Zeid Kootbally
Fall2022
UMD
College Park, MD

RWA2 (v0.1.1)

ENPM809Y: Introductory Robot Programming

Due date: **Monday, November 07, 2022, 5 pm**

Contents

1 Updates

This section describes updates added to this document since its first released. Updates include addition to the document and fixed typos. The version number seen on the cover page will be updated accordingly. There may be some mistakes even after proofreading the document. If this is the case, please let me know.

- **v0.1.1**.
  - Due date is changed. I would like to give you the weekend to finish and polish the assignment.
  - Added a new item in Section 13 (see item #3). This item was mentioned in the document but not clearly stated as a deliverable.
- **v0.1.0**: Updated Figures 5(a) and 10(a). The depictions of the following algorithm were not correct for the left-hand rule.
- **v0.0.0**: Original release of the document.

2 Conventions

In this document, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

- This is a [file.txt]
- This is a folder
- Best practice.
- Important note.
- Reminder.
- To do.
- Warning.

3 Group Assignment

This assignment can be done as a group of 3 students or less. Create your groups by filling out the ›Google sheet (add your name and group number).

4 Introduction

This assignment consists of implementing the "wall following" algorithm to drive a robot to a goal. The goal is always reachable and is always adjacent to one of the 4 outer walls of a maze. For this assignment you will need to use a third-party library which provides a simulator to visualize the outputs of your program. This simulator also provides functionalities to detect walls, set walls, change the color of cells, and set text.

5 Package

A package is provided for this assignment and has the structure shown in Figure 1. Add folders and files to this package and submit the whole package.

5.1: Package.

- Download the package: `>_ git clone https://github.com/zeidk/FALL2022_RWA2.git`
- Rename rwa2_groupX by replacing X with your group number.
- Rename [DoxyfileGroupX] by replacing X with your group number.

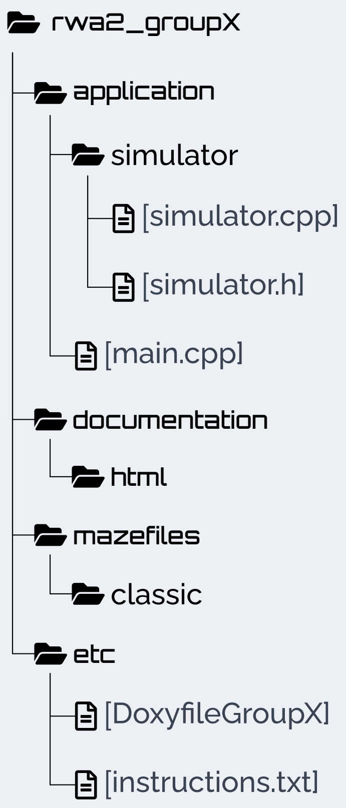


Figure 1: Package structure for RWA2.

Here is a description of the content of the package.

- application can only contain subfolders with [*.cpp] and [*.h] files. Your [*.cpp] and [*.h] files have to be placed in subfolders and not directly in application. The only source file that can be directly placed in this folder is [main.cpp].
- documentation contains HTML documentation generated from Doxygen. This folder currently contain the HTML documentation for the code found in the folder simulator. You can overwrite these HTML files when you generate the final documentation for your code.
- mazefiles contains the subfolder classic, which in turn contains mazes that can be loaded in the simulator.

- `etc` contains other types of file. This folder currently contains the Doxyfile that was used to generate the HTML files located in `documentation`. This folder also contains the file [instructions.txt] which you need to update before submitting your package.
- [main.cpp] contains the `main()` function. This function currently has some examples of methods from the class `Simulator`.

6 Simulator

A simulator is graphical user interface (GUI) which can be used to visualize the output of a search algorithm. The simulator used in this assignment can be found at ›https://github.com/mackorone/mms.

