Zeid Kootbally
University of Maryland
College Park, MD

zeidk@umd.edu

RWA-2: Maze Solving Algorithm (v1.2)

ENPM809Y : Fall 2021
Due **Saturday, November 13, 2021, 11 pm** **(Section I)**
Due **Saturday, November 13, 2021, 11 pm** **(Section II)**

Contents

1 Prerequisites

- You need to be part of a group of 3 students (or less) for this assignment.
- A micromouse simulator to visualize the output of your algorithm.
- A collection of maze files.

2 Objectives

- Write a depth-first search (DFS) algorithm to help a robot reach the center of a maze.
- Understand how to use a third-party library with your project.
- Visualize the output of your algorithm using the third-party library.

3 Background

Micromouse is an event where small robot mice solve a 16×16 maze. The mouse needs to keep track of where it is, discover walls as it explores the maze, and map out the maze and detect when it has reached the goal. Having reached the goal, the mouse will typically perform additional searches of the maze until it has found an optimal route from the start to the finish. Once the optimal route has been found, the mouse will run that route in the shortest possible time.
The aim of this assignment is to perform only the first part of the competition, which is making the mouse reach the goal. Finding the most optimal path and running a route in the shortest possible time are not the focus. Students will need to implement a depth-first search algorithm and will then visualize the result in the micromouse simulator.

Take a look at some impressive runs (the crazyness starts at the 1:20 min mark).

4 Setup

The following steps are required to download, compile, run, and test the micromouse simulator.

4.1 Get Required Files

1. Create a folder where you will download the micromouse controller. In this example the folder is created in the home directory but it can be created anywhere else.

   ```
   mkdir ~/RWA2_simulator
   ```

2. Clone the micromouse simulator in this new folder:

   ```
   cd ~/RWA2_simulator
   git clone https://github.com/mackorone/mms.git
   ```

3. Clone a set of maze files:

   ```
   git clone https://github.com/micromouseonline/mazefiles.git
   ```

4. Clone a test C++ program which simply turns the robot left and right:

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

4.2 Compile the Simulator

1. Compile the simulator.

   - Install QT: `sudo apt-get install qt5-default`
   - Build the simulator.

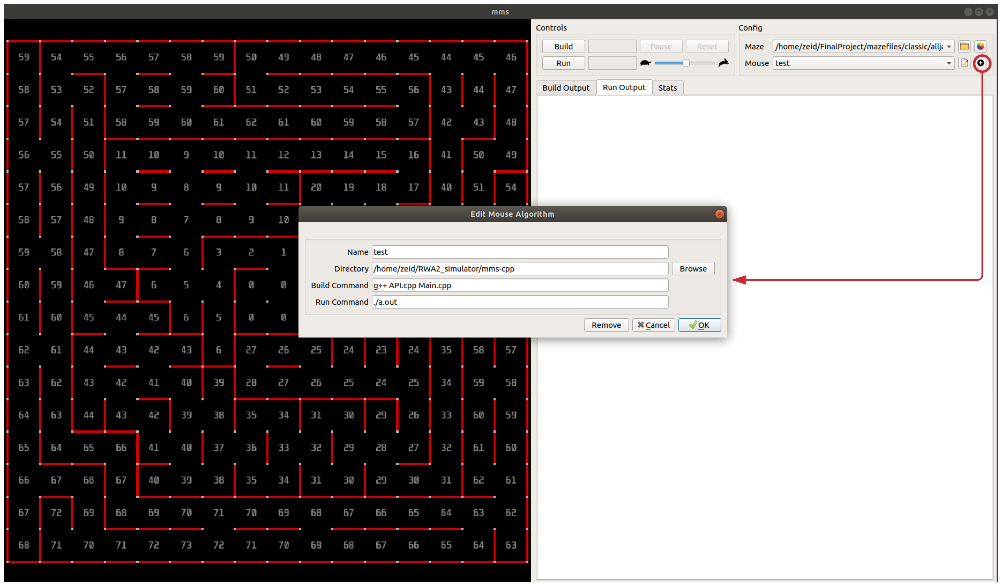
     ```
     cd mms/src
     qmake && make
     ```

     ∗ Contact me if you receive the following error: `/usr/bin/ld: cannot find -lGL`

4.3 Run the Simulator

```
cd ~/RWA2_simulator/mms/bin
```

`./mms` will open the simulator (Figure 1).



Maze: /home/zeid/FinalProject/mazefiles/classic/allj
Mouse: test

Name: test
Directory: /home/zeid/RWA2_simulator/mms-cpp
Build Command: g++ API.cpp Main.cpp
Run Command: ./a.out

Figure 1: Micromouse simulator (mms).

