COMP329 Programming Assignment Report

Lizhenghe.Chen_201521681

Catalogue

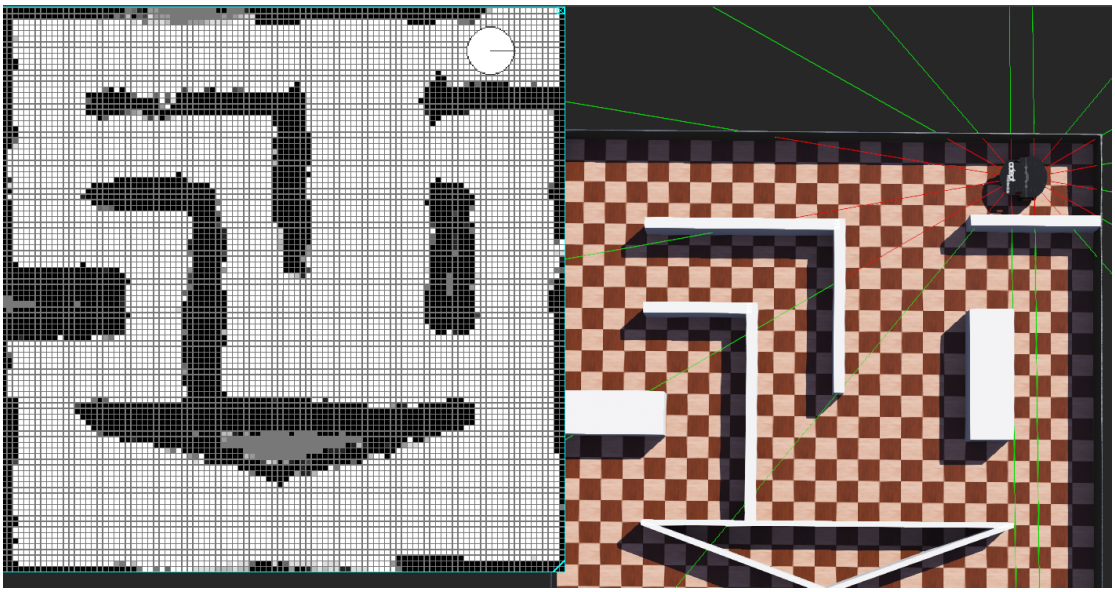
Results of my Assignment 1

The overall thought 2

Implementation details 3

Reflection and summary 6

Results of my Assignment



The degree of completion: All of the obstacles have been fully observed, but with some noise, some acute angles might be a bit smooth or with more noise

Time: with a simple PID control, no more than 3:10 minutes

Stability: Good, guaranteed not hit the wall or in a dead loop in this map, also can fit other maps (**mostly**).