Download the simulator. In the instructions below the simulator is downloaded in the Home directory.

```
cd
```
(this will take you to your Home directory).

```
git clone https://github.com/mackorone/mms.git
```

6.1 Install Qt

The simulator needs to be built with Qt before it can be used.

- Create a Qt account for free at ›https://www.qt.io (click on the small round icon on the top-right corner of the screen).
- To install Qt, see ›instructions for Windows, Mac, and Linux. For Linux users, you need to go with Option #2. Here are more up-to-date instructions for Ubuntu:
  1. Download the Qt open source installer: ›https://www.qt.io/download-qt-installer. The file [qt-unified-linux-x64-4.4.2-online.run] will be downloaded. We assume in the following instructions that you downloaded this file in your Home folder.
  2. `cd`
  3. Make this file executable: `chmod +x qt-unified-linux-x64-4.4.2-online.run`
  4. Run the installer: `./qt-unified-linux-x64-4.4.2-online.run`
     - **Welcome**: Enter your login and password for your Qt account. Press **Next**.
     - **Open Source Obligations**: Check both boxes. Press **Next**.
     - **Setup-Qt**: Press **Next**.
     - Select whatever you want in the next window. Press **Next**.
     - **Installation Folder**: Select **Design Tools** and **Qt 6.4 for desktop development**.
     - The remaining instructions are straightforward.
     - When the installation is done, uncheck all boxes and press **Finish**.

6.2 Building the Executable

After the installation, the tool `qmake` can be found at `/home/user/Qt/6.4.0/gcc_64/bin` where `user` should be replaced with your actual Linux user name.

Build the simulator:

```
cd ~/mms/src
/home/user/Qt/6.4.0/gcc_64/bin/qmake && make
```

If everything worked fine, the `mms` executable can be found in `~/mms/bin`.

6.3 Configure the Simulator

Configure the simulator to run your program.

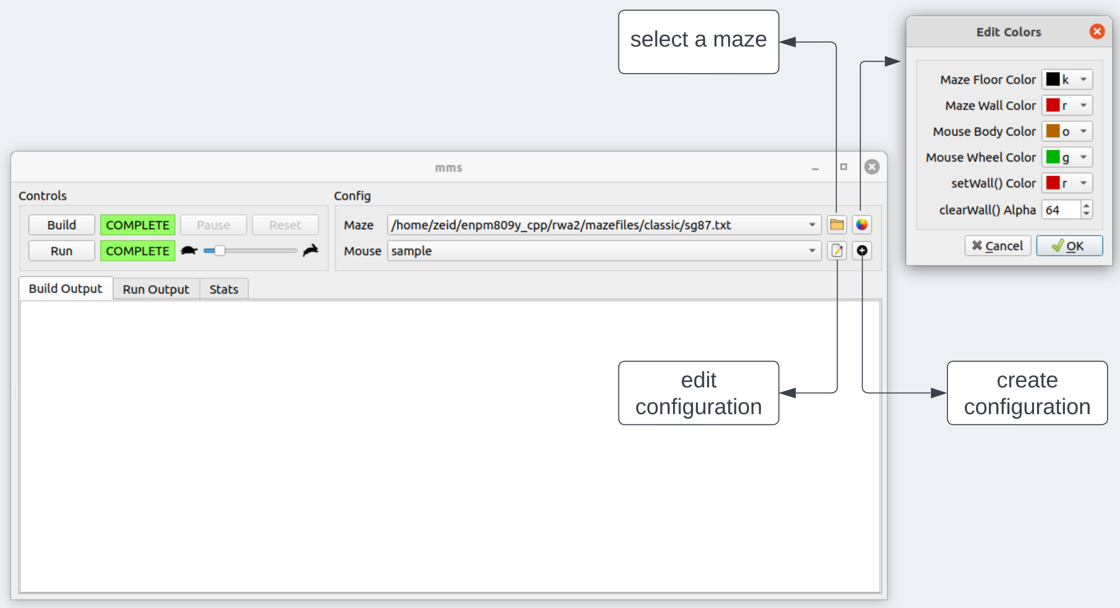
- Run the simulator:

  ```
  ~/mms/bin/mms
  ```