4.4 Configure New Mouse Algorithm

Clicking the + button in Figure 1 opens a new window to set up the algorithm. Here is how to configure the different fields:

- **Name**: This can be anything (`test` in this example).
- **Directory**: Click on the `Browse` button and navigate to the folder **mms-cp**, which you cloned earlier (see Figure 2).
- **Build Command**: Command to compile the project. In **mms-cp** there are two .cpp files which need to be compiled with:

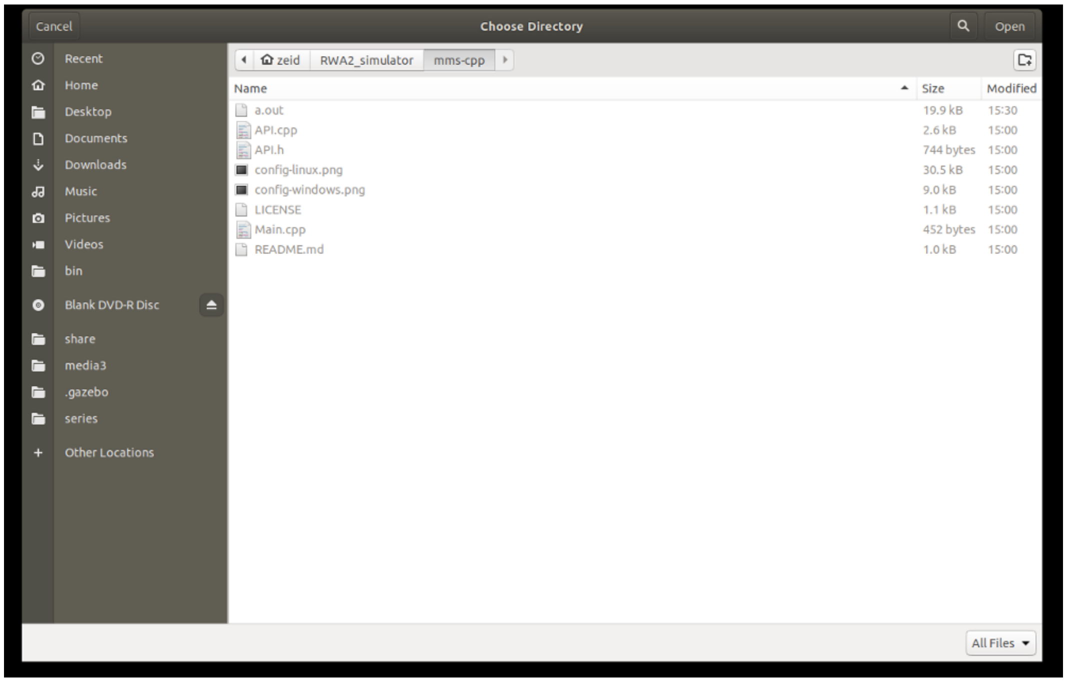
  ```
  g++ API.cpp Main.cpp
  ```

- **Run Command**: The previous step generates the executable `a.out`. Run this executable with:

  ```
  ./a.out
  ```

4.5 Build and Run

1. Click on the `Build` button to build your project and generate `a.out`.
2. Click on the `Run` button to run the executable `a.out`. You should see the mouse moving. The algorithm used by the mouse is implemented in Main.cpp.



Name	Size	Modified
a.out	19.9 kB	15:30
API.cpp	2.6 kB	15:00
API.h	744 bytes	15:00
config-linux.png	30.5 kB	15:00
config-windows.png	9.0 kB	15:00
LICENSE	1.1 kB	15:00
Main.cpp	452 bytes	15:00
README.md	1.0 kB	15:00

Figure 2: Source directory.

5 Simulator API

The methods to move the mouse, check for walls, set cell colors, etc, can be found on the simulator Github page. Students are required to read the documentation before moving forward with the assignment.
Main.cpp (located in the folder **mms-cpp**) contains a very simple program to show the user how to apply some of the methods that come with the API. The content of Main.cpp is presented in Figure 3 and is discussed below.

```cpp
1  #include <iostream>
2  #include <string>
3
4  #include "API.h"
5
6  void log(const std::string& text) {
7      std::cerr << text << std::endl;
8  }
9
10 int main(int argc, char* argv[]) {
11     log("Running...");
12     API::setColor(0, 0, 'G');
13     API::setText(0, 0, "abc");
14     while (true) {
15         if (!API::wallLeft()) {
16             API::turnLeft();
17         }
18         while (API::wallFront()) {
19             API::turnRight();
20         }
21         API::moveForward();
22     }
23 }
```

Figure 3: Main.cpp

- `line 4`: API.h is the interface of the simulator. It contains all the C++ methods to interact with the simulator.
- `lines 6-8`: A user-defined function to output text in the simulator. Note that `std::cerr` is the only way to print messages in the right pane of the simulator while `std::cout` is used to interact with the maze (left pane).
- `line 12`: This line sets the color of the cell $(0,0)$ to green.
- `line 13`: This line writes "abc" in the cell $(0,0)$.
- `lines 14-22`: Algorithm used to move the mouse: Rotate left if there is no wall on the left. Rotate right if there is no wall in front of the robot. After rotating, move forward. As one can see, this algorithm does not really attempt to solve the maze. It only provides an example of how to use some of the methods that come with the API.
  - `wallLeft()` returns true if there is a wall on the left of the mouse, otherwise it returns false.
  - `turnLeft()` makes the mouse rotate 90° CCW.

