sensors are reading white and `false` if one or more is not reading white. The `centeredOnLine()` function returns `true` if the two left sensors are reading white, the two right sensors are reading white, and the middle two sensors are reading black; it returns `false` otherwise.

5.2 Motor Manager Code

The motor manager code has functions for turning the robot by controlling the motors using the PololuQik class. The motor manager code is shown below, in Figure 16.

```
#include <PololuQik.h>

//the motor speeds
#define M0Speed 30 //left motor
#define M1Speed -30 //right motor

//sets up the motor controller
PololuQik2s9v1 qik(2, 3, 4);


void turnLeft() {
 qik.setM0Speed(-M0Speed);
 qik.setM1Speed(M1Speed);
```

```
 //if the turn is started on a black line, it needs to get off the
line first
 while (centeredOnLine());
 delay(50);
 while (!centeredOnLine());
 qik.setM0Speed(M0Speed);
 path[pathLength] = 'L';
 pathLength++;
}

void turnRight() {
 qik.setM0Speed(M0Speed);
 qik.setM1Speed(-M1Speed);
 //if the turn is started on a black line, it needs to get off the
line first
 while (centeredOnLine());
 delay(50);
 while (!centeredOnLine());
 qik.setM1Speed(M1Speed);
 path[pathLength] = 'R';
 pathLength++;
}

void turnBack() {
 qik.setM0Speed(-M0Speed);
 qik.setM1Speed(M1Speed);
 //if the turn is started on a black line, it needs to get off the
line first
 while (centeredOnLine());
 delay(50);
 while (!centeredOnLine());
 qik.setM0Speed(M0Speed);
 path[pathLength] = 'B';
 pathLength++;
}

void turnBackForOptimized() {
 qik.setM0Speed(-M0Speed);
 qik.setM1Speed(M1Speed);
 while (centeredOnLine());
 delay(50);
 while (!centeredOnLine());
 delay(50);
 while (centeredOnLine());
 delay(50);
 while (!centeredOnLine());
 qik.setM0Speed(M0Speed);
}
```

```
void goStraight() {
 qik.setM0Speed(M0Speed);
 qik.setM1Speed(M1Speed);
 path[pathLength] = 'S';
 pathLength++;
}
```

Figure 15 - Motor Manager Code

The `qik` object is setup with `PololuQik2s9v1 qik(2, 3, 4);`. The three parameters are the transmit pin of the motor controller (TX), the receiving pin (RX), and the reset pin (RESET), respectively. In this program, only RX is actually used because only the motor controller is receiving data and it never needs to be reset with the reset pin.

Two functions from the `PololuQik` class are used: `setM0Speed(int speed)` and `setM1Speed(int speed)`. These two functions control the speeds of the two motors connected to the Qik motor controller. They take values between `-127` and `127` as a parameter; positive values will make the motor turn forward and negative values will make the motor turn in reverse.

The speeds that the motors will run at are defined by the constants `M0Speed` and `M1Speed`. For the purposes of this project, the speeds are set to a fairly low speed of `30` because we found that the robot would run mazes slightly more reliably at lower speeds. The reason that `M1Speed` is negative is because the M1 motor is mirrored to the M0 motor. Therefore, for both motors to run in the same direction, one motor must go in reverse.

All of the functions in the motor manager code are functions that have the robot turn in a certain way, with the exception of the `goStraight()` function. They are all decisions that allow the robot to traverse a maze, such as the one shown in Figure 16.

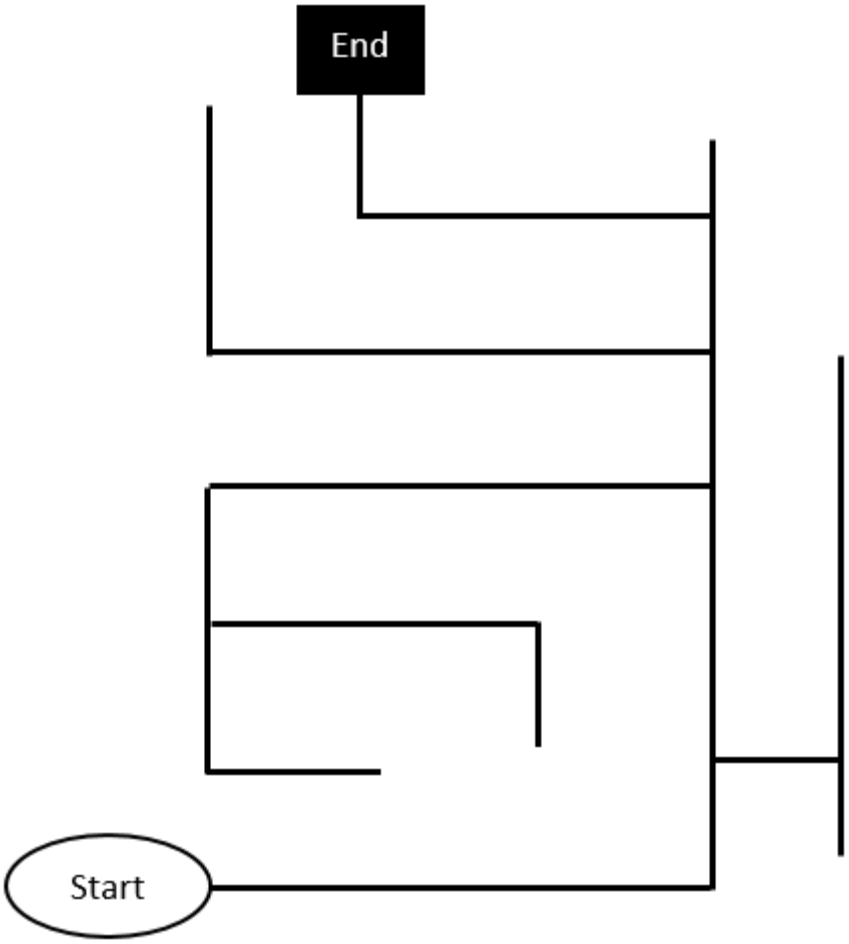

Figure 16 - Maze

The `turnLeft()` function makes the robot turn left. The function starts by reversing the speed of the left motor (M0), which causes the robot to rotate counterclockwise. The function then waits for the sensor to leave the black line, and then waits for it to find the black line again. When the black line is found again, the turn has been completed and the left motor is reset to its original speed.

