Control of Mobile Robots

CDA4621.001 S17

Dr. Alfredo Weitzenfeld

Path Planning

Tyler Simoni & Esthevan Romeiro

April 25, 2017

1. Lab Objectives

Lab 4 objectives consisted of the task enumerated below as part of a Path Planning project. Each task can be viewed via a YouTube link provided within each task description under the "Lab Design Overview" section. Each link will be appropriately time stamped as per the described task.

1. Path Planning with a Known Maze
2. Path Planning with an Unknown Maze

2. Lab Design Overview

Descriptions of each design implementation can be viewed below under their appropriate headings. The maze will consist of a 4x4 square grid structure with interchangeable inner walls. As this project builds upon the last project, some of the code was recycled and used for this project as well. For both parts, the maze and position were created using a maze class (Maze.cpp) and a position class (Position.cpp), respectively. Both are written as C++ files with associated header files. The blank maze constructor initializes a 9-by-9 array, creates the outer walls (WALL), blue and red (COLOR) tape, and open (OPEN) positions.

2.1. Path Planning with a Known Maze

The *Path Planning with a Known Maze* task should allow the robot to navigate from the user specified starting grid and heading to the user specified ending grid, using the shortest path possible, *with* knowledge of the inner walls. Due to this stipulation, the robot should, inherently, not pass through the same grid twice. The three mazes shown below could be programmed into the robot for this task. Both a queue and a stack were used to complete this task.

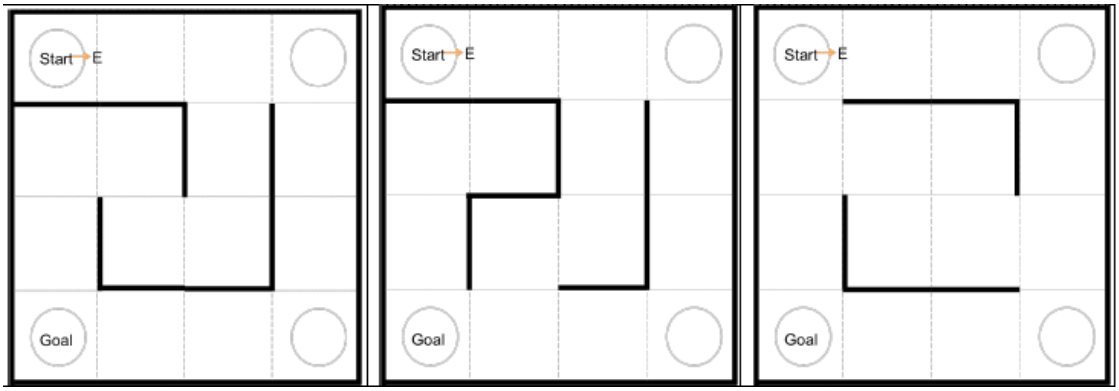

Figure 2-1. The 3 maze configurations with start and goal locations.

The setup for this task first initializes a new 9-by-9 maze and sets the LCD background to white. It then enters the `MazeSelect` function, which allows the user to select which maze configuration to run. Each maze initialization sets the inner walls of the respective maze. The robot then enters the `Init_Position` function. This allows the user to set the starting

position, ending position, and initial compass heading for the task. Then, the initial position is set, as well as the start and exit in the maze class. The button layout is shown below.

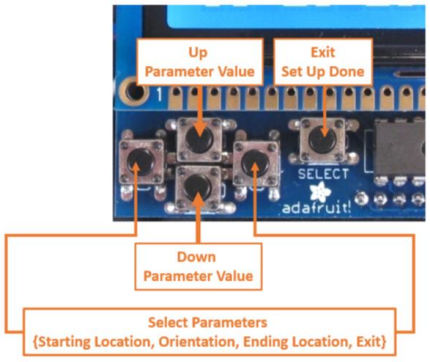

Figure 2-2. Button layout on LCD shield

Once the user has input all the information, the robot then runs the `FindShortestPath` function. Due to each of the mazes being programmed internally, the robot can perform a breadth-first search (bfs) algorithm without actually moving. Once the algorithm is complete, the robot has a stack of sequential positions (FinalPath) in order to reach the goal from start using the shortest path. Once this algorithm completes, the robot can finally move to the execution loop in the Arduino code.