  - `wallFront()` returns true if there is a wall in front of the mouse, otherwise it returns false.
  - `turnRight()` makes the mouse rotate 90° CW.
  - `moveForward()` makes the mouse move forward. After the mouse rotates, a `moveForward()` must be issued, otherwise the mouse will stay idle.

6 Search Algorithm

Finding a path in a maze is similar to finding a path from the root of a tree (start position in the maze) to a leaf of the tree (goal position in the maze). Each cell in the maze is a node in the tree and each new cell you can go to from the current node is a child of that node.

In this assignment, students will implement a depth-first search (DFS) algorithm. DFS does not always generate an optimal solution and will usually not generate the shortest path. The algorithm starts at the root node and explores **as far as possible** along each branch before backtracking. To traverse a maze with DFS, a search order is required. In this assignment, the order is as follows:

- ↑: Check North.
- →: Check East.
- ↓: Check South.
- ←: Check West.

6.1 Backtracking

When moving forward and there are no more nodes along the current path, move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path until all the unvisited nodes have been traversed after which the next path will be selected.

6.2 Pseudocode

DFS can be implemented using stacks with the last-in-first-out (LIFO) approach. The stack will be used to add/remove nodes while the algorithm is traversing the maze. A list of visited nodes is required to prevent any infinite cycles. In C++, `std::stack` and `std::vector` can be used for the stack of nodes and for the list of nodes, respectively. The algorithm starts at the start node n and needs to find a path from n to g, where g is the goal node to reach. A pseudocode for recursive DFS is provided in Algorithm 1.

Algorithm 1: A recursive algorithm for depth-first search.

```
Function search_maze($n$): bool is
    Data: $s$ – stack of nodes (std::stack) to be initialized before calling this function
    Data: $v$ – list of nodes (std::vector) to be initialized before calling this function
    Data: $m\_maze$ – 2D array
    Data: $m\_goal$ – goal node
    Input: $n$ – current node
    Output: true if a path from $n$ to $g$ is found, otherwise false

    if not $n$ is $m\_goal$ then                         /* current node is not the goal */
        if $s$.empty() then
            $s$.push($n$);
        end
    end
    else                                                 /* current node is the goal */
        return true;
    end

    if not $n$ in $v$ then
        $v$.push_back($n$);
    end

    if $m\_maze[n.x][(n.y)+1]$ is valid then             /* check North */
        $n.y+=1$ ;
        $s$.push($n$);
    else if $m\_maze[(n.x)+1][n.y]$ is valid then        /* check East */
        $n.x+=1$ ;
        $s$.push($n$);
    else if $m\_maze[n.x][(n.y)-1]$ is valid then        /* check South */
        $n.y-=1$ ;
        $s$.push($n$);
    else if $m\_maze[(n.x)-1][n.y]$ is valid then        /* check West */
        $n.x-=1$ ;
        $s$.push($n$);
    else                                                 /* no valid nodes: backtrack */
        if not $s$.empty() then
            $s$.pop();                                   /* remove $n$ from $s$ */
        end
        else
            return false;                                /* empty $s$ = $m\_goal$ not found */
        end
    end
    if not $s$.empty() then
        $n = s$.top();                                   /* current node is now the one at the top of $s$ */
        search_maze($n$);
    end
    else
        return false;                                    /* empty $s$ = $m\_goal$ not found */
    end
end
```

6.3 Example

This section describes the application of DFS algorithm for a 4x4 maze. The black lines in the maze represent walls that can not be crossed. $(0,3)$ is the start position in the maze and $(2,2)$ is the goal position. The objective is to use DFS algorithm to find a path from the start position to the goal position. A list of visited nodes and a stack of nodes will be updated as the algorithm is applied to the nodes. Figure 4 shows the steps of DFS applied to the maze until no more nodes are available. Figure 4 is described below.

0 and **1** – Start with the current node located at $(0,3)$ in the maze.

1. Check current node is goal: No.
2. Check stack is empty: Yes→Add current node $(0,3)$ to stack.
3. Check $(0,3)$ is in visited list: No→Add current node $(0,3)$ to visited list.
4. Check North is valid: No (wall).
5. Check East is valid: Yes→Move East and set $(1,3)$ as the current node.

2 – Start with the current node $(1,3)$ repeat the same process. Current node is now $(2,3)$.

3 – Start with the current node $(2,3)$ repeat the same process. Current node is now $(3,3)$.

4 – Start with the current node $(3,3)$ repeat the same process.