The `turnRight()` function makes the robot turn right. The function is the same as the `turnLeft()` function except that the right motor's direction is reversed instead of the left motor.

The `turnBack()` function makes the robot turn back. The function is exactly the same as the `turnLeft()` function but it records a "back" turn instead of a left turn. This function is only called when the two middle sensors on the sensor array are detecting white. Therefore, it is ensured that this function is only called when there is actually a back turn.

The `goStraight()` function makes the robot go straight. The main purpose of this of function is to record a "straight" in the path array, as it is usually called when the robot is already going straight.

The `turnLeft()`, `turnRight()`, `turnBack()`, and `goStraight()` are all decisions that the robot makes and records into the `path` array. At the end of the maze shown in Figure 16 the robot will only see and remember the characters shown in Figure 17, which will be used later to optimize the path that the robot takes to solve the maze.

L S L L L R B L L L B R S R L L R B L L L R

Figure 17 - Visible path for the robot

The `turnBackForOptimized()` function is only used when the robot is running the maze after it has been optimized. It makes the robot turn back, but this will be further explained in the Maze Learning section.

5.3 Maze Learning Code

The maze learning code can be split into two main parts: solving the maze and optimizing the maze. First, the robot solves the maze using a maze-solving algorithm. It then optimizes the path that the robot took to solve the maze by eliminating the turns that leads to "dead ends". After the optimization is done, it has the robot run through the maze with the optimized path.

```
//delay times
#define checkTime 30
#define timeBeforeTurn 200

//max size of the path
```

```
#define pathSize 100

//set to true when the maze has been solved.
boolean mazeSolved;

//stores all the turns that the robot takes when it is solving the
maze
char path[pathSize] = {};

//stores the path after it has been optimized
char optimizedPath[pathSize] = {};

int pathLength;
int optimizedPathLength;

void mazeFinished() {
 mazeSolved = true;
 qik.setM0Speed(0);
 qik.setM1Speed(0);
 delay(2000);
 qik.setM0Speed(50);
 qik.setM1Speed(50);
 delay(3000);
 qik.setM0Speed(0);
 qik.setM1Speed(0);
}

void followLine() {
 while (veeringLeft()) {
   qik.setM1Speed(0);
 }
 qik.setM1Speed(M1Speed);

 while (veeringRight()) {
   qik.setM0Speed(0);
 }
 qik.setM0Speed(M0Speed);
}

void checkForTurns() {
 if (turnDetected()) {
   delay(checkTime);
   if (isLeftTurn()) {
     delay(timeBeforeTurn);
     if (allBlack()) {
       mazeFinished();
     } else {
       turnLeft();
     }
```

```
    } else if (isRightTurn()){
      delay(timeBeforeTurn);
      if (allBlack()) {
        mazeFinished();
      } else if (allWhite()) {
        turnRight();
      } else {
        goStraight();
      }
    }
  }

  if (allWhite()) {
    turnBack();
  }
}

void solveMaze() {
 qik.setM0Speed(M0Speed);
 qik.setM1Speed(M1Speed);
 while(mazeSolved == false) {
    followLine();
    checkForTurns();
 }
}

void nullify(char a[]) {
 for (int i=0; i<pathSize; i++) {
    a[i] = NULL;
 }
}

void optimizeMaze() {
 boolean changed = true;
 char a1[pathSize] = {};
 char a2[pathSize] = {};
 int tempLength = pathLength;
 optimizedPathLength = pathLength;
 int i = 0;
 int j = 0;

 //copies the path array
 for (i=0; i<pathLength; i++) {
    a1[i] = path[i];
 }

 while (changed == true) {
    changed = false;
    for (i=0, j=0; i<tempLength; i++, j++) {
```

```
  if (a1[i+1] != 'B') {
    a2[j] = a1[i];
  } else if (a1[i+1] == 'B') {

    if (a1[i] == 'L' && a1[i+2] == 'R') {
      a2[j] = 'B';
    } else if (a1[i] == 'L' && a1[i+2] == 'S') {
      a2[j] = 'R';
    } else if (a1[i] == 'R' && a1[i+2] == 'L') {
      a2[j] = 'B';
    } else if (a1[i] == 'S' && a1[i+2] == 'L') {
      a2[j] = 'R';
    } else if (a1[i] == 'S' && a1[i+2] == 'S') {
      a2[j] = 'B';
    } else if (a1[i] == 'L' && a1[i+2] == 'L') {
      a2[j] = 'S';
    }

    i+=2;
    optimizedPathLength -= 2;
  } //end if

} //end for
nullify(a1);
tempLength = optimizedPathLength;

for (i=0, j=0; i<tempLength; i++, j++) {
  if (a2[i+1] != 'B') {
    a1[j] = a2[i];
  } else if (a2[i+1] == 'B') {

    if (a2[i] == 'L' && a2[i+2] == 'R') {
      a1[j] = 'B';
    } else if (a2[i] == 'L' && a2[i+2] == 'S') {
      a1[j] = 'R';
    } else if (a2[i] == 'R' && a2[i+2] == 'L') {
      a1[j] = 'B';
    } else if (a2[i] == 'S' && a2[i+2] == 'L') {
      a1[j] = 'R';
    } else if (a2[i] == 'S' && a2[i+2] == 'S') {
      a1[j] = 'B';
    } else if (a2[i] == 'L' && a2[i+2] == 'L') {
      a1[j] = 'S';
    }

    i+=2;
    optimizedPathLength -= 2;
    changed = true;
  } //end if
```

```
    } //end for
    nullify(a2);
    tempLength = optimizedPathLength;
  } //end while

  for (i=0; i<optimizedPathLength; i++) {
    optimizedPath[i] = a1[i];
  }
}

void runOptimizedMaze() {
  int i = 0;
  qik.setM0Speed(M0Speed);
  qik.setM1Speed(M1Speed);

  while (i < optimizedPathLength) {
    followLine();
    if (turnDetected() || allWhite()) {
      delay(checkTime);
      if (optimizedPath[i] == 'L') {
        delay(timeBeforeTurn);
        turnLeft();
      } else if (optimizedPath[i] == 'R') {
        delay(timeBeforeTurn);
        turnRight();
      } else if (optimizedPath[i] == 'S') {
        delay(timeBeforeTurn);
        goStraight();
      } else if (optimizedPath[i] == 'B') {
        if (isLeftTurn()) {
          delay(timeBeforeTurn);
          turnBackForOptimized();
        } else {
          delay(timeBeforeTurn);
          turnBack();
        }
      }
      i++;
    } //end if
  } //end while

  while (!allBlack()) {
    followLine();
  }
  delay(timeBeforeTurn);
  mazeFinished();
}
```

```
void reset() {
 mazeSolved = false;
 nullify(path);
 nullify(optimizedPath);
 pathLength = 0;
 optimizedPathLength = 0;
}

void setup() {
 qik.init();
}

void loop() {
 reset();
 solveMaze();
 delay(5000);
 optimizeMaze();
 runOptimizedMaze();
 delay(5000);
}
```