- Figure 2 shows the different options.
  - You can select the maze file to load in the simulator. Any file located in the folder `mazefiles/classic` will work.
  - You can edit colors for the maze and the robot.
  - Create a configuration to open a new window (see Figure 3).
    - ◇ Name – Use an arbitrary name to this configuration. Different names can be used to create different configurations.
    - ◇ Directory – Browse to the root directory of your package. This should be the directory `rwa2_groupX` in Figure 1.
    - ◇ Build Command – Command to run to generate the executable. You only need to provide relative paths to all your [*.cpp] files. Your program will not build if this field is missing some [*.cpp] files. The last part of the command **-o main** is the name of the executable.
    - ◇ Run Command – Command to run your program. Run the executable.
  - You can also edit the configuration you just saved.

6.4 Build and Run

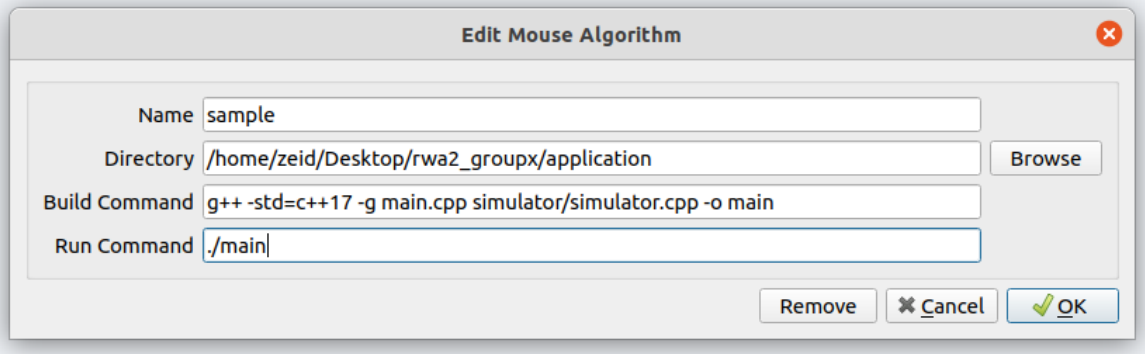
Once you are done with the project configuration, build your program with the button Build (see Figure 2). Any problem with the build will appear in the tab Build. If the build is successful, you can run your program with the button Run.



Maze /home/zeid/enpm809y_cpp/rwa2/mazefiles/classic/sg87.txt
Mouse sample

Maze Floor Color	k
Maze Wall Color	r
Mouse Body Color	o
Mouse Wheel Color	g
setWall() Color	r
clearWall() Alpha	64

Figure 2: Configure the simulator.



Name	sample
Directory	/home/zeid/Desktop/rwa2_groupx/application
Build Command	g++ -std=c++17 -g main.cpp simulator/simulator.cpp -o main
Run Command	./main

Figure 3: Configure the program.

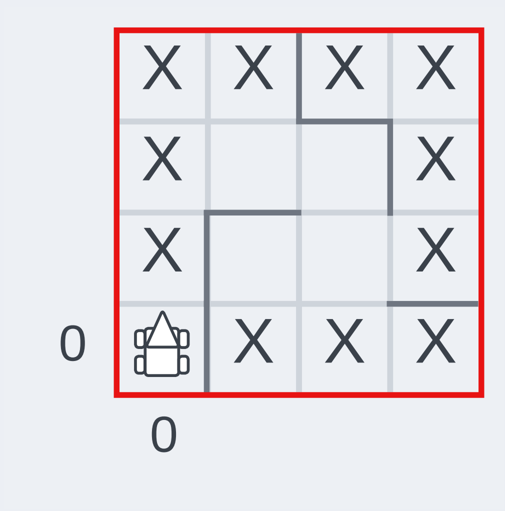


Figure 4: X shows the different possibilities for a goal in a 3×3 maze. Note that the position (0,0) cannot be a goal.