1. Check current node is goal: No.
2. Check stack is empty: No.
3. Check $(3,3)$ is in visited list: No→Add current node $(3,3)$ to visited list.
4. Check North is valid: No (wall).
5. Check East is valid: No (wall).
6. Check South is valid: Yes→Move South and set $(3,2)$ as the current node.

5 – Start with the current node $(3,2)$ and repeat the same process. Although G is very close to $(3,2)$, the algorithm will miss it since it has to follow the order specified. In this case, North is not valid (already visited), East is not valid (wall), South is valid. Current node is now $(3,1)$. **Note**: Students can try to improve DFS by first checking all adjacent nodes for the goal. If the goal is not part of the adjacent nodes then proceed as usual.

14 – Repeating the process, the algorithm reaches $(0,0)$. From $(0,0)$, North is not valid (already visited node), East is not valid (wall), South is not valid (wall), and West is not valid (wall). From here, the algorithm needs to backtrack to previous nodes until it finds a non-visited node. The backtracking phase is explained in Figure 5.

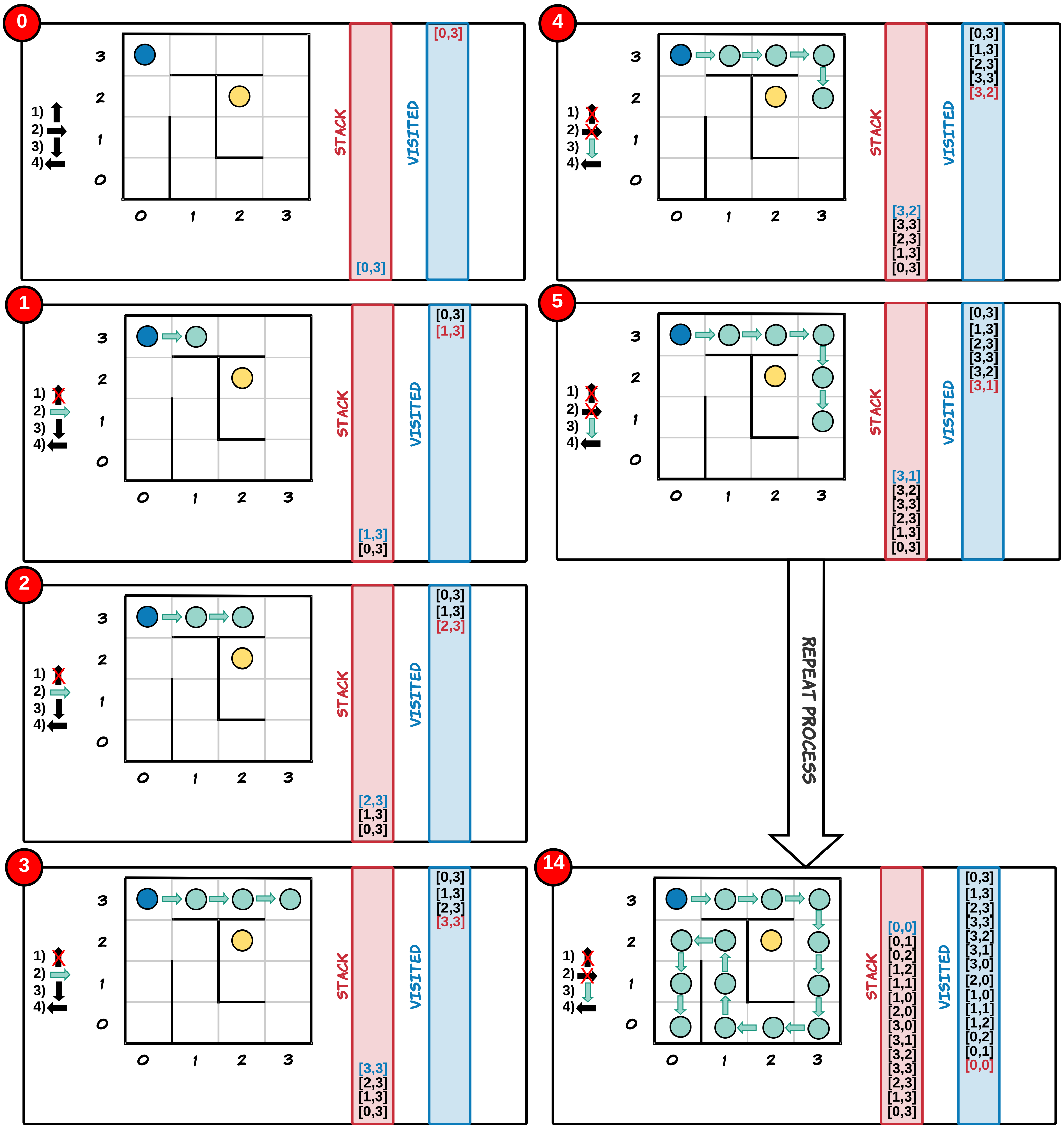


Figure 4: DFS: Forward.