The overall thought

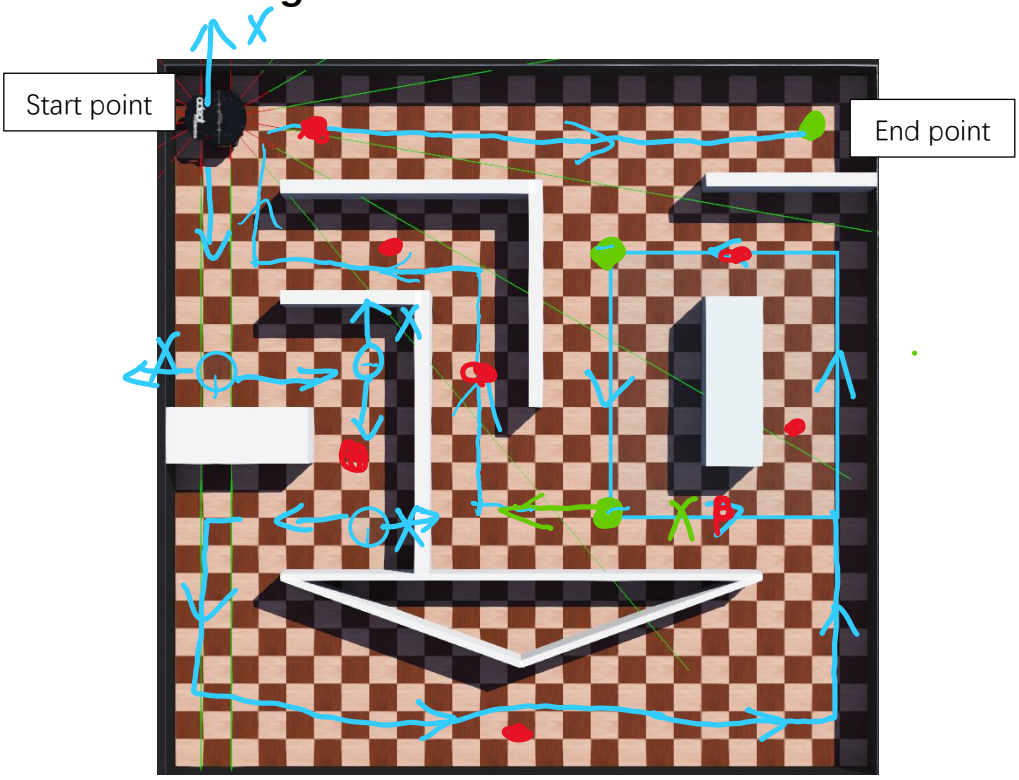


Figure 1

To let the robot find the correct path, my idea is to use the left side sensors and right-side sensors, when the robot is at the beginning (as the picture shows) or the front is almost hit the wall, then if the distance from the left is bigger than the right, turn left 90 degrees:

if (leftd > rightd) {turn left}, and vice versa for turn right shows on the figure1 with **blue**.

When the robot finds a dead loop, find the farthest duplicate point and turn another side, shown as **green** in figure1. When the robot is in a narrow zone, scan around, then go next, it shows as **red**.

Except for the above situations, the robot will only go forward.

How to avoid incorrect reflection?

When the sensor rays hit the wall, if the incident angle is too large (like more than 25 degrees), the sensor may not receive the feedback because the rays were reflected another way. This situation is more likely to happen when the wall is too close to the robot.

The way to solve this problem is to frequently change robots to make sure at least one ray can hit the wall vertically. For example, let robots rotate around or shake left and right. I will mainly let the robot rotate around when making a turning and let the robot shake to some

extent.

Implementation details

I will draw on the overall logic of Lab2 but also will try to use other labs knowledge to implement my navigation and kinematic code. Robots will have an Enum list to represent all possible actions:

```
  public enum MoveState {
    STOP, ROTATE_AROUND_STOP, FORWARD, TURN_RIGHT_90, TURN_LEFT_90,
ROTATE_AROUND, TURN_LITTLE_RIGHT, TURN_LITTLE_LEFT,
    AWAY_FROM_LEFT, AWAY_FROM_RIGHT
  };
```

How to get different distances from sensors:

The distances like forwarding left or right will **use two sensors from the same side**. Using single sensors cannot represent a wall near that side because some walls might be very thin (like two walls near the start area in Fiure1), or it is just an obstacle that will confuse the robot. I implemented a **getDistance()** function to calculate the average distance from two input

```
  public static double getDistance(DistanceSensor psA, DistanceSensor psB) {
    // we can know the elements 5.0 and 1024 from: ps[0].getLookupTable();
    // this method will return and caculate the averange distance from two sensors
    double A = 5.0 - (5.0 / 1024.0 * psA.getValue());
    double B = 5.0 - (5.0 / 1024.0 * psB.getValue());
    return (A + B) / 2;
  }

  public static double forward_distance(DistanceSensor[] ps) {
    return getDistance(ps[3], ps[4]);
  }

  public static double leftward_distance(DistanceSensor[] ps) {···
```

sensors, then **forward_distance()** will use **ps[3]** and **ps[4]** from the **DistanceSensor[] ps** to return the final forward distance (same as other directions).

Notice that I changed the ps and timestep to static parameter because it will be convenient to deliver information to different functions (don't have problems from now on**,** If it does affect the codes, create other static parameters to get the value from these two).

How to judge turn left or right:

Once the front of sensors reports close to the wall, then call the **judgeLeftRight()**, it will let the robot turn(set the **state**) to the farthest side, as I mentioned before.

All movements will be managed in command(MoveState state) functions. When turning left or right, similar to Lab2, the **ICR** would remain in the centre of the axel between the wheels, let two wheels with the opposite but same value of rotate speed, and let the robot turn left or right with 90°.

However, the π/4 circle still may lose some details, so I set robots to turn around then turn 90°, which means **turn π5/4 circle**. This will significantly enhance some unscanned areas and walls around the robot.

How to keep direction & void walls

Once the robot is in a **FORWARD** state, **keepOrientation()** will ensure the robot keeps the correct direction and avoids hitting the wall.