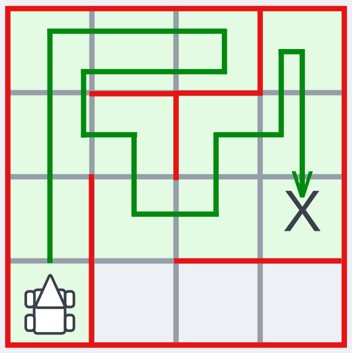
7 Wall Following Algorithm

The “wall following” algorithm can be used to reach a goal as long as the goal is adjacent to one of the outer walls of the maze (see Figure 4). Using this algorithm, the robot follows either the right or left wall to reach the goal.

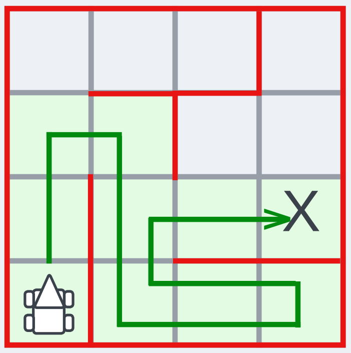
There are two types of “wall following” algorithm [1]: left-hand rule and right-hand rule. The two algorithms work the same way except turning priority will be either to the left or to the right depending on the type of rule used. An example of pseudo-code for “wall following” using left-hand rule is shown in Algorithm 7.1. You can easily write the right-hand rule counterpart from the left-hand rule version. The result of the left-hand rule and the right-hand rule approaches can be seen in Figure 5.

Algorithm 7.1: Left-hand rule for the “wall following” algorithm.

```
while not current cell is G:
    if left is open:
        turn left
    elif front is open:
        go forward
    elif right is open:
        turn right
    else:
        turn around
```



(a) Left-hand rule.



(b) Right-hand rule.

Figure 5: Results of the "wall following" algorithm using the left-hand rule approach (a) and the right-hand rule approach (b).

⚠ 7.1: Move forward after turning.

When you implement this algorithm using the methods from the class `Simulator` you need to move the robot forward after you turn right or left. In other words, explicitely call `move_forward()` after `turn_right()` and after `turn_left()`.

8 Application

There are two parts to this assignment. The first part is about writing the structure of your program (classes, attributes, and methods) and the second part is about testing and running your program. Your program will not be able to generate the correct outputs if it does not interact with the simulator. To implement the "wall following" algorithm you need to discover walls in the maze so your robot can go around them. This is only possible with the methods `wallFront()`, `wallLeft()`, and `wallRight()` from the `Simulator` class. These methods use the current position and orientation of the robot in the maze to check for walls. To move your robot in the maze, you also need to use methods from the `Simulator` class, namely, `moveForward()`, `turnLeft()`, and `turnRight()`.
Your program should represent maze and robot information in order to implement the wall follower approach.

8.1 Maze

Your program should be able to work on any maze that has the following characteristics (refer to Figure 6).

- The maze dimension is 16×16.
- The maze has outer walls that are closed (shown in purple in Figure 6).
- The position $(0,0)$ is the cell located in the lower left corner of the maze.

8.1: Testing with different mazes.

All the maze files in the folder mazefiles/classic have those characteristics and your program will be tested on some of these files.

8.2 Robot

The robot can navigate the maze with `moveForward()`, `turnRight()`, and `turnLeft()`.

8.1: Navigation methods.

`moveForward()` moves the robot only in the next cell. If you want the robot to keep moving forward, you need to call `moveForward()` multiple times. `turnLeft()` rotates the robot 90 deg CCW only once in the current cell and `turnRight()` rotates the robot 90 deg CW only once in the current cell. If you want the robot to perform multiple in-place rotations then you need to call `turnRight()` or `turnLeft()` multiple times.