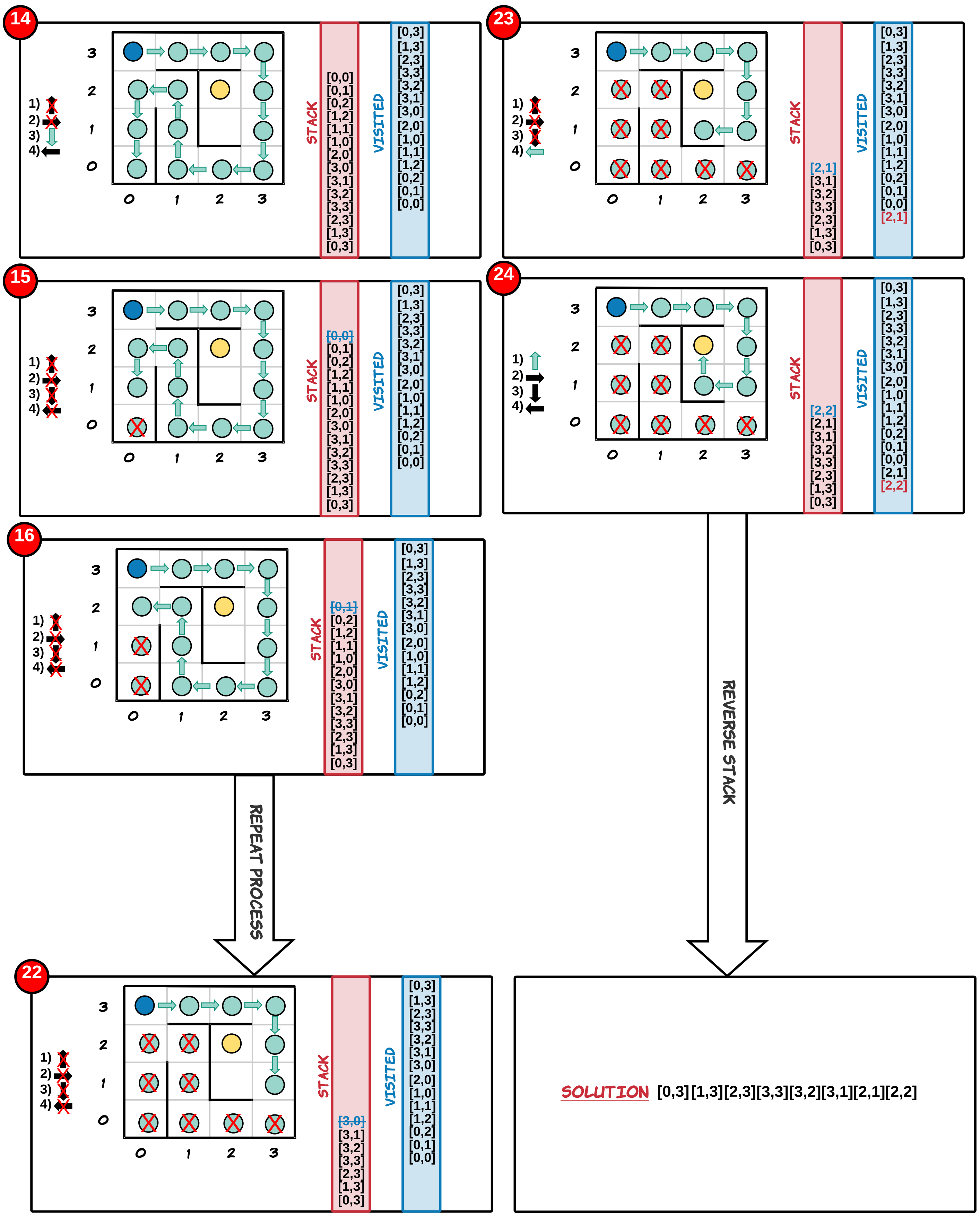


Figure 5: DFS: Backtrack.

6.4 Application

Algorithm 1 only computes a path from start to goal. The result now needs to be applied to the mouse to make it move along the path in the simulator. One important aspect of this assignment is that the robot has no *a priori* knowledge of the walls in the maze except for the boundary walls. In other words, while the mouse is moving towards the goal, it will encounter new walls which may make the current path obsolete. Algorithm 1 will need to be executed again including the newly discovered walls. When using the micromouse simulator, the only information known at the start of the simulation is:

- The mouse's current location in the maze: Always $(0,0)$.
- The mouse's direction in the maze: Always facing North.
- The goal location in the maze. Students need to pick one of the four goals to reach. The goals are: $G_1(7,7)$, $G_2(7,8)$, $G_3(8,7)$, and $G_4(8,8)$.
- The size of the maze: Always 16x16.
- Boundary walls.

Figure 6 describes the expected flow when running an RWA2 submission.

