Yanpu Huang 1725298 k1763861

Robotics Individual Report

Term 1:

1.1 Team achievement

The final goal that our team achieved in term 1 was successfully operating the EV3 robot to move follow the black line, stop while detecting red colour and avoid an collision to obstacle on the lane by PID controlling.

1.2 My contribution of term 1

I typed the first edition of java code for the EV3 robot at begin(the constructor of the EV3, some basic moving forward and backward, moving speed controlling, stopping when red colour detected).And I helped our team to test the error and recorded the data by the calculation and the real-world test adjustment.

1.3 Detailed contribution of term 1

1.3.1 Coding for the early edition of EV3

At the begin of the robot laboratory, all of our group members were like totally blank white paper facing to the robotics group project. Luckily we all studied the same java course in the first year. I searched the online information about EV3 programming. I collected some useful information in the EV3 java code library online and imported it into our code.

The requirement of first lab was writing the code that operate the EV3 robot to stop while detecting the red line, if not the robot was supposed to keep moving. I recorded the red line value's range by writing the code "system print out the colour sensor detected value", then I putted it in the "if" condition to make the robot stop when the colour sensor detected value is inside the red colour value's range.

But what we used first was the red colour fetch sample only which was not that accurate for colour detection, the robot can only recognised a small range of the red colour. And also the other colour couldn't be recognised by robot if we only use a sample. So we changed into the RGB

mode(sample(0)+sample(1)+sample(2)) to detect colour by the advise from TA. The RGB mode is much more artificial and preciser which combines the three basic colour(red blue green). It made our robot have a more powerful colour recognising ability. Which was helpful for the PID controlling later as well.

In the end we changed the structure of red detecting method. We wrote a new method called isRed(), which precisely used RGB mode to compare the red value with the sum value, then return the boolean. In the run() method, we used the while condition(while isRed()) to make the robot stop every time detected the red value.

1.3.2 PID controlling

The PID controlling part confused me a lot at the start. We tried to figure out the principle of the PID calculating formula but actually it took a lot of time. So we just wrote the same formula as the power point given from course, and putted some random data to see how the robot work by PID. We spent half-term time on modifying the PID data. As more and more times data-modification, we learned deeper about the PID controlling. Here is the final data of the PID set.

Offset = (9+47)/2

Kp = 1.6

Ki = 0.00

Kd = 3.0

Error = 0.0001

Integral = 0

Derivative = 0

lastError = 0

Correction = 0

Speed = 30

1.3.3 Data testing and error analysis

There were plenty of data we need to modify During the coding progress. As the code became more complicated, the data testing and error analysis became

harder. I needed to compare the balance of the values to make a better edition of data than the last time. Also the data recording was very important as well, because if we modify several times, it is not that lucky every time we can find a way out, so the old partly successful data edition was very important for us, it was an insurance for our EV3 robot.

For the example of analysis the data, sometimes the robot was not able to turn enough degrees to back to the black line. So I needed to modify the wheel rotation speed of turn left, to make it turn more degrees. Sometimes the robot turned too much degrees and resulting in detaching black line too, cause the robot was inside of black line, so I needed to reduce the left wheel rotation speed, or increase the right wheel rotation speed.

1.3.4 obstacle detecting

The obstacle detecting was a hard part of this term. I recorded and putted the best data of the obstacle distance and calculated the turning degrees of the robot after detecting a obstacle. After lots of times test, the robot still couldn't work successfully every time.

At first we wrote the code to make the ultra sonic sensor keep turning right and left, in order to prepare for the different degree facing the obstacle. But in the end we gave up this method because the factors are too much, it is really hard for the data modification.

Finally we wrote code to make the sensor stable when it didn't detect obstacle. But when it detected the obstacle, the sensor will turn 90 degree to keep detecting the obstacle, meanwhile the robot will turn and avoid collision with obstacle.

1.4 Conclusion and discussion of term 1 work

The final version robot that we designed was good, the only mistake it made is when we were doing the practical test, the robot couldn't recognise a small black line part, because that part was a bit damaged and the lab light was reflected on the black line. And also the obstacle detection has a little bit problem, when the TA putted the obstacle in a turning point, the robot couldn't go back to black line every time successfully. Luckily in the practical test we paused the test and modified the turning data, the robot worked great and gain the full mark of test.