Figure 18 - Maze Learning Code

5.3.1 Maze Solving Code

The first part of the maze learning code will have the robot solve a line maze using the maze-solving algorithm illustrated by the flowchart in Figure 19 below. The `solveMaze()` function is where the maze-solving algorithm is implemented. In `solveMaze()`, the motors are started, and the program enters a `while` loop that runs the `followLine()` function and `checkForTurns()` function until the Boolean variable `mazeSolved` is set to `true`.

When solving a maze, the robot must first be able to follow the black line so that is does not leave the maze completely. This is done with the `followLine()` function. This function has two while loops that use the `veeringLeft()` and `veeringRight()` functions to check is the robot is veering off of the line. If the robot is veering to the left, the right motor is stopped until the robot is no longer veering left. The same goes for if the robot is veering to the right.

Most of the maze-solving algorithm is implemented in the `checkForTurns()` function. Simply put, when a turn is detected, the robot will always take a left turn if possible. If there is no left turn, it will continue going straight if possible. If there is neither a left turn nor a straight

path, then the robot will make a right turn. To check to see if there is a straight path after a turn, the robot must go past the turn before it makes the turn. This is accomplished with `delay(timeBeforeTurn)`, where `timeBeforeTurn` is defined as `200`. 200 ms is the approximate time that is takes the robot to go past the turn before it is centered on the turn. This is also needed because the robot turns by rotating clockwise and counterclockwise, so it must be near the center of a turn to be able to turn cleanly.

`checkForTurns()` is also where the robot checks to see if it has solved the maze. The end of the maze is represented by a large black square. So whenever a turn is detected, after the delay before every turn, the function uses `allBlack()` to see if all the sensors are reading black. If they are, then the robot has reached the end of the maze and `mazeFinished()` is run. `mazeFinished()` stops the motors, and after a two second delay, makes the robot spin around in place for three seconds as if it was celebrating.

The maze learning code is also home to `setup()` and `loop()`. These two functions are required functions by Arduino as they are where the program execution begins, similar to `main()` in other programming languages. `setup()` is only run once and is generally used for setting up objects and variables that will be used in `loop()`. As its name implies, `loop()` will run indefinitely in a loop. Therefore, the Arduino board will continue running the program until it loses a power source.