1 – The start position of the mouse is $(0,0)$ and the goal to reach is at $(1,2)$. At the start of the simulation the mouse only knows about the boundary walls and the walls around the node where the mouse is located. Known walls are colored in black and unknown walls are colored in orange. As one can see, the path generated does not include unknown walls as they have not been encountered by the mouse yet.

2 – The mouse follows the path. During path execution, the walls in each node visited by the mouse must be updated in the C++ program. Detected walls should also be displayed in the simulator. In this example, 4 new walls are discovered (previously orange). In node $(3,2)$ a wall prevents the mouse from following the original path. A new path has to be recomputed using the new discovered walls.

3 – Before recomputing a new path, the stack of nodes and the list of visited nodes must be emptied. If they are not emptied, DFS will not be able to explore other paths and a solution may not be found. In this current step, DFS is executed on *m_maze* with the 4 newly discovered walls.

4 – Use the simulator to move the mouse along the new computed path. In this example, the mouse reached the goal and the program stops.

Multiple alternations of path search and path following may be needed for the mouse to reach the goal.

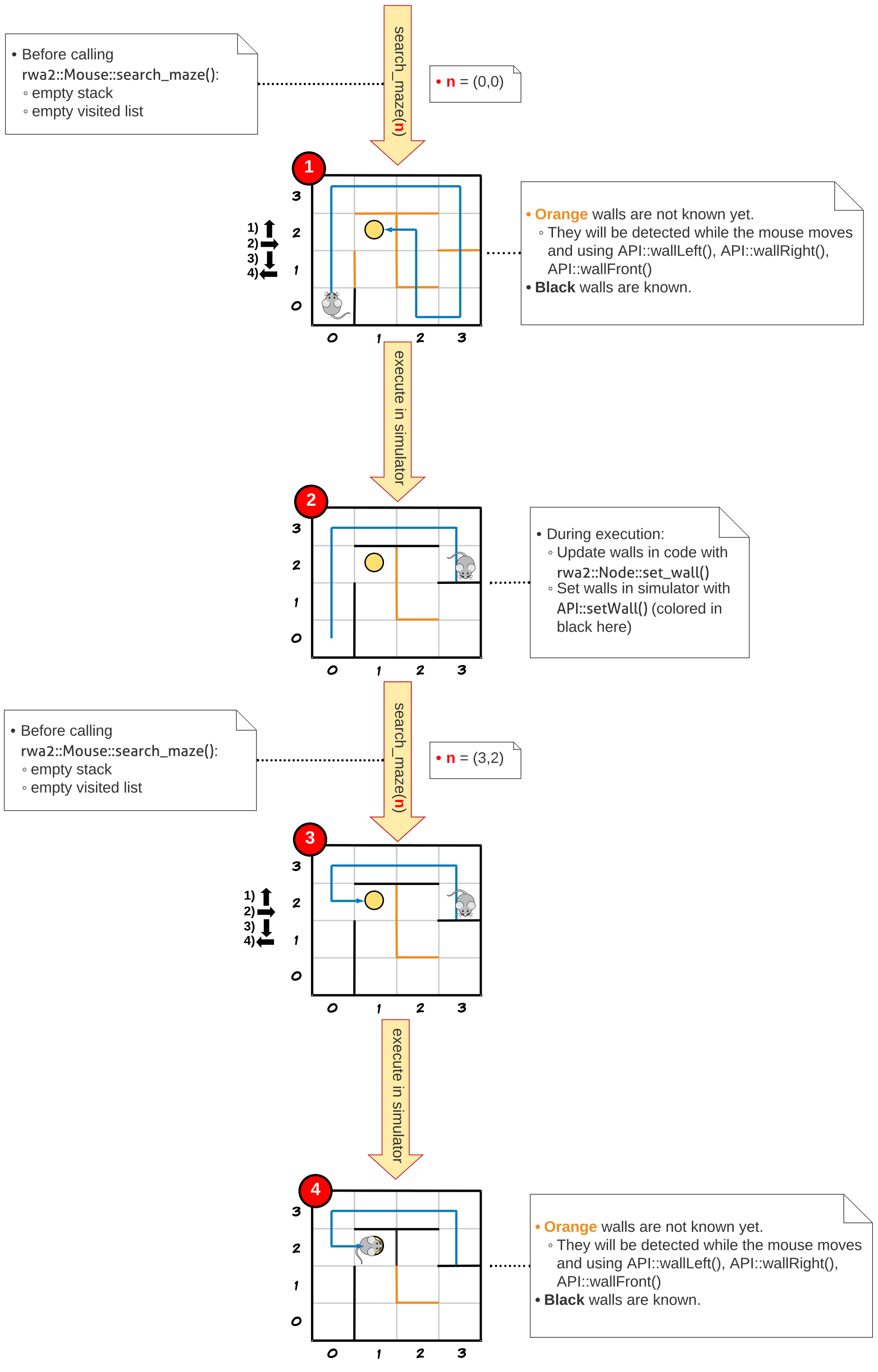

• Before calling
rwa2::Mouse::search_maze():
◦ empty stack
◦ empty visited list
search_maze(n)
• n = (0,0)
1
1)
2)
3)
4)
• Orange walls are not known yet.
◦ They will be detected while the mouse moves and using API::wallLeft(), API::wallRight(), API::wallFront()
• Black walls are known.
execute in simulator
2
• During execution:
◦ Update walls in code with rwa2::Node::set_wall()
◦ Set walls in simulator with API::setWall() (colored in black here)
• Before calling
rwa2::Mouse::search_maze():
◦ empty stack
◦ empty visited list
search_maze(n)
• n = (3,2)
3
execute in simulator
4
• Orange walls are not known yet.
◦ They will be detected while the mouse moves and using API::wallLeft(), API::wallRight(), API::wallFront()
• Black walls are known.
0
1
2
3

Figure 6: Path search and execution.

Algorithm 2 provides the different steps that are required when running the program.

Algorithm 2: Program flow for RWA2.

```
1  initialize the maze and boundary walls;
2  initialize the position of the mouse in the maze;
3  initialize the direction of the mouse in the maze;
4  while true do
5      clear the color for all tiles;
6      set color and text for goal cell;
7      get the mouse's current position;
8      update maze walls in the program (mark the walls found);
9      update maze walls in the simulator (display the walls);
10     generate a path from current position to goal;
11     if no path found then
12         break;                                  /* maze unsolvable */
13     end
14     color path tiles in the simulator;
15     move the robot along the path;
16 end
```

Explanation for some of the statements in Algorithm 2 is given below.

- Line 1: Initializing the maze consists of programmatically creating a 16x16 maze and initializing the boundary walls of the maze. These two steps are performed in the constructor of the class `rwa2::Mouse` (see **mouse.h**).
- Line 2: The position is always initialized to (0,0). This is performed in the constructor of the class `rwa2::Mouse` (see **mouse.h**).