If more time given, we could achieve following things

①: We could do better to make the robot adapt the environment. Whatever the light of lab change, the robot can recognise black , white , red more artificially by writing a better algorithm to recognise the colour.

②: We could reduce the time cost and distance cost while the robot is turning, which will be more artificial.

③: We could make robot move faster and more stable, I think it is possible to finish 6-7 circles by adding the rotation speed of wheel and updating the data of turning.

Term 2:

2.1 Team achievement

The final goal that our team achieved in term 2 was successfully operating the EV3 robot to confirm the position itself and plan the route by the self-localisation and A* planning.

2.2 My contribution of term 2

I am the mainly responsible for the data testing and the error analysis. I measured the map for the self-localisation, the moving distance of the robot each time, the rotate degree of the robot each time. I recorded the data every test, compared to the real-world test situation and chose the best performed data to put it in the code. Also I contributed part of the algorithm design of self-localisation and A* planning with teammates together.

2.3 Detailed contribution of term 2

2.3.1 Data testing and error analysis

I putted the first initial data in the code by a simple calculation basic on the self-localisation and A* algorithm. Then I did several testing for every lab time. Every time I needed to update a new data until we get the perfect one.

How I update was basically by observing the situation while the robot was moving. Checking every rotation and the distance of moving forward and backward. Then I modified the previous data to a new one. I recorded the every change of the modification, then compared each data and put the most suitable one.

For example, when I did the rotation test, because of the friction, I found that the robot rotated 88 degrees averagely, even if I wrote the code to let it rotate 90 degrees . So if I wanna rotate 360 degrees I need to add 8 more degrees at the end. Finally we achieve 354 degrees for 4 times 90 degrees rotations.

*Programmed to turn 90 degree.(testing 4 times)

•Actual turned in the first edition of the robot:

87 degrees 92 degrees 87 degrees 86 degrees

•Actual turned in the final edition of robot:

90 degrees 90 degrees 87 degrees 87 degrees

And when I did the moving forward and backward distance test, I found that the robot move 98 cm averagely even if I wrote the code to let it move 100 cm. So if I wanna move 100 cm I need to add 2 more cm in the code.

*Programmed to move 100 cm forward.(testing 4 times)

•Actual moved in the first edition of the robot:

98cm 98 cm 99cm 97cm

•Actual moved in the final edition of the robot:

100cm 100cm 100cm 100cm

PS: The first time we installed the ultra sensor I found that the friction increased because of the sensor was too heavy, so we gave up on using ultra sensor. After that, the data modification become much easier. The error also become smaller.

2.3.2 self-localisation algorithm

For the self-localisation part, we divided the map into 1/37. In order to have the correct data for self-localisation, I measured the each step that robot move which was 1.7 cm. It was the one of 37 grids. I needed to write the pseudo code with my teammate and put the data into the code. The code was designed basic on Recursive Bayesian, I used the formula in the tutorial given and combined it with the real-world measured value to calculate the probability.

```
for all grids                    Recursive Bayesian Updating
    if measure= grid [i,j]
        P([i,j], t=k) =P([i,j], t=k-1)*P(Sensor_work)
    if measure ≠ grid [i,j]
        P([i,j], t=k) =P([i,j], t=k-1)*P(Sensor_wrong)
end

P(i,j)= P(i,j)/ [Sum P(i,j)]

MOVE robot (u,v) and update P
P[i, j]=P(Move_work)*P[i-u, j-v]+P(Move_fail)*P[ i, j]
```

	b1	b2	b3	b4
a1	0.01	0.1	0.01	0.01
a2	0.2	0.3	0.01	0.01
a3	0.01	0.05	0.01	0.01
a4	0.01	0.01	0.01	0.01

Pseudo-code:

*Set up a probability of sensor work and motor work.