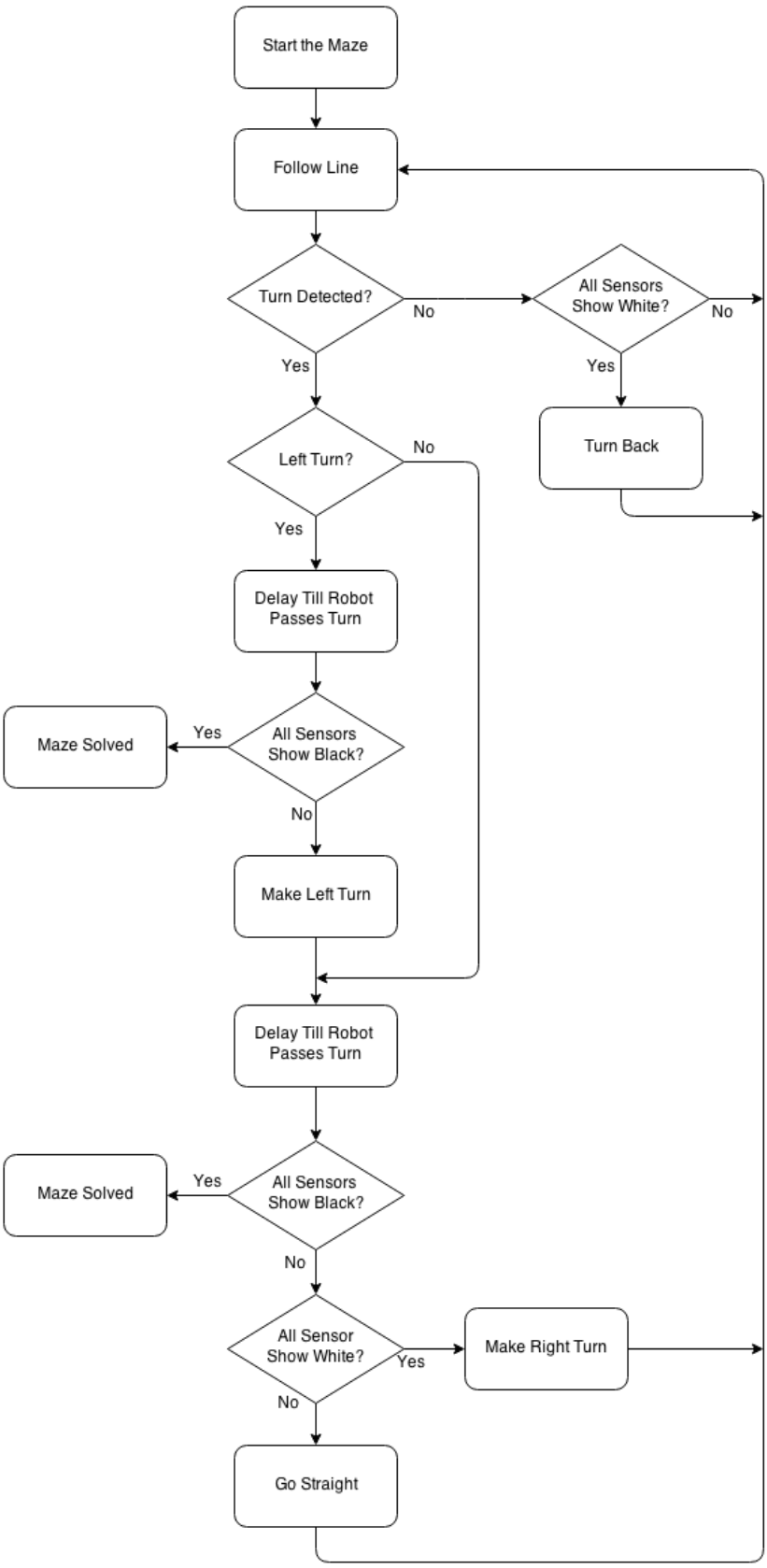

Figure 19 - Maze-Solving Algorithm Flowchart

5.3.2 Maze Optimizing Code

The maze optimizing code takes the turns that were recorded in the `path` array and uses an algorithm to simplify that path into it simplest form.

The `path` array is an array of characters initialized to size `100`, as defined by `pathSize`. This size should be more than sufficient to solve a reasonably sized maze. The characters used to represents turns are as follows: "`L`" for left turn, "`R`" for right turn, "`B`" for back turn, and "`S`" for straight. The turns are recorded during maze solving with the turning functions. At the end of each of these functions, the turn that was executed is added to `path` and `pathLength`, which keeps track of the number of turns taken, is incremented.

The set of turns stored in `path` is optimized by treating it as a math equation and simplifying it based on a set of rules. These rules are as follows:

LBR = B **SBL = R**

LBS = R **RBL = B**

SBS = B **LBL = S**

Figure 20 - Simplification rules

The rules center around eliminating the back turns from the maze. Each of these rules is illustrated in Figure 21 through Figure 26 below.