In the main execution loop, the robot performs the `PositionScan`, `PositionDisplay`, `GridDisplay`, and `Movement` functions. At the end of the loop, the robot also checks to see if it has reached the goal position. PositionScan and PositionDisplay set up the LCD display of the current display and headings. The GridDisplay displays the currently traversed grids in the form of X's and O's, as seen below in Figure 2-3 and Figure 2-4.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	X	X	X	0	0	0	X	0	0	X	X	0	X	X	0	0
2	G	1		W	U		N	X		S	0		E	0		

Figure 2-4. Proposed LCD display

Figure 2-3. Actual LCD display

The Movement function using the final path stack, initialized by the BFS algorithm, to move from each grid to the next until it reaches the end goal position. A flowchart detailing the robots processes as it completes this task can be found below in Figure 2-5.

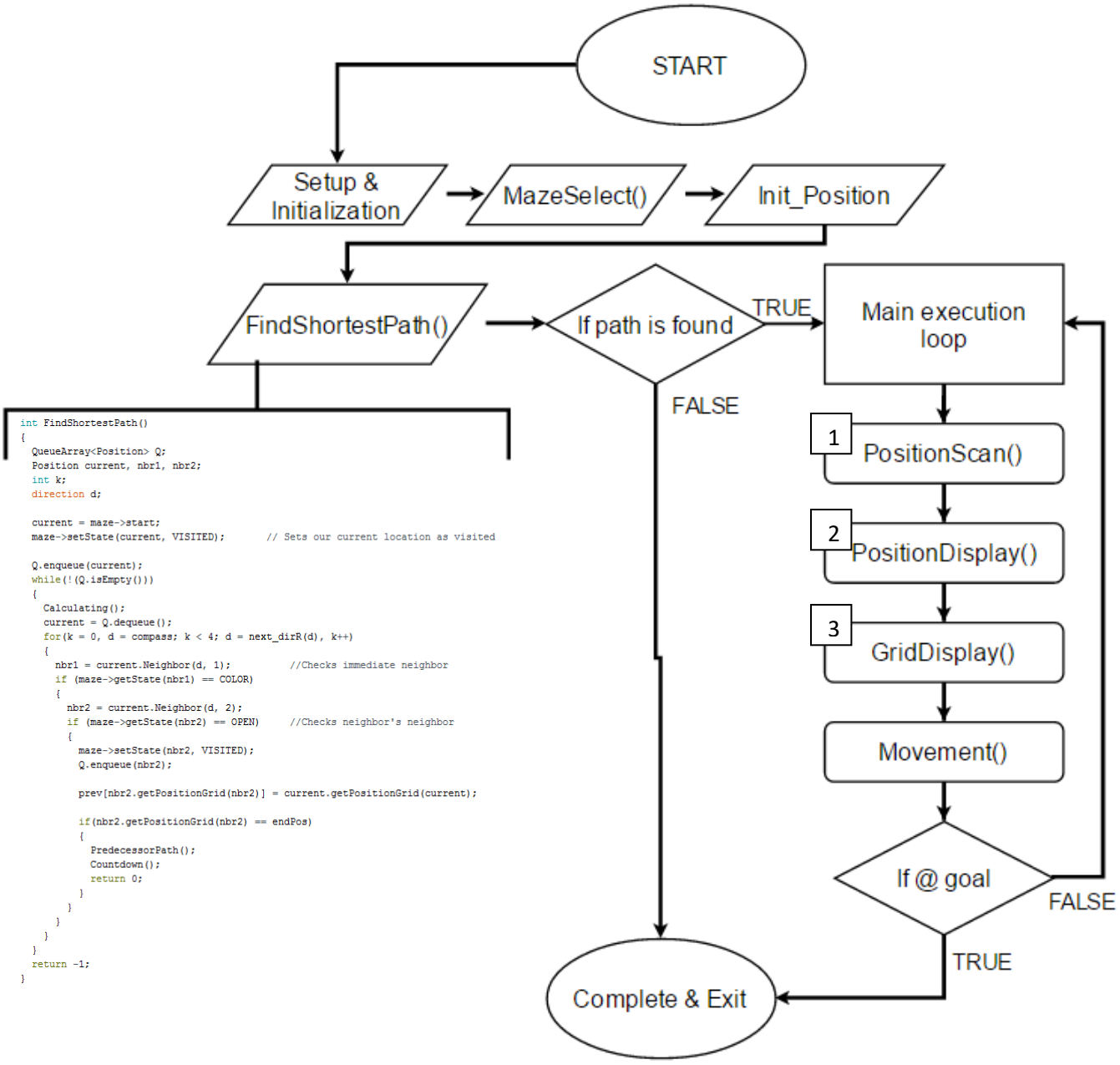

Figure 2-5. Process flowchart for Task 1

1

```
void PositionScan()
{
  /* Pos[] elements:
   *  0 = Grid Location 0-15
   *  1 = SOUTH
   *  2 = WEST
   *  3 = NORTH
   *  4 = EAST
   */

  //WallScan()
  //Set each equal to f, l, or r individually

  SensorRead(); //Remove this

  switch(compass)
  {
    case EAST:
      pos[1] = WallScan(r); //SOUTH Wall/Open
      pos[2] = -1;          //WEST always unknown
      pos[3] = WallScan(l); //NORTH Wall/Open
      pos[4] = WallScan(f); //EAST Wall/Open
      break;
    case WEST:
      pos[1] = WallScan(l); //SOUTH Wall/Open
      pos[2] = WallScan(f); //WEST Wall/Open
      pos[3] = WallScan(r); //NORTH Wall/Open
      pos[4] = -1;          //EAST always unknown
      break;
    case SOUTH:
      pos[1] = WallScan(f); //SOUTH Wall/Open
      pos[2] = WallScan(r); //WEST Wall/Open
      pos[3] = -1;          //NORTH always unknown
      pos[4] = WallScan(l); //EAST Wall/Open
      break;
    case NORTH:
      pos[1] = -1;          //SOUTH always unknown
      pos[2] = WallScan(l); //WEST Wall/Open
      pos[3] = WallScan(f); //NORTH Wall/Open
      pos[4] = WallScan(r); //EAST Wall/Open
      break;
  }
  delay(500);
}
```

2

```
void PositionDisplay()
{
  //Format: G1 NX SX E0 WU
  lcd.setCursor(0,1); //Column 0, Row 1
  lcd.print("G");