*Create a colour array of size 37

*Fetch the sample of colour to recognise which blue is set to be 0 and white is set to be 1

-for all grid compare colour:

•Compare read colour with colour array, if they aren't the same, times P(sensorWrong) to all the grid

•If colour read is equal to colour in colour array

•Then array of colour is equal to P(sensorWork) times Array[colour]

•Else P[colour] is equal to P(sensorWrong) times P[colour]

•Sum up all the probability of grids, and divided by each grid to make the sum of result is equal to 1

•P[colour] is equal to {P[colour] is divided by sum of Array[colour]}

•Move forward by 1.7 cm (1 grid), so now probability of current grids is equal to current probability times moveWork

•Previous probability times moveWrong

Note: moveWork is that the robot successfully move forward for 1 grid.

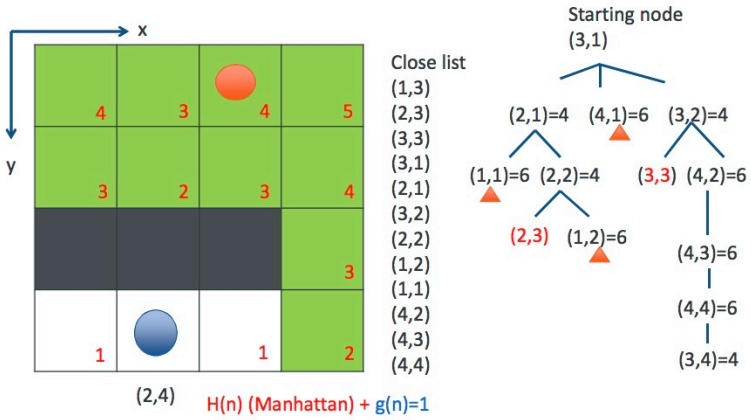
•P[colour] is equal to {P[motorWork] times P[colour]} plus {P[motorWrong] times P(previousColour)}

•If current probability is bigger or equal to 55% which is the only value

•Then return location.

2.3.3 A* planning

For the A* planning we divided the whole map by a number of 5 cm times 5cm grid and set obstacle from a real-world map. For example, for the garage part, the real-world garage was placed in a different position and the size was different as well. So I draw a sketch for it to module our path of robot. I wrote the A* planning pseudo code with my teammates especially the path finding part.



Pseudo code:

•We set initial (start) node and final node in the map, and the area which the robot can search.

•We set the nodes which contains obstacle as “closedSet” and the node which robot can assess in the next step as “openList”. “OpenList” is a priority queue, calculate Heuristic cost (from the node in the openList to final node).

-Actual map of A*

•For the final node, we divide a map into two part: one of this from the location which we get it from self-localisation, to the garage. Then, start from reversing out of garage, to the wall in the end.

-Set up neighbours

•Middle row is row

•If col minus1 is bigger or equal to 0

•Then add neighbour // detect left side of current node, if the current node is not in the most left part of search area.

•If col plus 1 is smaller than the search area

•Then add neighbour // detect right side of current node, if the current node is not in the most right part of searchArea.

•Lower row is equal to row plus 1

•If lower row is smaller than the search area

•Then add neighbour // detect bottom side of current node, if the current node is not in the bottom of searchArea.

•Upper row is equal to row minus 1

•If upper row is bigger or equal to 0

•Then add neighbour // detect top side of current node, if the current node is not in the top of searchArea.

-Path finding

•Check the node which is able to move forward and add them into open list.

•Get path from all possible situation

•Create an array to store nodes

•Set up parent

•Add current node to array. //keep adding nodes below the parent while •There is a parent

•Add nodes in the array until it find a root.

•Return this path. // find a best path from all possible situation

-Simple adjust of some part could avoid collision:

•The original path that is generated by A* can only go on a 90 degree route, which may spend more time on travelling.

•Compare them and help the robot find a closer route to move without collision of the obstacle.

2.4 Conclusion and discussion of term 2 work

In the end I knew how to combine the robot with the mathematical algorithm like A* planning to let the robot have a artificial brain to navigate itself. The software design, hardware design, data collection and error analysis are all important to form a robot. And the pure data calculating is definitely not enough, what makes a successful robot is by hundred times of testing and modifying.

If more time given, we could do better

①: We could make A* algorithm more "smart", with no help from adjusting of our team to select a best route without collision.

②: We could use 1cm*1cm grid for A* if the CPU performance was better, which means less error and better selection of route.

③: We could make robot move faster and more stable without collision.