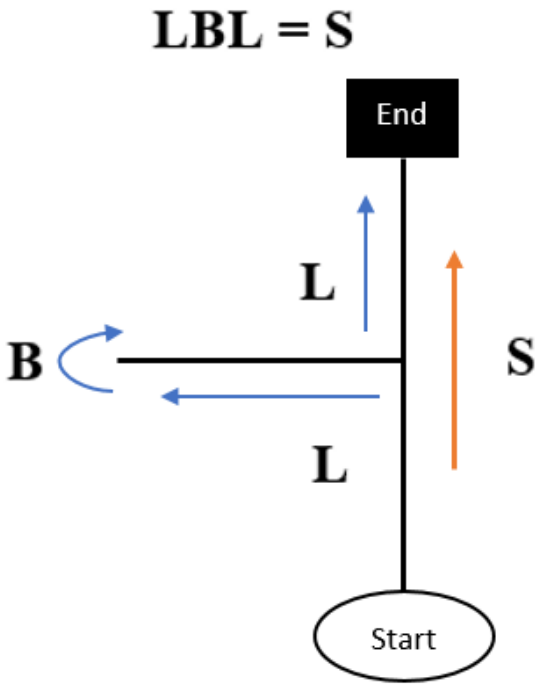

Figure 21 - Left-Back-Left to Straight Simplification Rule

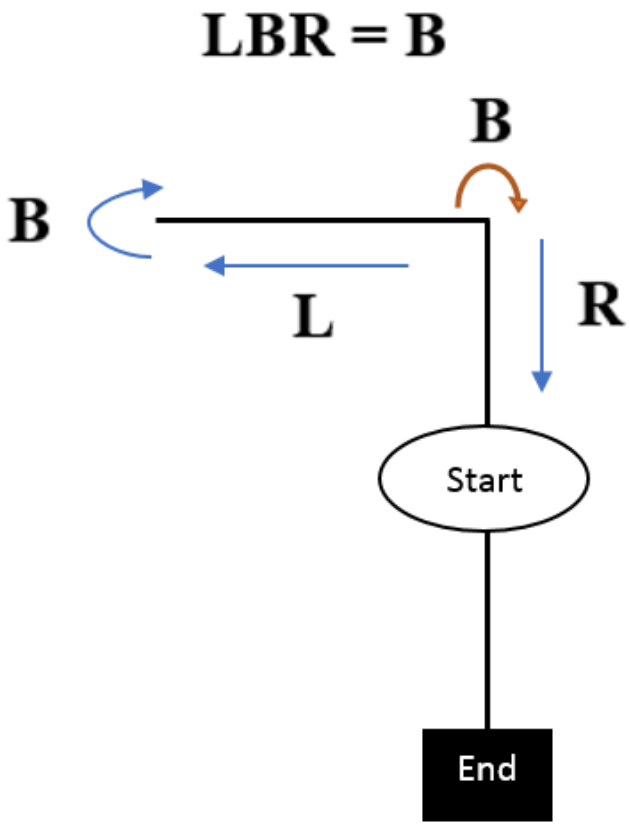

Figure 22 - Left-Back-Right to Back Simplification Rule

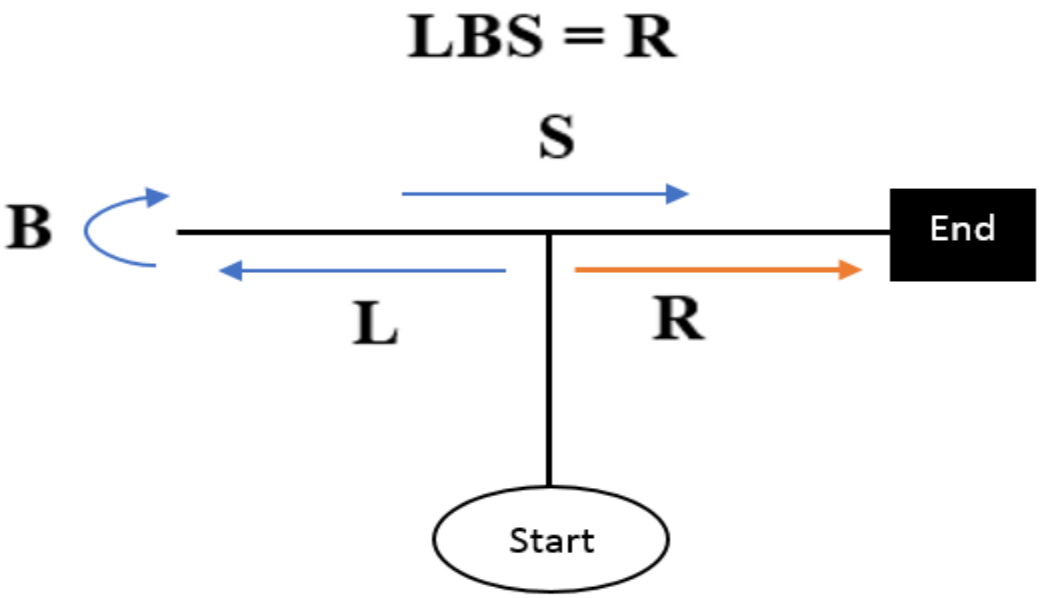

Figure 23 - Left-Back-Straight to Right Simplification Rule

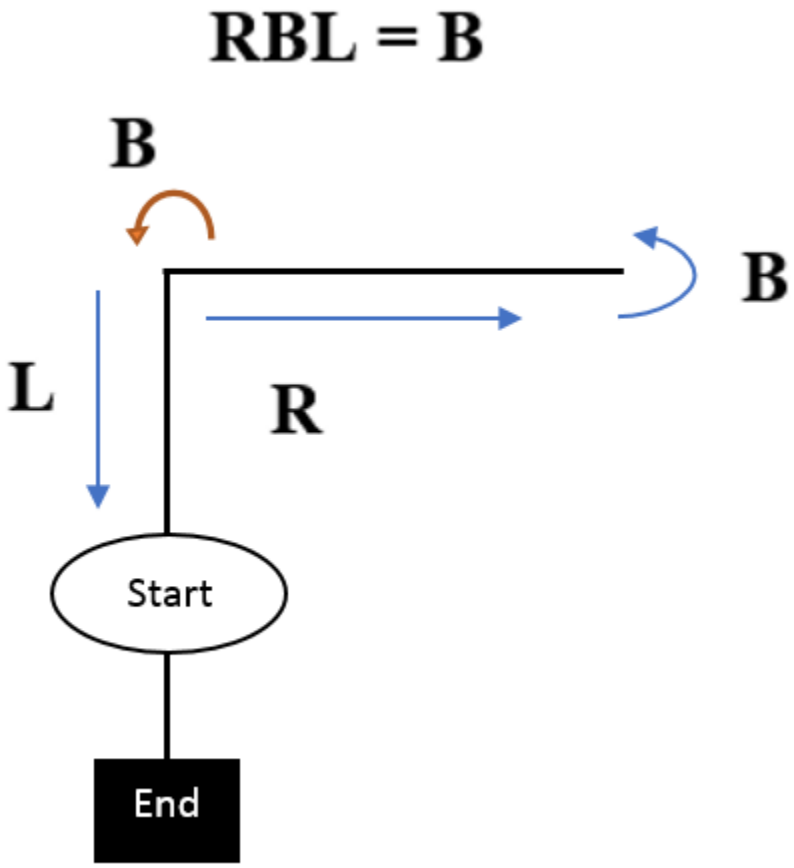

Figure 24 - Right-Back-Left to Back Simplification Rule

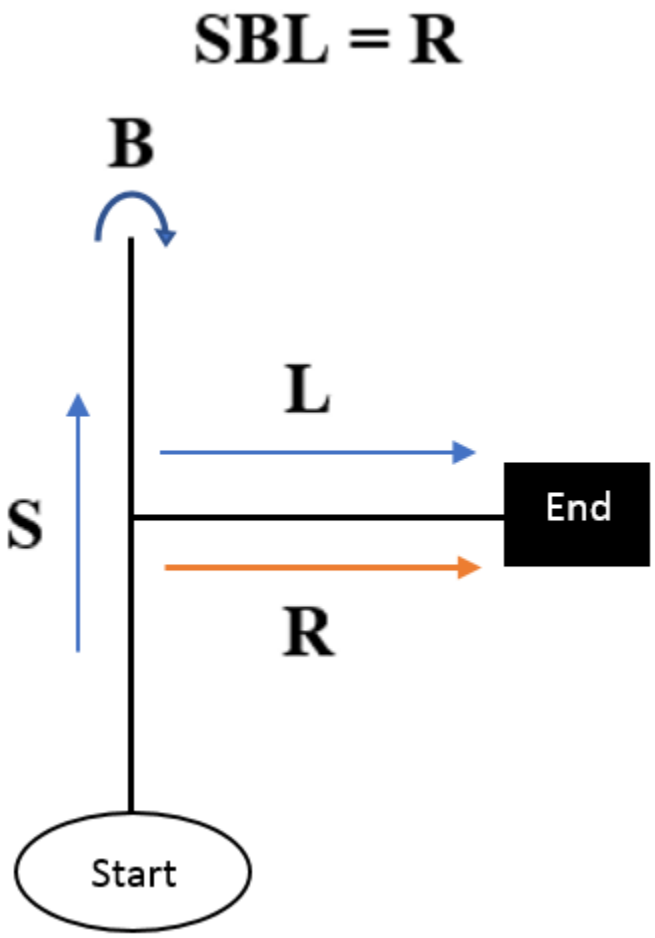

Figure 25 - Straight-Back-Left to Right Simplification Rule

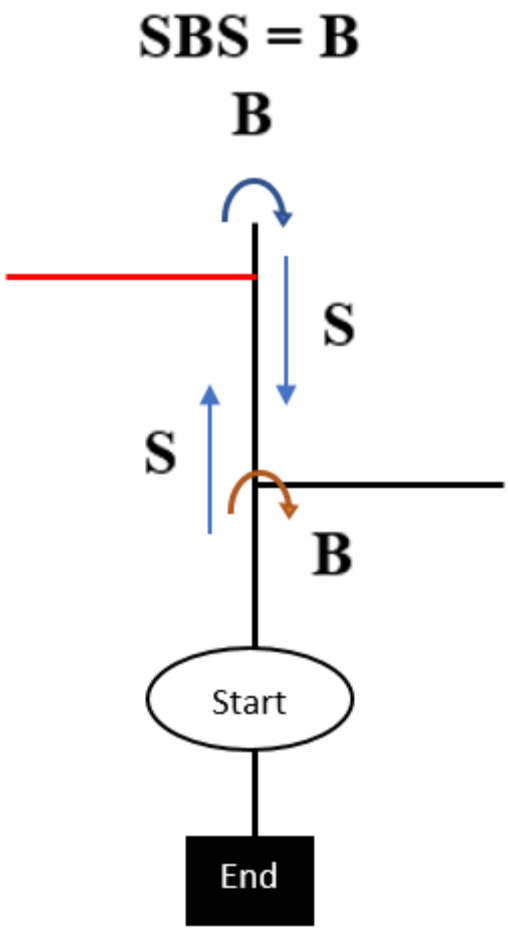

Figure 26 - Straight-Back-Straight to Back Simplification Rule

It should be noted that it is impossible to have an SBS set of turns in a maze because a left turn would be made instead