- Line 3: The direction of the mouse is always North. The initialization of the direction is already performed in the constructor of the class `rwa2::Mouse` (see **mouse.h**).
- Line 5: Use `API::clearAllColor()` to clear the color of all tiles.
- Line 6: The mouse has to reach one of the four center nodes $G_1(7,7)$, $G_2(7,8)$, $G_3(8,7)$, or $G_4(8,8)$. The student has to programmatically choose (hard code the goal) one of these nodes for the mouse to reach. Once the goal is chosen, set a color and a text in the simulator for the goal node. The color can be set using `API::setColor()` and the text can be using `API::setText()`. The student is free to choose the color. As for the text, it should be the goal name. For instance, if the student chooses G_1 as the goal then `API::setText(7,7,"G1")` should be used.
- Line 7: To keep track of the position of the mouse in the maze, the mouse's direction and displacement must be used. For instance, if the mouse is at (0,0) and facing North, then the execution of `Mouse::move_forward()` moves the mouse

to node $(0,1)$. If the mouse is at $(0,0)$ and facing East, then the execution of `Mouse::move_forward()` moves the mouse to node $(1,0)$.
- Line 9: Use `API::setWall()` to visually display a wall in the simulator.
- Line 10: This is done by executing `Mouse::search_maze()`.
- Line 14: Use `API::setColor()` the tiles from the path generated by `Mouse::search_maze()`.
- Line 15: Use `Mouse::move_forward()`, `Mouse::turn_left()`, and `Mouse::turn_right()`. These three methods need to be implemented in the `Mouse` class. To keep it simple, make:
  - `Mouse::move_forward()` call `API::moveForward()`
  - `Mouse::turn_left()` call `API::TurnLeft()`
  - `Mouse::turn_right()` call `API::TurnRight()`

7 Implementation

This assignment can be done with the implementation of at least two classes: `Mouse` and `Node`. The class diagrams is shown in Figure 7. A minimal C++ example for the project is provided on Canvas as `rwa2_minimal.zip`. The Doxygen documentation is inside the **doc** folder in the zipped project (open `index.html` in a browser). To complete this assignment, students will need to:

- Augment the provided classes with new functionalities.
- Modify existing functionalities in the provided classes if needed.
- Create new classes if needed.

A summary of the overall tasks to do for the assignment:

1. Implement the DFS algorithm using a representation of the maze in your program.
2. Generate a path from the current position to the goal using your representation of the maze in your program.
3. Move the mouse in the simulator by following the path generated by DFS. While the mouse is moving, in each node the mouse is passing through, use `API::wallFront`, `API::wallRight`, `API::wallLeft` to detect walls and update your maze in your program.
4. If the mouse finds a wall which prevents it from going any further, stop the robot and recompute the path using DFS. Now DFS is computed using the walls found by the robot in the previous run.
5. Repeat from step 2. until the mouse reaches the goal.

In the provided minimal example, a maze (`rwa2::Mouse::m_maze`) is defined as a 2D array of size 16x16. The type of items in a maze is `rwa2::Node`, which is a custom class, defined in `node.h`.