Appendix

3.1 term 1 code

```
package robotics;

import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3ColorSensor;
import lejos.hardware.sensor.EV3UltrasonicSensor;
import lejos.robotics.SampleProvider;
import lejos.utility.Delay;
import lejos.hardware.motor.UnregulatedMotor;

public class PID extends Thread {

   UnregulatedMotor motorC = new UnregulatedMotor(MotorPort.C);
   UnregulatedMotor motorB = new UnregulatedMotor(MotorPort.B);

   EV3LargeRegulatedMotor motor = new
EV3LargeRegulatedMotor(MotorPort.A);

   EV3UltrasonicSensor ultra = new
EV3UltrasonicSensor(SensorPort.S1);
   SampleProvider distanceMode = ultra.getDistanceMode();

   EV3ColorSensor colorSensor = new
EV3ColorSensor(SensorPort.S4);

   float[] ultrasonicSample = new float[distanceMode.sampleSize()];

   int lightValue;

   public PID() {
```

```
    }

    public static void main(String[] args) throws InterruptedException {
        Thread pd = new Thread(new PID());
        pd.start();
    }

    public void run() {

        int offset = (9 + 47) / 2;

        double kp = 1.6;
        double ki = 0.00;
        double kd = 3.0;

        double error = 0.0001;
        double integral = 0;
        double derivative = 0;
        double lastError = 0;
        double correction = 0;

        final int speed = 30;

        double powerC;
        double powerB;

        motor.rotateTo(0, true);

        while (true) {

            while(isRed()) {
                motorC.stop();
                motorB.stop();
                Delay.msDelay(50);
            }

            if (distance() < 0.12) {
                motorC.close();
                motorB.close();
```

```
        EV3LargeRegulatedMotor mR = new
EV3LargeRegulatedMotor(MotorPort.B);
        EV3LargeRegulatedMotor mL = new
EV3LargeRegulatedMotor(MotorPort.C);

        mR.rotate(-150, true);
        mL.rotate(180);

        motor.rotateTo(-80);

        mR.close();
        mL.close();

        motorC = new UnregulatedMotor(MotorPort.C);
        motorB = new UnregulatedMotor(MotorPort.B);

        while ((int) getAverage() > offset) {

            if (distance() > 0.12) {

            motorB.setPower((int) (30 + (30 / 1.5)));
            motorB.forward();

            motorC.setPower(30);
            motorC.forward();
            if((int) getAverage() <= offset)break;

            }

            if (distance() <= 0.12) {
            motorB.setPower((int) (30 - (30 / 1.5)));
            motorB.forward();

            motorC.setPower(30);
            motorC.forward();
            if((int) getAverage() <= offset)break;
            }

            if((int) getAverage() <= offset)break;
            getAverage();
```

```
            }

            motorB.stop();
            motorC.stop();

            motorB.close();
            motorC.close();

            Delay.msDelay(10);

            EV3LargeRegulatedMotor mRight = new
EV3LargeRegulatedMotor(MotorPort.B);
            EV3LargeRegulatedMotor mLeft = new
EV3LargeRegulatedMotor(MotorPort.C);

            mRight.rotate(-130, true);
            mLeft.rotate(280);

            motor.rotateTo(0);

            mRight.close();
            mLeft.close();

            Delay.msDelay(100);

            motorC = new UnregulatedMotor(MotorPort.C);
            motorB = new UnregulatedMotor(MotorPort.B);

            do {
                motorB.setPower(30);
                motorB.forward();
                motorC.setPower(30);
                motorC.forward();
            }while ((int) getAverage() > offset);

            error = 0.0001;
            integral = 0;
            derivative = 0;
            lastError = 0;
            correction = 0;
```

```
        } else {

            lightValue = (int) getAverage();

            error = lightValue - offset;
            integral = error + integral;
            derivative = error - lastError;

            correction = kp * error + ki * integral + kd * derivative;

            powerB = speed + correction;
            powerC = speed - correction;

            motorB.setPower(new Double(powerB).intValue());
            motorB.forward();
            motorC.setPower(new Double(powerC).intValue());
            motorC.forward();

            lastError = error;
        }

    }
}

float distance() {
    ultra.getDistanceMode().fetchSample(ultrasonicSample, 0);
    return ultrasonicSample[0];

}

public int getAverage() {
    float[] rgb = new float[3];
    int sum = 0;
    for (int i = 0; i < 3; i++) {
        colorSensor.getRGBMode().fetchSample(rgb, 0);
        sum += (rgb[0] + rgb[1] + rgb[2]) * 100;
    }
    return sum / 3;
}
```

```
    public boolean isRed() {
        float[] rgb = new float[3];
        int sum = 0;
        int red = 0;
        for (int i = 0; i < 3; i++) {
            colorSensor.getRGBMode().fetchSample(rgb, 0);
            sum += (rgb[0] + rgb[1] + rgb[2]) * 100;
            red  += rgb[0]*100;
        }
        return (red*3 > sum);
    }
}
```