of one of the straights. However, the rule must exist because this set of turns may occur during the optimization of the path. An example of this can be seen in the fourth line of Figure 29.

To show how this algorithm works, the sample maze shown in Figure 27 will be used. The path that results from the robot running this maze is shown in Figure 28.

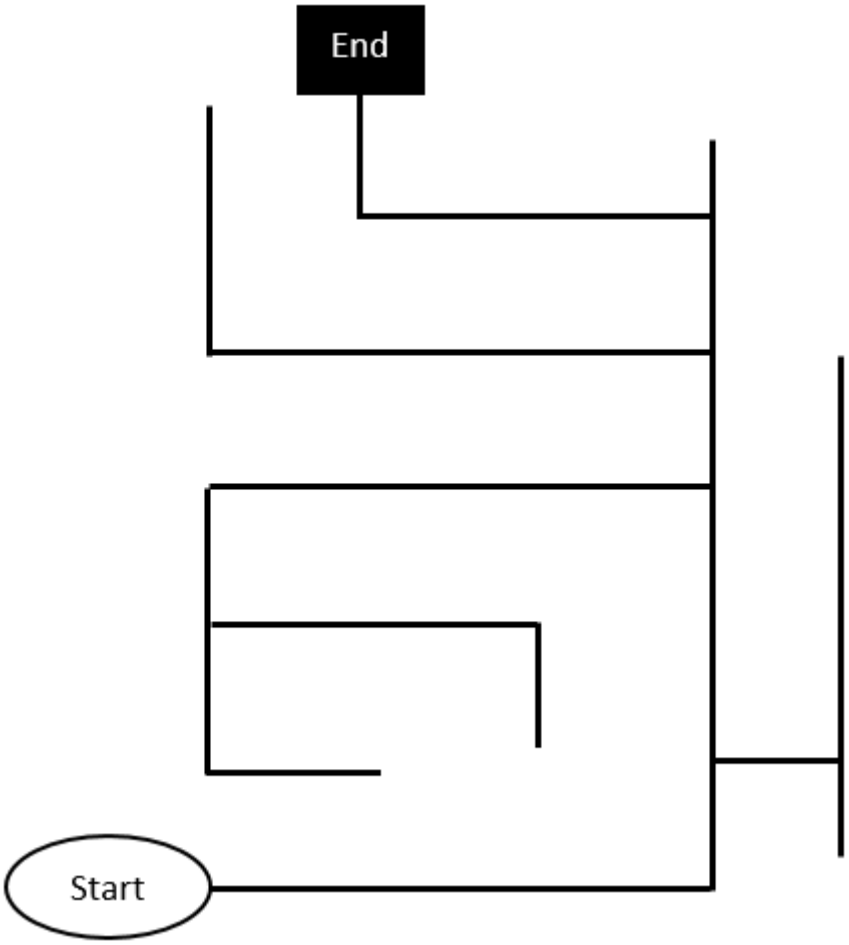

Figure 27 - An Example Maze

L S L L L R B L L L B R S R L L R B L L L R

Figure 28 - The Example Maze's Resulting Path

The way that the algorithm optimizes this path is illustrated in Figure 29. The set of three turns that is currently being simplified is highlighted in yellow, and result of the simplification is highlighted in red.