At the start of simulation, the robot is always at position $(0,0)$ and faces the North direction. When your program gives commands to the robot to move forward and to turn, you need to keep track of the current position of the robot and its direction after each move. Keeping track of the position and the direction of the robot will allow you to set walls, to modify the color of cells, and to store the path of the robot (more on this in Section 12).

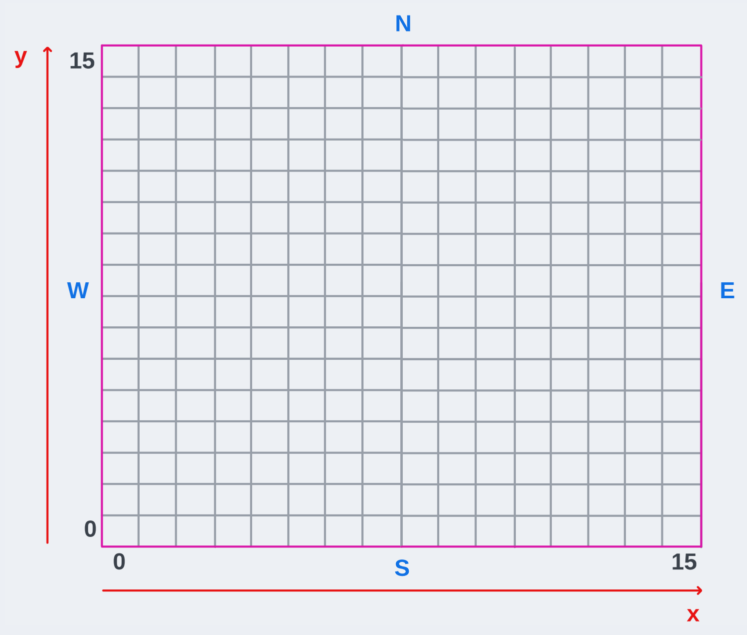


Figure 6: Characteristics for mazes used in this assignment.

Here are some examples describing how a move can affect the robot's position and orientation.

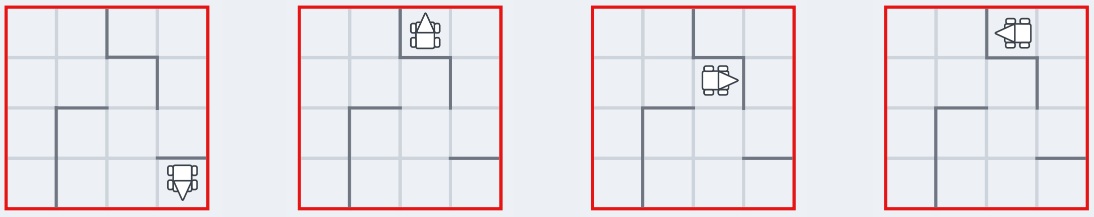
- If the robot is at position (x, y) and its direction is North, then moving the robot one step forward will take the robot to position $(x, y+1)$. If the robot is at position (x, y) and is facing East, moving the robot one step forward will take the robot to the position $(x+1, y)$. From these two examples you can figure out the next position of the robot based on its orientation.
- If the robot is facing North and it turns left then its new direction is West. If the robot is facing South and it turns left then its new direction is East. From these examples you can figure out the next orientation of the robot based on its current orientation.

8.2.1 Detecting Walls

The robot has no knowledge of walls until the robot looks for walls. The robot can discover walls with the methods `wallFront()`, `wallLeft()`, and `wallRight()`. The simulator uses the robot position and direction to check for walls. Figure 7 shows examples where `wallFront()`, `wallLeft()`, and `wallRight()` return `true`, i.e., walls are detected.

8.2.2 Setting Walls

Once a wall is found, your program will need to "set this wall" with the method `setWall(int x, int y, char direction)`. Setting a wall means changing the color of the wall from gray to red. This method is used only for visualization and is a good way to see which walls the robot has detected. This method needs the coordinates of a cell (x, y) and one of the 4 walls of a cell, represented by the parameter `direction`. The latter can take one of the 4



(a) Left and front walls detected. (b) Left and front walls detected. (c) Left and front walls detected. (d) Left, right, and front walls detected.