3.2 term 2 code

```
import java.util.*;

public class AStar {
    private static int DEFAULT_HV_COST = 10;
    private int hvCost;
    private Node[][] searchArea;
    private PriorityQueue<Node> openList;
    private Set<Node> closedSet;
    private Node initialNode;
    private Node finalNode;

    public AStar(int rows, int cols, Node initialNode, Node finalNode, int
hvCost) {
        this.hvCost = hvCost;
        setInitialNode(initialNode);
        setFinalNode(finalNode);
        this.searchArea = new Node[rows][cols];
        this.openList = new PriorityQueue<Node>(50,new
NodeComparator());
        setNodes();
        this.closedSet = new HashSet<>();
    }

    public AStar(int rows, int cols, Node initialNode, Node finalNode) {
        this(rows, cols, initialNode, finalNode, DEFAULT_HV_COST);
    }
```

```
    private void setNodes() {
        for (int i = 0; i < searchArea.length; i++) {
            for (int j = 0; j < searchArea[0].length; j++) {
                Node node = new Node(i, j);
                node.calculateHeuristic(getFinalNode());
                this.searchArea[i][j] = node;
            }
        }
    }
```

```
    public void setBlocks(int[][] blocksArray) {
        for (int i = 0; i < blocksArray.length; i++) {
            int row = blocksArray[i][0];
            int col = blocksArray[i][1];
            setBlock(row, col);
        }
    }
```

```
    public List<Node> findPath() {
        openList.add(initialNode);
        while (!isEmpty(openList)) {
            Node currentNode = openList.poll();
            closedSet.add(currentNode);
            if (isFinalNode(currentNode)) {
                return getPath(currentNode);
            } else {
                addAdjacentNodes(currentNode);
            }
        }
        return new ArrayList<Node>();
    }
```

```
    private List<Node> getPath(Node currentNode) {
        List<Node> path = new ArrayList<Node>();
        path.add(currentNode);
        Node parent;
        while ((parent = currentNode.getParent()) != null) {
            path.add(0, parent);
            currentNode = parent;
        }
```

```
        return path;
    }

    private void addAdjacentNodes(Node currentNode) {
        addAdjacentUpperRow(currentNode);
        addAdjacentMiddleRow(currentNode);
        addAdjacentLowerRow(currentNode);
    }

    private void addAdjacentLowerRow(Node currentNode) {
        int row = currentNode.getRow();
        int col = currentNode.getCol();
        int lowerRow = row + 1;
        if (lowerRow < getSearchArea().length) {
            checkNode(currentNode, col, lowerRow, getHvCost());
        }
    }

    private void addAdjacentMiddleRow(Node currentNode) {
        int row = currentNode.getRow();
        int col = currentNode.getCol();
        int middleRow = row;
        if (col - 1 >= 0) {
            checkNode(currentNode, col - 1, middleRow, getHvCost());
        }
        if (col + 1 < getSearchArea()[0].length) {
            checkNode(currentNode, col + 1, middleRow, getHvCost());
        }
    }

    private void addAdjacentUpperRow(Node currentNode) {
        int row = currentNode.getRow();
        int col = currentNode.getCol();
        int upperRow = row - 1;
        if (upperRow >= 0) {
            checkNode(currentNode, col, upperRow, getHvCost());
        }
    }

    private void checkNode(Node currentNode, int col, int row, int cost) {
        Node adjacentNode = getSearchArea()[row][col];
```

```
        if (!adjacentNode.isBlock() && !
getClosedSet().contains(adjacentNode)) {
            if (!getOpenList().contains(adjacentNode)) {
                adjacentNode.setNodeData(currentNode, cost);
                getOpenList().add(adjacentNode);
            } else {
                boolean changed =
adjacentNode.checkBetterPath(currentNode, cost);
                if (changed) {
                    getOpenList().remove(adjacentNode);
                    getOpenList().add(adjacentNode);
                }