This is a very important part of my development. Due to the dynamic friction and other reasons, which will lead to **Motion Errors**, the robot will not turn exactly 90° or stop and go forward as expected.

So it is essential to have a function to fix these. It can also save much time to calculate and complex code tests. When the robot cannot turn precisely the way we want or have others disturbed, **keepOrientation()** will force the robot to prescribed directions.

Another Enum list will represent the robot's orientation since the robot will only need to go vertically, so there are only four possible directions the robot will have:

```
public enum Orientation {
  NORTH, SOUTH, EAST, WEST
};
```

Corresponds to robot's rotate sensor are North=π/2, South=-π/2, East=0, West=+-π;

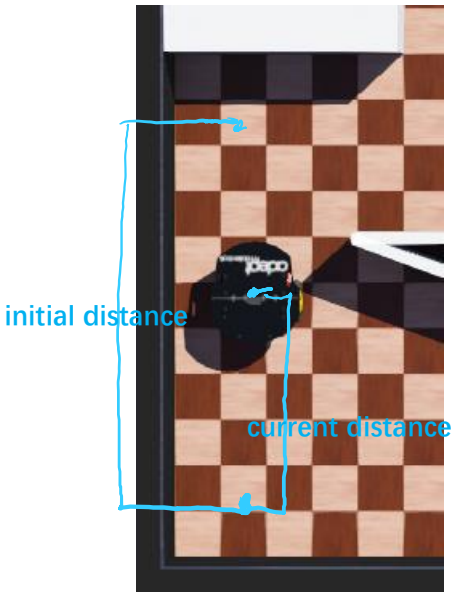
For example, if the previous orientation is towards North, once the robot detects face a wall and decide to go left, then set **Orientation** to **WEST**, a function called **judgeOrientation()**

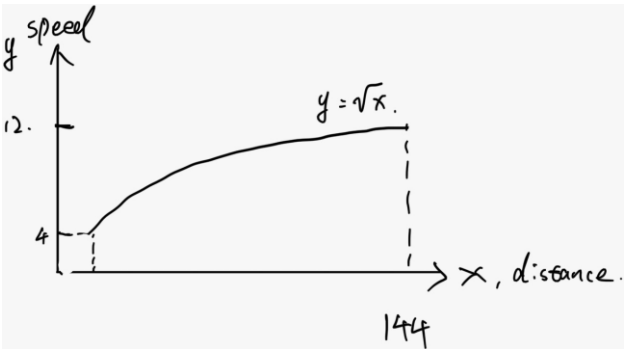
will manage this, then robot goes **FORWARD**, and until finding the next wall, **keepOrientation()** will make sure robot does not exceed +-**π range**.

Also, during the robot go **FORWARD**, to avid hit the wall, when the left/right side sensors are too small, **keepOrientation()** will let the robot way from that side a bit. This time will let two wheels with **the same direction but the different values of rotate speed,** so **ICR** will not be at the centre of the robot, making the robot yaw a bit.

Simple partial PID control

To let the robot, go forward as fast as possible and avoid the robot turning over when





immediately damping, so I set a function, which y is wheel speed(m/s), **x=current distance between front wall/initial distance** at the beginning, then let speed smooth: **y=**$\sqrt{x}$. Make sure the max speed is smaller than 12.3m/s (Pioneer 3-DX Model), and the lowest speed such as 4m/s, which is suitable for the robot to handle damping.

How to avoid trapped in cycles

The robot will work well with the above logic, but when it goes to the right side of the map, as the Fighre1 shows, it will face a **dead loop** because the left side distance is always larger than the right side.

To solve it, I will use an *ArrayList* to store turned locations when the robot Finnish a turn. If the zone appears again and can also turn left and right, I will keep them in another *ArrayList*.

Finally, find the farthest one between the endpoint using Manhattan Distance in this list. Once the robot reaches this point again (3 times), turn to another site instead.

—

Please find more details in codes and comments

Reflection and summary

Users need to set the initial **state** and **orientation** at the beginning manually.

My codes might be a bit complex that may be hard to refactor to another more straightforward logic solution, it could be simplified, however.

But it still has many valuable numbers of methods, also with some simple algorithms, so it works well in the current map. It can be used on many other maps with similar layouts. However, it cannot guarantee works well if the style of maps is changed. My robot can only go vertically and might face some unique situations that will waste more time overcoming a long-term dead loop or even miss some areas and hitting the sloping wall. Also, it is still possible to trap into some particular cycle for other maps.

I enjoyed this module and assignment sincerely.