Figure 7: `wallFront()` and `wallLeft()` return `true` in (a), (b), (c), and (d). `wallRight()` returns `true` in (d).

following values: `'n'`, `'s'`, `'e'`, or `'w'`. The 4 directions of a cell is shown in Figure 8.

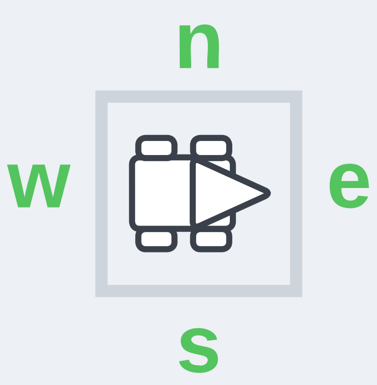
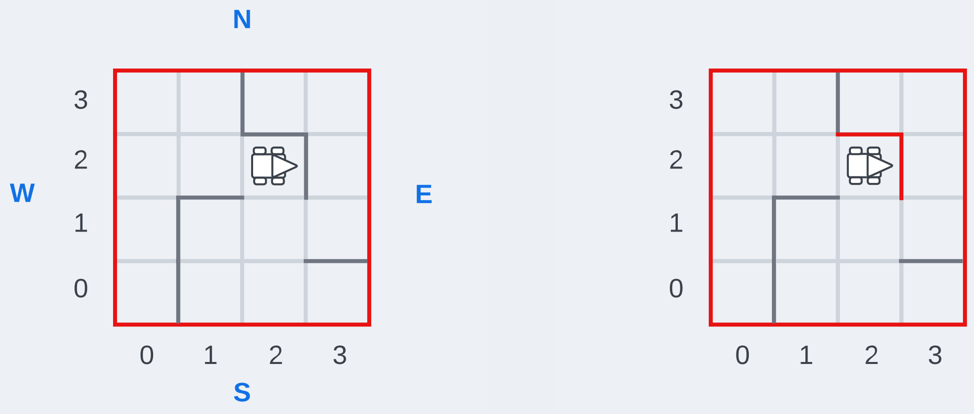


Figure 8: Walls of a cell and their value.

The coordinates of the cell in this case correspond to the coordinate of the robot and `direction` is the direction of the robot. Figure 9(a) shows a situation where the left wall and the front wall are detected (`wallLeft()` and `wallFront()` return `true`). To set the front wall, the method `setWall(2,2,'e')` needs to be called. Since the robot is facing East, the front wall in this case is the East wall of the cell (2,2). To set the left wall, the method `setWall(2,2,'n')` needs to be called. Since the robot is facing East, the left wall in this case is the North wall of the cell (2,2). The current position (e.g., (2,2)) and the direction of the robot cannot be retrieved from the maze. The only way to know this information is to keep track of the robot's moves from the start.



(a) `wallFront()` and `wallLeft()` return `true`.

(b) `setWall(2,2,'n')` sets the left wall and `setWall(2,2,'e')` sets the front wall.

Figure 9: Once walls are detected, set the walls to change their colors from gray to red.

9 Requirements

The following is required for this assignment.

- You have to use OOP. Think about the design of your program before you start writing any code. OOP is different from imperative programming. You will see yourself stuck if you do not work on the design first.
- You have to enclose all your classes within the namespace `rwa2groupx` where `x` is your group number.
- You have to document your classes and methods using Doxygen documentation. You can edit the file [DoxyfileGroupX] located in the folder `etc`.
  - On Linux, you can open this file with the command:
    ```
    doxywizard DoxyfileGroupX
    ```
- You have to use smart pointers to store class instances on the heap. If you are using shared pointers instead of unique pointers, provide an explanation as to why in the documented code.
- You have to generate the html documentation in the folder `documentation` (you can replace the existing html files in this folder).