            }
        }
    }

    private boolean isFinalNode(Node currentNode) {
        return currentNode.equals(finalNode);
    }

    private boolean isEmpty(PriorityQueue<Node> openList) {
        return openList.size() == 0;
    }

    private void setBlock(int row, int col) {
        this.searchArea[row][col].setBlock(true);
    }

    public Node getInitialNode() {
        return initialNode;
    }

    public void setInitialNode(Node initialNode) {
        this.initialNode = initialNode;
    }

    public Node getFinalNode() {
        return finalNode;
    }

    public void setFinalNode(Node finalNode) {
```

```
        this.finalNode = finalNode;
    }

    public Node[][] getSearchArea() {
        return searchArea;
    }

    public void setSearchArea(Node[][] searchArea) {
        this.searchArea = searchArea;
    }

    public PriorityQueue<Node> getOpenList() {
        return openList;
    }

    public void setOpenList(PriorityQueue<Node> openList) {
        this.openList = openList;
    }

    public Set<Node> getClosedSet() {
        return closedSet;
    }

    public void setClosedSet(Set<Node> closedSet) {
        this.closedSet = closedSet;
    }

    public int getHvCost() {
        return hvCost;
    }

    public void setHvCost(int hvCost) {
        this.hvCost = hvCost;
    }
}
```

```
import lejos.hardware.Sound;
```

```
import lejos.hardware.motor.EV3LargeRegulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3ColorSensor;
import lejos.hardware.sensor.EV3GyroSensor;
import lejos.hardware.sensor.EV3TouchSensor;
import lejos.hardware.sensor.SensorMode;
import lejos.robotics.RegulatedMotor;
import lejos.robotics.SampleProvider;
import lejos.robotics.navigation.DifferentialPilot;
import lejos.utility.Delay;
import java.util.*;


public class movement1 {
    RegulatedMotor motorR = new
EV3LargeRegulatedMotor(MotorPort.C);
    RegulatedMotor motorL = new
EV3LargeRegulatedMotor(MotorPort.D);

    EV3ColorSensor colorSensor = new
EV3ColorSensor(SensorPort.S1);
    SensorMode color = colorSensor.getRGBMode();
    float[] sample = new float[color.sampleSize()];

    EV3TouchSensor touchSensor = new
EV3TouchSensor(SensorPort.S2);
    SensorMode touch = touchSensor.getTouchMode();
    float[] touchSample = new float[touch.sampleSize()];

    EV3GyroSensor gyroSensor = new EV3GyroSensor(SensorPort.S4);
    SampleProvider gyro = gyroSensor.getAngleMode();
    float[] gyroSample = new float[gyro.sampleSize()];

    DifferentialPilot pilot = new DifferentialPilot(3f, 7.7f, motorL, motorR,
true);

    static int[] colorArray =
{0,0,1,0,0,0,1,1,0,0,1,0,0,0,1,1,0,0,0,1,1,0,0,1,0,0,0,1,1,0,0,1,0,0,0,1,1};

    static int number_Of_Grids = 37;
```

```
    static float sensor_Work_P = 0.95f;
    static float sensor_Wrong_P = 0.05f;
    static float move_Work_P = 0.95f;
    static float move_Wrong_P = 0.05f;

    int accurateAngel = 0;

    public static void main(String[] args){

        movement1 m = new movement1();
        int pos = 0;
        float[] P = new float[number_Of_Grids];
        for(int x = 0;x < number_Of_Grids;x++){
            P[x] = 1f/(number_Of_Grids);
        }

        for(int x = 0;x < 20;x++){
            P =
m.moveAndUpdate(m.normalization(m.changeToP(P,m.compare())));
            if(m.maxOfArray(P) > 0.55 &&
m.numberOfMax(P,m.maxOfArray(P)) == 1){
                pos = m.indexOfMax(P,m.maxOfArray(P));
                System.out.println("S" + m.maxOfArray(P) + "pos" +
m.indexOfMax(P,m.maxOfArray(P)));
                break;
            }
            System.out.println(m.maxOfArray(P));
        }

        Node initialNode = new Node(4, 7);
        Node finalNode = new Node(0, 0);

        int rows = 5;
        int cols = 8;
        AStar aStar = new AStar(rows, cols, initialNode, finalNode);
        int[][] blocksArray = new int[][]{{0, 3}, {0, 4}, {0, 5},{0,6},{0,7}};
        aStar.setBlocks(blocksArray);
        List<Node> path = aStar.findPath();