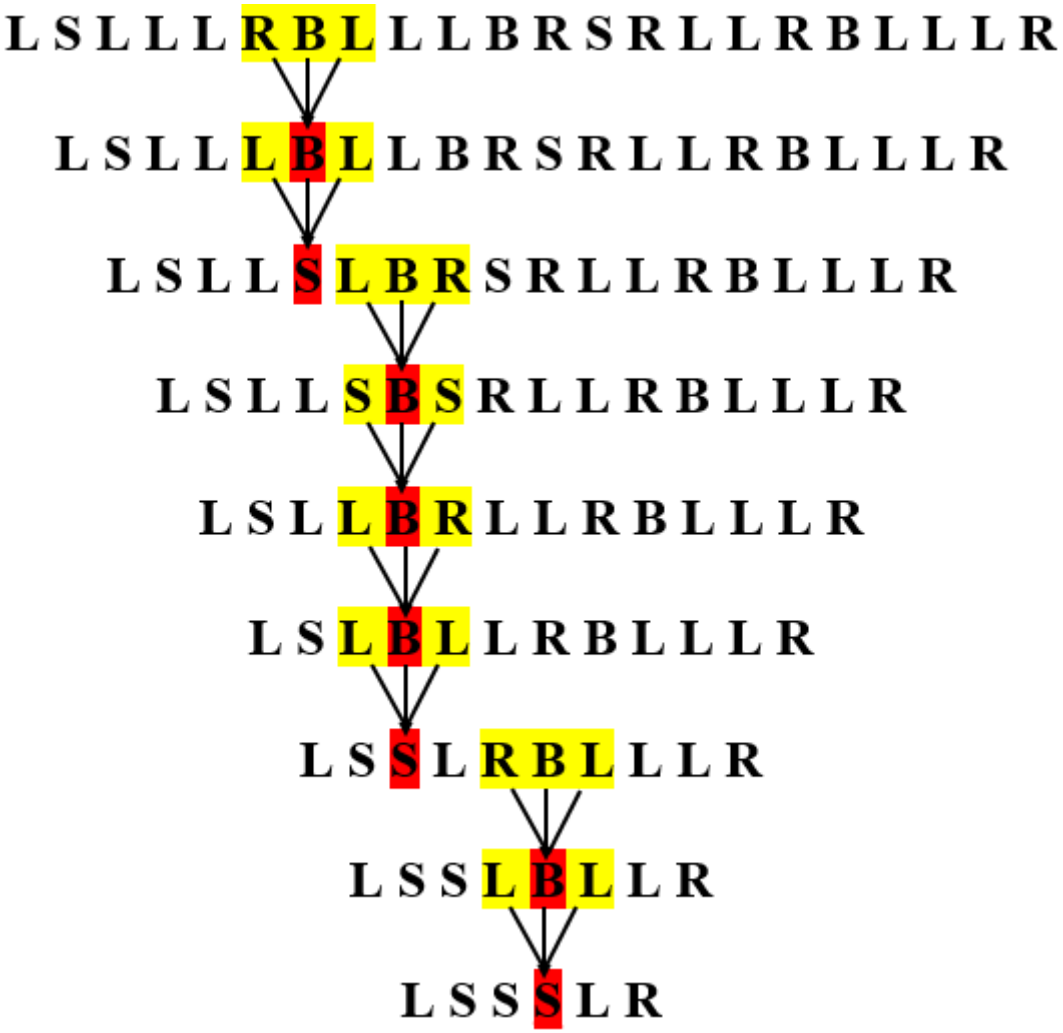

Figure 29 - Maze Optimization Visualized

Once the path has been optimized, the program must have the robot run the optimized path. During the optimization process, the optimized path is stored in `optimizedPath`, which is an array of characters. The `runOptimizedMaze()` function has the robot run the maze based on the `optimizedPath()` array. The robot must be started at the same position and in the same direction that it did when it began solving the maze initially.

`runOptimizedMaze()` works similarly to `solveMaze()`. It runs in a `while` loop that uses `i` as a counter. It uses the same `followLine()` function that `solveMaze()` uses to have the robot follow the line. It also checks for turns, but instead of using the turn priority system from the maze-solving algorithm, whenever it encounters a turn, it checks the next character in `optimizedPath` to see what turn the robot should take.

A unique situation that can occur at this point is when there is a "`B`" in `optimizedPath`. Because the optimization algorithm works by eliminating back turns from the path, one would think it is impossible to have a back turn in the optimized path, but it is possible if the back turn is the first turn in the optimized path. This can occur in situations where the robot starts the maze in a way where it is most efficient to immediately turn back. Also, when the robot has to turn back at a left turn, then the function `turnBackForOptimized()` must be used, which essentially makes the robot take two left turns. This is because the robot will be crossing over a black line while it is making the back turn.

After the robot has taken all the turns in `optimizedPath`, it will then simply follow the line until the end of the maze has been found and then it runs the `mazeFinished()` function.