10 Classes

This section provides minimal examples of classes that can be used for this assignment. You do not have to implement your program using these examples as there are multiple ways to implement an application in OOP. However, these examples can give you some ideas.

- Before writing any code, try some methods from the class `Simulator` to see what they do and how they work.
- A robot object from the class `Robot` has a position and a direction. A robot can turn left, turn right, and move forward.

Listing 10.1: Robot class.

```
namespace rwa2groupx {
class Robot {
   public:
    void turn_left();
    void turn_right();
    void move_forward();
   private:
    std::pair<int, int> m_position;
    char m_direction;
};
}  // namespace rwa2groupx
```

- A cell object from the class `Cell` has 4 walls and each wall can be open or closed.

Listing 10.2: Cell class.

```
namespace rwa2groupx {
/**
 * @brief This class represents a cell in the maze.
 */
class Cell {
   public:
    // initialize all 4 walls of a cell
    void init_cell_walls();
    // set one of the walls based on is_wall
    void set_wall(int wall, bool is_wall);
    // return true if wall is closed
    bool is_wall(int wall);

   private:
    //!<@brief The walls of the cell.
    //!< true means the wall is open
    //!< false means the wall is closed
    //!< 0 = North, 1 = East, 2 = South, 3 = West
    std::array<bool, 4> m_walls;
};
}  // namespace rwa2groupx
```

- An algorithm object from the class `Algorithm` consists of a maze, a robot, and a goal to reach. Whenever the robot discovers a wall in the maze, use the robot position and orientation to set the walls.

Listing 10.3: Algorithm class.

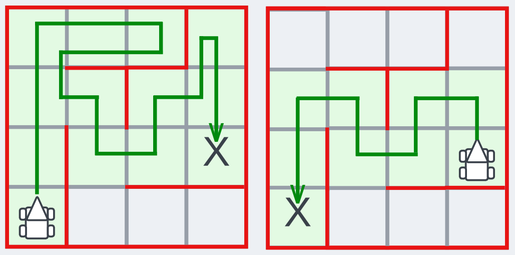
```
namespace rwa2groupx {
/**
 * @brief This class implements the search algorithm.
 */
class Algorithm {
   public:
    // initialize outer walls, generate goal, execute search algorithm, etc
    void run();
    // color outer walls
    void init_outer_walls();
    // algorithm for left-hand rule approach
    void follow_wall_left();
    // algorithm for right-hand rule approach
    void follow_wall_right();
    // generate random goal adjacent to an outer wall
    void generate_goal();
    // set right wall in m_maze and in simulator
    void set_right_wall();
    // set left wall in m_maze and in simulator
    void set_left_wall();
    // set front wall in m_maze and in simulator
    void set_front_wall();

   private:
    //!<@brief A maze is a 2D array of cells.
    std::array<std::array<Cell, 16>, 16> m_maze;
    std::unique_ptr<Robot> robot;
    std::pair<int, int> goal;
    int m_maze_height;
    int m_maze_width;
};
}  // namespace rwa2groupx
```

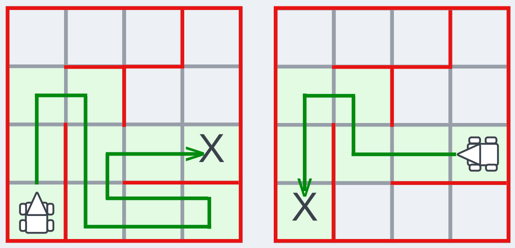
11 Implementation

Running your program must achieve the following:

1. Write a function to generate a random goal position. The goal has to be adjacent to one of the outer walls. If the X coordinate of the goal is 0 or 15 then its Y coordinate is a number between 0 and 15. If the X coordinate is not 0 or 15, then its Y coordinate is 0 or 15.
   - Mark the goal cell with the letter G: Use the method `Simulator::setText()`.
   - Set the color of the goal cell to red: Use the method `Simulator::setColor()`.
2. Set the outer walls using the method `Simulator::setWall()`. Set the walls using a double `for` loop.
3. Implement the “wall following" algorithm for both the right-hand rule and the left-hand



(a) Use the left-hand rule approach to reach the goal then drive the robot back to its original position by avoiding dead ends.