        for (Node node : path) {
            System.out.println(node);
        }
```

```
        Delay.msDelay(1000);

        int colChange = initialNode.getCol()-finalNode.getCol();
        int rowChange = initialNode.getRow()-finalNode.getRow();

        /*double angle = Math.atan((colChange)/(rowChange));
        m.rotate(-60);
        float distance = (float) Math.sqrt(colChange * colChange +
rowChange * rowChange)*5;
        m.moveDistance(distance);
        m.rotate(63);
        m.moveDistance(48);
        m.rotate(45);
        m.moveDistance(15); */

        m.rotate(-78);
        m.moveDistance(57);
        m.rotate(90);
        m.moveDistance(36);
        m.rotate(40);
        m.moveDistance(35);

        m.fetchTouch();
        Sound.beep();
        m.redOrGreen();

        //Delay.msDelay(50000);
    }

    private void move2(){
        motorL.setSpeed(200);
        motorR.setSpeed(200);
        motorL.rotate(65,true);
        motorR.rotate(65);
    }

    private void fetchTouch(){
        touch.fetchSample(touchSample,0);
        while(touchSample[0] == 0){
```

```
            move2();
            touch.fetchSample(touchSample,0);
        }
    }

    private void redOrGreen(){
        color = colorSensor.getRedMode();
        color.fetchSample(sample, 0);
        //G
        if(sample[0] < 0.3){
            moveDistance(-38);
            rotate(45);
            moveDistance(74);
            rotate(90);
            moveDistance(50);
            rotate(45);
            moveDistance(20);
            rotate(-62);
            moveDistance(5);
            fetchTouch();
        }
        //R
        else {
            moveDistance(-38);
            rotate(45);
            moveDistance(90);
            rotate(45);
            moveDistance(8);
            rotate(45);
            moveDistance(30);
            rotate(90);
            moveDistance(37);
            accurateAngel = -90;
            rotate(-103);
            moveDistance(40);
            fetchTouch();
        }
    }

    private void rotateM(){
        gyro.fetchSample(gyroSample,0);
```

```
        if(accurateAngel > gyroSample[0]){
            pilot.rotate(abs(abs(accurateAngel) - abs(gyroSample[0])));
        }
        else if (accurateAngel < gyroSample[0])
        {
            pilot.rotate(-abs(abs(accurateAngel) - abs(gyroSample[0])));
        }
        else {
            System.out.println("succ");
        }
    }

    private float abs(float x){
        if(x >= 0) {return x;}
        else {return (-x);}
    }
```

```
    private void moveDistance(float distance){
        int distanceP = (int)((distance/9.4245)*360);
        motorL.setSpeed(500);
        motorR.setSpeed(500);
        motorL.rotate(distanceP,true);
        motorR.rotate(distanceP);
    }

    private void rotate(float angel){
        accurateAngel += angel;
        motorL.setSpeed(200);
        motorR.setSpeed(200);
        pilot.rotate(angel);
        rotateM();
    }

    private int fetchColor(){
        color.fetchSample(sample, 0);
        if(sample[0]+sample[1]+sample[2]<0.2 &&
sample[0]+sample[1]+sample[2]>0.1){
            return 0; //blue
        }
        else {
            return 1; //white
```

```
        }
    }

    private float[] compare(){
        float[] temp = new float[number_Of_Grids];
        int color = fetchColor();
        for(int x = 0;x < number_Of_Grids;x++){
            temp[x] = color;
        }
        for(int x = 0;x < number_Of_Grids;x++){
            if(temp[x] == colorArray[x]){
                temp[x] = 1;
            }
            else{
                temp[x] = 0;
            }
        }
        return temp;
    }

    private float[] changeToP(float[] old,float[] compareP){
        for(int x = 0; x < compareP.length;x++){
            if(compareP[x] == 1.0){
                old[x] *= sensor_Work_P;
            }
            else {
                old[x] *= sensor_Wrong_P;
            }
        }
        return old;
    }