5.4 The Robot

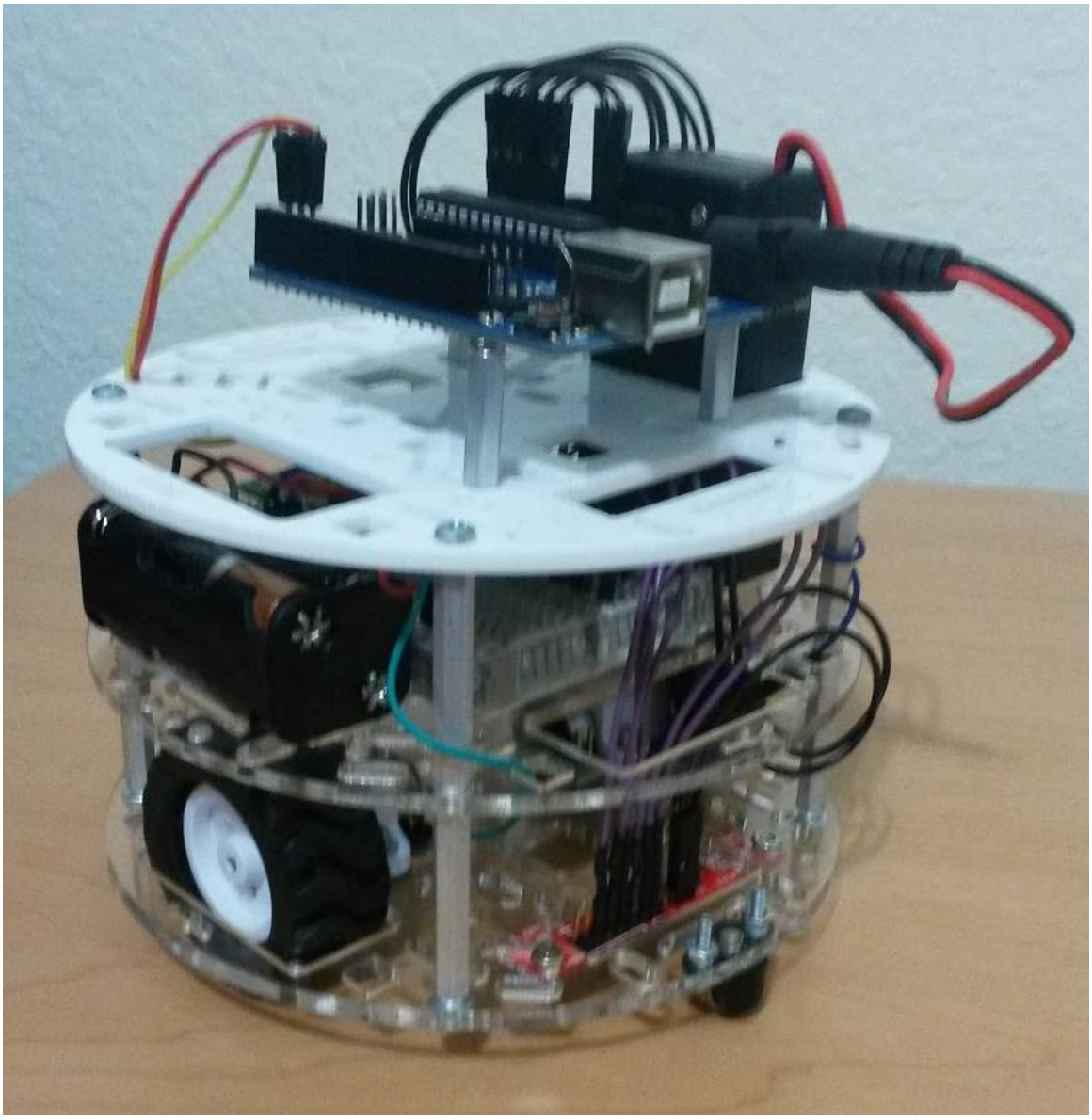

Figure 30 - Arduino Line Maze Solver Version 3.

Once the code had been finalized and tested, preparations began by adding a new level to the Arduino robot which would allow the team greater ease when dealing with cable management as well as improving the aesthetics of the robot. Figure 30 shows the final physical design of the robot, which uses three chassis shown in Figure 7. Three chassis were used to divide the components in three parts: at the bottom level, there is the QTR-8A Reflectance Sensor Array, the DC motors, the encoders and the wheels. In the middle level, there is the breadboard, which manages all the wire connections between the top and bottom levels, and the motor batteries. The

top level contains an Arduino Uno board, the brain of the robot, and a battery pack housing a 9V battery, the board's power supply.

5.5 Limitations

As with most projects, our robot has certain limitations when solving mazes. First of all, the QTR-8A Sensor Array detects reflectivity. For the sensor to work reliably, the maze and robot must be in a brightly lit area. As for the maze itself, the line must be a solid black line that is no more than a .75 inches wide and no less than .25 inches wide, and the surface that the line is on must be near solid white. It is recommended to use black electrical tape for the black line and regular white poster board to form the maze on.

The turns that robot will take must be near 90 degree turns so that the robot can detect the turns and can execute the turns cleanly. Also, if there is a closed loop within the maze, the robot cannot be started within this loop or there is a chance that the robot will not be able to get out of that loop, depending on which direction it is started in.

Last, but not least, make sure that the batteries are replaced periodically or the motors and the reflectance sensor may malfunction.

6. Conclusion

This project has been quite a journey and the team has overcome many difficulties to complete this project. The main goal of this project was to have a robot learn a line maze. This involves two parts: having the robot solve the maze, and having the robot optimize that path that it took to solve that maze. The team believes this goal has been fully accomplished but it was not without many difficulties.

Most of this project's difficulties came from the hardware side of the project. This project required the learning of many different aspects of hardware: from mounting each part to the chassis to wiring the sensors to the Arduino board, all of it had to be done from scratch. The team made some very silly mistakes along the way, such as wiring the QTR-8A Sensor Array to the digital pins on the Arduino board instead of the analog pins. The team also had a problem with the battery that powers the Arduino board. When the 9V battery that powers the Arduino board begins to die, the Arduino board will reduce the power allocated to its 5V output first. This 5V output is used to power the sensor array, so when the battery began to die, the sensor seemed to switch between working and not working randomly, which was very confusing. Luckily, it only took a couple hours for the team to figure this out.

Of course, there were also some difficulties on the software side too. The hardest part of the coding came from trying to make the robot turn cleanly. At first, the robot turned by stopping one of the motors instead of making that motor run in reverse. The team then realized that it is better to have the robot turn by having it go a certain distance past the turn and then rotate clockwise/counterclockwise. This also allowed the robot to check for the black line past the turn, which is necessary for the maze-solving algorithm. Figuring out how to implement the maze optimizing algorithm was also rather difficult.

If this project were to be expanded upon in the future, the team has a couple of ideas. The first would be to make the robot able to solve a maze with walls using ultrasonic/distance sensors. This would most likely require multiple ultrasonic sensors and may require changes to the maze-solving and optimization algorithms. Another idea is to add a speaker to the robot. This could be used to have the robot say something or make a sound effect at certain points in the maze. For

instance when the robot makes a left turn, it could say “Left turn!”. An idea that would be much harder to implement would be to make the robot web compatible by hooking it up to a Wi-Fi adapter. Many different things could be done with this such as commanding the robot to start the maze over the internet. It might also be possible to save the mazes that the robot records to a database.

On a final note, this project was a harsh experience, but the team learned a lot from it. If another group decides to expand on this project, good luck.

7. References

[1] Reference Pending...

[2] Arduino - ArduinoBoardUno 2014

URL: http://arduino.cc/en/Main/arduinoBoardUno

[3] Pololu - 75:1 Micro Metal Gearmotor HP 2014

URL: http://www.pololu.com/product/2361

[4] Users' Guide, Qik 2S9V1 – Pololu Corp. 2014

URL: http://www.pololu.com/docs/pdf/0J25/qik_2s9v1.pdf

[5] Qtr-8A and Qtr-8Rc Reflectance Sensor Array User's Guide – Pololu Corp. 2014

URL: http://www.pololu.com/docs/pdf/0J12/QTR-8x.pdf

[6] Robot Chassis RRC04A Transparent Clear – Pololu Corp. 2014

URL: http://www.pololu.com/product/1505

[7] RoboGames – Robot Maze Solving Competition

URL: http://robogames.net/rules/maze.php

[8] The Robotics Society of Southern California – Line Maze Solving Competition

URL: http://www.rssc.org/content/line-maze-solving

[9] "Pololu - Source Code, Libraries, and Software Utilities."

URL: http://www.pololu.com/resources/software