(b) Use the right-hand rule approach to reach the goal then drive the robot back to its original position by avoiding dead ends.

Figure 10: The robot should drive back to its original position by using and optimizing the previously stored path.

rule. Each run uses one of the two algorithms, you cannot run both at the same time. When your assignment is graded, we will test the left-hand rule method first and then re-run your program using the right-hand rule method.

- Use `Simulator::setColor()` to color the robot cell in `'c'` (cyan) while the robot is following the wall.

4. If you implemented the functionality described in Section 12, once the robot has reached the goal, clear all the cells you colored in step 3. Color the cells from the robot current position to $(0,0)$ and drive your robot to $(0,0)$.

11.1: Printing in the simulator.

Printing text in the simulator with `std::cout` will not work. The author of the simulator mentions the use of `std::cerr` instead.

12 Extra Credit

Drive the robot back to $(0,0)$ with the following conditions:

- Do not use the "wall following" algorithm or any search algorithm.
- Use the stored path the robot took to drive from $(0,0)$ to the the goal.
- Optimize this path by removing paths that lead to dead ends. See examples in Figure 10.
- Students will get either 0 or 7 pts as extra credit, no points in-between. If driving back the robot to the original position works for the left-hand rule approach but not for the right-hand rule approach then no points are granted. If the approach works once in a while but not all the time then no points are granted. If the approach works with some mazes but not with other mazes then no points are granted.
- Extra credit will be applied to the current assignment if students do not have full points already, otherwise it will be applied to the next assignment.

13 Deliverables

- Hard deadline: No extension is granted for this assignment unless you provide a documented excuse (doctor note, note from your supervisor, etc).
- Submit the zipped package by making sure the following is performed:
  1. Modify the package name appropriately.
  2. Modify the name of the file [DoxyfileGroupX] appropriately and replace the original file with the one you have used to generate the HTML documentation.
  3. It was mentioned that two versions of the algorithm should be implemented (right-hand and left-hand rule approaches). You need to tell us how we can run each version. One suggestion is to use a flag to call each version. For instance, in Figure 3, you could use **./main -right** to start the right-hand rule approach and **./main -left** to start the left-hand rule approach. In the lecture on Functions we saw how to retrieve command-line arguments in the `main(int argc, char** argv)` function (slide 57).
  4. The HTML documentation for your package should be placed in the folder documentation.
  5. Modify the namespace using your group name.
  6. Edit [instructions.txt] and include the command to run to build your package. This is the command you have in the field Build Command in Figure 2. Also, describe what we need to change in your program to switch between the left-hand rule approach and the right-hand rule approach.
  7. All the [.h] and [.cpp] should be in a subfolder under application. It is a good idea to build and execute your program one last time before you zip and upload your package.
  8. Once you have submitted your assignment you cannot submit again, even if you have a small typo in your program. Make sure you tested everything before you submit.
  9. Canvas rules will apply to late submissions. 10 % deduction per day for late submissions.

14 Grading Rubric

15 pts Your submission will be tested on 2 different mazes. Full points will be awarded if the goal is properly generated and the robot reaches the goal during each run. Your submission will be tested with the right-hand rule approach and the left-hand rule approach.

15 pts Correct implementation and documentation are important. Correct implementation includes best practices seen in class (e.g., `private` attributes but `public` methods). Correct documentation includes class and methods documentation with Doxygen along with the generated HTML files in the folder documentation.

7 pts Extra credit.

References

[1] Abu Bakar Sayuti Saman and Issa Abdramane. Solving a Reconfigurable Maze using Hybrid Wall Follower Algorithm. International Journal of Computer Applications, 82(3):22–26, 2013.