    private float[] normalization(float[] temp){
        float base = sumOfArray(temp);
        for(int x = 0;x < temp.length;x++){
            temp[x] = temp[x]/base;
        }
        return temp;
    }

    private float sumOfArray(float[] temp){
```

```
        float sum = 0f;
        for(float i : temp) {
            sum += i;
        }
        return sum;
    }

    private float maxOfArray(float[] temp){
        float max;
        max = temp[0];
        for (int x = 1;x < temp.length;x++) {
            if(temp[x] > max){
                max = temp[x];
            }
        }
        return max;
    }

    private int numberOfMax(float[] array, float max){
        int counter = 0;
        for(int x = 0;x < array.length;x++){
            if(array[x] == max){
                counter++;
            }
        }
        return counter;
    }

    private int indexOfMax(float[] array, float max){
        int index = 0;
        for(int x = 0;x < array.length;x++){
            if(array[x] == max){
                index = x;
            }
        }
        return index;
    }

    private float[] moveAndUpdate(float[] temp){
        move2();
        float start = temp[0];
```

```
        for(int x = 0; x < temp.length -1;x++){
            temp[x] = move_Work_P * temp[x+1] + move_Wrong_P *
temp[x];
        }
        temp[temp.length-1] =  move_Work_P * start + move_Wrong_P *
temp[temp.length-1];
        return temp;
    }

    private int getAccurateAngel(){
        return accurateAngel;
    }
    private float getAngel(){
        gyro.fetchSample(gyroSample,0);
        return gyroSample[0];
    }

}

public class Node {

    private int g;
    private int f;
    private int h;
    private int row;
    private int col;
    private boolean isBlock;
    private Node parent;

    public Node(int row, int col) {
        super();
        this.row = row;
        this.col = col;
    }

    public void calculateHeuristic(Node finalNode) {
        this.h = Math.abs(finalNode.getRow() - getRow()) +
Math.abs(finalNode.getCol() - getCol());
    }

    public void setNodeData(Node currentNode, int cost) {
```

```
        int gCost = currentNode.getG() + cost;
        setParent(currentNode);
        setG(gCost);
        calculateFinalCost();
    }

    public boolean checkBetterPath(Node currentNode, int cost) {
        int gCost = currentNode.getG() + cost;
        if (gCost < getG()) {
            setNodeData(currentNode, cost);
            return true;
        }
        return false;
    }

    private void calculateFinalCost() {
        int finalCost = getG() + getH();
        setF(finalCost);
    }

    @Override
    public boolean equals(Object arg0) {
        Node other = (Node) arg0;
        return this.getRow() == other.getRow() && this.getCol() ==
other.getCol();
    }

    @Override
    public String toString() {
        return "Node [row=" + row + ", col=" + col + "]";
    }

    public int getH() {
        return h;
    }

    public void setH(int h) {
        this.h = h;
    }

    public int getG() {
```

```
        return g;
    }

    public void setG(int g) {
        this.g = g;
    }

    public int getF() {
        return f;
    }

    public void setF(int f) {
        this.f = f;
    }

    public Node getParent() {
        return parent;
    }

    public void setParent(Node parent) {
        this.parent = parent;
    }

    public boolean isBlock() {
        return isBlock;
    }

    public void setBlock(boolean isBlock) {
        this.isBlock = isBlock;
    }

    public int getRow() {
        return row;
    }

    public void setRow(int row) {
        this.row = row;
    }

    public int getCol() {
        return col;
```

```
    }

    public void setCol(int col) {
        this.col = col;
    }
}


import java.util.Comparator;

public class NodeComparator implements Comparator<Node> {

  @Override
    public int compare(Node node0, Node node1) {
    return Integer.compare(node0.getF(), node1.getF());
    }

}
```