  lcd.setCursor(1,1); //Column 1, Row 1
  lcd.print("  ");
  lcd.setCursor(1,1);
  lcd.print(pos[0]);
  lcd.setCursor(3,1); //Column 3, Row 1
  lcd.print("N");
  lcd.setCursor(6,1); //Column 6, Row 1
  lcd.print("S");
  lcd.setCursor(9,1); //Column 9, Row 1
  lcd.print("E");
  lcd.setCursor(12,1); //Column 12, Row 1
  lcd.print("W");

  //North Logic
  lcd.setCursor(4,1);
  switch(pos[3])
  {
    case -1:
      lcd.print("U");
      break;
    case 0:
      lcd.print("O");
      break;
    case 1:
      lcd.print("X");
  }

  //South Logic
  lcd.setCursor(7,1);
  switch(pos[1])
  {
    case -1:
      lcd.print("U");
      break;
    case 0:
      lcd.print("O");
      break;
    case 1:
      lcd.print("X");
  }

    //East Logic
  lcd.setCursor(10,1);
  switch(pos[4])
  {
    case -1:
      lcd.print("U");
      break;
    case 0:
      lcd.print("O");
      break;
    case 1:
      lcd.print("X");
  }

  //West Logic
  lcd.setCursor(13,1);
  switch(pos[2])
  {
    case -1:
      lcd.print("U");
      break;
    case 0:
      lcd.print("O");
      break;
    case 1:
      lcd.print("X");
  }
}
```

3

```
void GridDisplay()
{
  //Replace O's in grid array with X's as new grids are visited
  grid[pos[0]-1] = 'X';

  for(int i = 0; i < 16; i++)
  {
    lcd.setCursor(i,0);
    if (grid[i] == 'X')
      lcd.print('X');
    else
      lcd.print('O');
  }
}
```

2.2. Path Planning with an Unknown Maze

The *Path Planning with an Unknown Maze* task should allow the robot to navigate from the user specified starting grid and heading to the user specified ending grid, using the shortest path possible, *without* knowledge of the inner walls of the maze.

The setup for this task first initializes a new 9-by-9 maze, sets the initial position, sets the initial compass heading, and sets the LCD background to white.

The robot then enters the main execution loop. The loop then checks the DFS and BFS Boolean flags. If they are both false (which is the initial case), the robot enters the `ChooseDFSorBFS` function. This allows the user to choose whether to run the `MazeMapper` function or the `BFS_Setup` function. The code is written in such a way that the user must run the `MazeMapper` first.