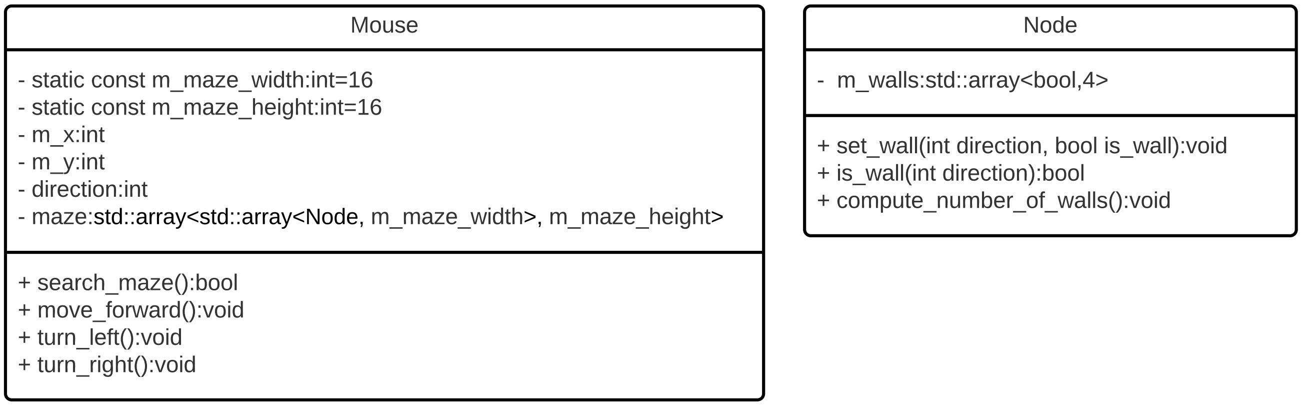

Mouse
- static const m_maze_width:int=16
- static const m_maze_height:int=16
- m_x:int
- m_y:int
- direction:int
- maze:std::array<std::array<Node, m_maze_width>, m_maze_height>
+ search_maze():bool
+ move_forward():void
+ turn_left():void
+ turn_right():void
Node
- m_walls:std::array<bool,4>
+ set_wall(int direction, bool is_wall):void
+ is_wall(int direction):bool
+ compute_number_of_walls():void

Figure 7: RWA2 classes.

8 Project Structure

1. In VSC create a C++ project **RWA2Group_<number>**. For instance group 1 from section 1 will create a project named **RWA2Group_1_1**.
2. Replace the **src** and **include** folders with the ones downloaded from Canvas.
3. Also paste the **doc** folder and the Doxyfile in the root folder of your project. You should have the following in the root folder of your project:
   - **src**
   - **include**
   - **doc**
   - `Doxyfile`
4. To test your project with the simulator:
   - Run the simulator by following the steps in Section 4.3.
   - Configure "New Mouse Algorithm" window by following the steps described in Section 4.4. Configure the fields as follows:
     - **Name**: I suggest you name it as `RWA2_Group_<number>` (e.g., `RWA2_Group_1_1`).
     - **Directory**: Navigate to root folder of your VSC project.
     - **Build Command**: Using the files downloaded from Canvas, the build command is:

```
g++ src/main.cpp src/micromouse.cpp src/node.cpp src/api.cpp
```

    * If new .cpp files are added to the project, this command will need to be updated to include the new files.
  - **Run Command**: `./a.out`
- Test your algorithm with different mazes by selecting different files from the simulator.

9 Submission

1. Complete the assignment.
2. Document your code with Doxygen. Remember to only document the .h files. Comments should be added in .cpp files to explain what is happening in your code.
3. Update the Doxyfile provided if needed. In a terminal, `cd` into the folder where Doxyfile is located and execute `doxywizard`. This should open the GUI with Doxyfile preloaded. Once you are done modifying this file, save it. Generate the HTML documentation for yourself to make sure everything looks good. There is no need to submit the HTML documentation.
4. Include an instructions.txt file in your project to inform TAs what to enter in the field **Build Command**.
5. Compress your project (zip, rar, etc).
6. Upload it on Canvas.

10 Grading Rubric

- Total points: 50 pts.
- 30 pts if your program works as expected.
  - Implementation and execution of DFS.
  - Mouse follows the generated path.
  - Walls updated in the simulator.
  - Path colored in the simulator.
  - Color and text set for the goal.
- 10 points for comments and Doxygen documentation.
- 10 points for following C++ guidelines and best practices seen in class.
- Penalty is applied for late submission, even if the submission is one minute late. A 10% per day late submission policy is applied.
- Important: Our TAs have been vetted by UMD and have followed all appropriate guidelines. It is not the TAs' responsibility if your program does not work as expected because you forgot to check something. You will be graded based on

your submission and there is no second chance. Unless the TAs did not follow the grading rubric, there is no reason to ask for regrading your assignment. Make sure your program works before you submit your code.