The `MazeMapper` function works first checks to see if the current position has been visited or not. If it has, then it scans the positions around it to check for walls. If a wall is found, it is assigned to that position in the maze. Once this loop finishes, the robot sets the current position's state to "visited". Next it starts looking in the current compass heading in the maze in memory to determine a direction to move in. It keeps moving until it visited all 16 grid spaces or it finds a dead end. In this case, it turns around checking each position on its way back for an alternate open position. Once it has been in every grid, the maze is now mapped, it enables the Shortest Path option and it returns to the `ChooseDFSorBFS` function.

The user can now choose the Shortest Path option. Once chosen, the robot then performs the `BFS_Setup`. The robot first resets the maze, positions, and compass headings to be used in the next part. It then enters the `Init_Position` function which allows the user to set the starting position, ending position, and initial compass heading for the task. From this, the initial position is set, as well as the start and exit in the maze class. The button layout is the same as in the previous task. It then performs the `FindShortestPath` function. Because the maze is now mapped out in memory, the robot can perform a breadth-first search algorithm as in Task 1, in order to find the shortest path from start to goal. A check is made to see if the algorithm was successful in both finding a path and populating a stack of position objects. This stack will be used by the robot as in the previous task to traverse now known maze.

If the check was unsuccessful, an error message is displayed and the robot stops there. If successful, the robot will navigate from start to goal and display a success message upon reaching it. A flowchart detailing the robots processes as it completes this task can be found below in Figure 2-6.

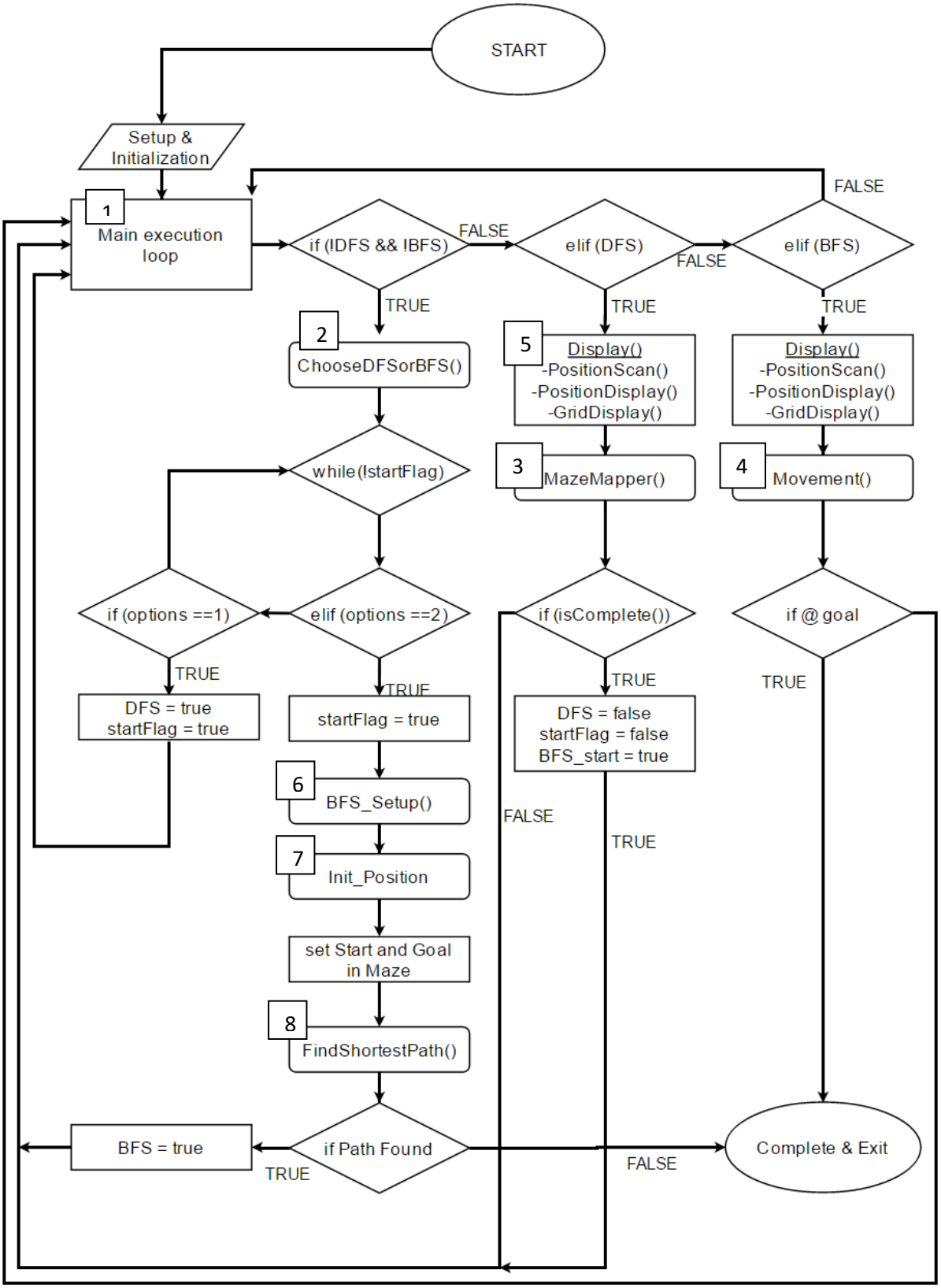

Figure 2-6. Process flowchart for Task 2

1

```
void loop()
{
  if (!DFS && !BFS)        //If a choice needs to be made
  {
    ChooseDFSorBFS();
  }
  else if (DFS)            //If Mapping Maze
  {
    Display();
    MazeMapper();          // Maps the maze

    if (isComplete())
    {
      PositionScan();      // Gets last positions walls
      lcd.clear();
      lcd.print("Maze mapped!");

      DFS = false;         // Maze mapping is done
      BFS_start = true;    //BFS enabled
      startFlag = false;   //Choose loop enabled
      delay(1000);
    }
  }
  else if (BFS && startFlag) //If Finding shortest path
  {
    Display();
    Movement();

    if(currPos.getPositionGrid(currPos) == endPos)
    {
      CompleteMessage(0);
    }
  }
}
```

2

```
void ChooseDFSorBFS()
{
  lcd.clear();
  lcd.print("L: Map maze");
  lcd.setCursor(0,1);
  lcd.print("R: Shortest Path");

  while(!startFlag)
  {
    buttons = lcd.readButtons();
    if(buttons)
    {
      lcd.setCursor(0,0);

      if (buttons & BUTTON_LEFT)
      {
          lcd.clear();
          lcd.print("1. Map the Maze");
          options = 1;
      }
      else if (buttons & BUTTON_RIGHT)
      {
        if(BFS_start){
          lcd.clear();
          lcd.print("2. Shortest Path");
          options = 2;
        }
        else{
          options = 0;
          lcd.clear();
          lcd.print("ERR: Must map");
          lcd.setCursor(0,1);
          lcd.print("the maze first.");
        }
      }
      else if (buttons & BUTTON_SELECT)
      {
        switch (options)
        {
          case 1:
            DFS = true;
            startFlag = true;
            break;
          case 2:
            startFlag = true;
            BFS_Setup();
            break;
          default:
            lcd.clear();
            lcd.print("Please select");
            lcd.setCursor(0,1);
            lcd.print("an option");
            break;
        }
        options = 0;
      }
      else{}
    }
  }
}
```

3

```
void MazeMapper()
{
  int i, j, n = 0, t = 0;
  direction d, e, f;
  Position nbr1, nbr2;

  if (maze->getState(currPos) != VISITED)
  {
    for (f = SOUTH, i = 1; i <= 4; f = next_dirR(f), i++)
    {
      nbr1 = currPos.Neighbor(f, 1);
      if (pos[i] == 1)
        maze->setState(nbr1, WALL);
    }

    maze->setState(currPos, VISITED);
  }else{}

  for (d = compass, i = 0; i < 4; d = next_dirR(d), i++)
  {
    nbr1 = currPos.Neighbor(d, 1);          //Checks immediate neighbor
    if (maze->getState(nbr1) == WALL)
      n++;
    else{
      nbr2 = currPos.Neighbor(d, 2);        //Checks neighbor's neighbor
      if (maze->getState(nbr2) == OPEN){
        Turn(n);                            // Turns 90 degrees, n times;
        currPos = nbr2;                     // Updates position
        MoveForward();                      // Moves forward
        return;
      }
      else
        n++;
    }
  }
  if(i == 4)  //If above loop did not find a space
  {
    e = opposite_dir(Heading.peek());
    while(e != d)
    {
      d = next_dirR(d);
      t++;
    }
    Heading.pop();
    Path.pop();

    Turn(t);
    MoveForward();
    Path.pop();
    Heading.pop();

    currPos = Path.peek();
    return;
  }
}
```

4

```
void MoveForward()
{
  RefreshCount();

  int lSpeed = STOP + 70;
  int rSpeed = STOP - 63;  //Adjust this speed

  ColorFlash(compass);

  while(lEncCount < 140 && rEncCount < 140)
  {
    LServo.write(lSpeed);
    RServo.write(rSpeed);
    EncoderRead();
  }
  pos[0] = GetGrid(pos[0]);

  if(DFS){
    Heading.push(compass);
    Path.push(currPos);
  }

  Stop();
}
```

5

```
void Display()
{
  PositionScan();      // Sets values in position array
  PositionDisplay();   // Displays heading on LCD
  GridDisplay();       // Displays array of grid positions
}
```

6

```
void BFS_Setup()
{
  maze->ResetToOpen();
  Init_Position();    //Sets initial start, end and compass

  pos[0] = startPos;
  maze->start = maze->start.getGridPosition(startPos);
  maze->exitPos = maze->exitPos.getGridPosition(endPos);

  int rc = FindShortestPath();     //Performs the BFS algo to find the shortest path

  if (rc != 0 || FinalPath.isEmpty())
      CompleteMessage(rc);
  else
  {
     BFS = true;
     currPos = FinalPath.peek(); //sets current position from finalpath queue
     FinalPath.pop();
  }
}
```

7

```
void Init_Position()
{

    //Set starting Grid location
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Please select");
  lcd.setCursor(0,1);
  lcd.print("Start Grid");
  delay(1000);

  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Start Grid");
  startPos = SetGrid();

  //Set ending Grid location
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Please select");
  lcd.setCursor(0,1);
  lcd.print("End Grid");
  delay(1000);

  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("End Grid");
  endPos = SetGrid();

  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Start Grid: ");
  lcd.setCursor(13,0);
  lcd.print(startPos);


  lcd.setCursor(0,1);
  lcd.print("End Grid: ");
  lcd.setCursor(13,1);
  lcd.print(endPos);
  delay(1000);

  SetOrientation();
}
```

8

```
int FindShortestPath()
{
  int k;
  QueueArray<Position> Q;
  Position current;
  Position nbr1;
  Position nbr2;
  direction d;

  current = maze->start;
  maze->setState(current, VISITED);       // Sets our current location as visited

  Q.enqueue(current);

  while(!(Q.isEmpty()))
  {
    //Calculating();
    current = Q.dequeue();
    for(k = 0, d = compass; k < 4; d = next_dirR(d), k++)
    {
      nbr1 = current.Neighbor(d, 1);          //Checks immediate neighbor
      if (maze->getState(nbr1) == COLOR)
      {
        nbr2 = current.Neighbor(d, 2);
        if (maze->getState(nbr2) == OPEN)     //Checks neighbor's neighbor
        {
          maze->setState(nbr2, VISITED);
          Q.enqueue(nbr2);

          prev[nbr2.getPositionGrid(nbr2)] = current.getPositionGrid(current);

          if(nbr2.getPositionGrid(nbr2) == endPos)
          {
            PredecessorPath();
            Countdown();
            return 0;
          }
        }
      }
    }
  }
  return -1;
}
```

3. Conclusions

A challenge for this lab was getting the robot to move, both in a straight line and making a near 90 degree turn. Since the encoders were used to move the robot, the exact amount of ticks needed to be calculated to move the robot as accurately and precisely as possible.

It was also made easier due to the similarities between this lab and lab 3. Much of the code was recycled for this lab from lab 3. For Task 1, just some minor user interface changes and a new algorithm needed written to be used in tasks. On the other hand, Task 2 not only needed the new algorithm but a way to map the maze from almost nothing. Most likely due to electronic noise, the IR sensors would sometimes not pick up a wall and, thus, would not mark it in the maze.

Due to this, it ended up creating one of the biggest challenge for this lab. If the maze was not mapped correctly and the BFS algorithm was ran to find a path, it had the potential to find and take a shorter or longer path than was actually available. Of course, this depended on which wall(s) were not mapped correctly, as well as the start and goal positions.

Ultimately, our group was successful in the completion of this lab. The robot performed both tasks as described in the lab handout.