

Introductory Robot Programming

ENPM809Y

Lecture 10 – Final Project Discussion

Zeid Kootbally

zeidk@umd.edu

Fall 2020



Overview I

01 Final Project

Micromouse

Tree

Depth-First Search (DFS)

Simulator

Final Project

- Objectives

- Write a program that generates a path from the current position (S) of a robot in a maze to the center of the maze (G).
- Once a path is generated, the robot is tasked to follow the path.
- The robot that navigates the maze will always be a robot from a derived C++ class.
- The micromouse simulator will be used as the visual interface.

- Tools and Resources

- The micromouse simulator.
- A collection of maze files.
- Your maze solver algorithm written in C++.
- Google C++ Style Guide.
- Best practices.

Final Project | Micromouse

- Micromouse is an event where small robot mice solve a maze.
- The maze is made up of a 16×16 grid of cells, each 180 mm square with walls 50 mm high.
- The mice are completely autonomous robots that must find their way from a predetermined starting position to the central area of the maze unaided.
- The mouse needs to keep track of where it is, discover walls as it explores, map out the maze and detect when it has reached the goal.
- Example.

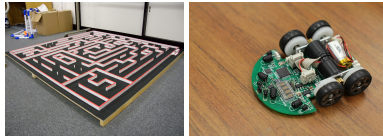


Figure: <https://en.wikipedia.org/wiki/Micromouse>

Final Project

- 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.

Final Project | Tree

- Tree: A hierarchical data structure which stores the information in the form of hierarchy. This is different from linear data structures such as arrays, vectors, linked lists, etc.
 - A tree contains nodes and connections which should not form a cycle.

Final Project | Tree | Common Terminologies

- Node: A structure which may contain a value or condition, or represent a separate data structure.
- Root: The top node in a tree, the prime ancestor.
- Child: A node directly connected to another node when moving away from the root, an immediate descendant.
- Parent: The converse notion of a child, an immediate ancestor.
- Leaf: A node with no children.
- Edge: The connection between one node and another.

Final Project | Tree | Tree Traversal

- A form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once.¹
 - Such traversals are classified by the order in which the nodes are visited.
 - For the final project, the order to visit nodes is: \downarrow , \rightarrow , \uparrow , \leftarrow .
- Tree traversal algorithms can be classified mainly in the following two categories:
 - Depth-First Search (DFS): Start with the root node and first visit all nodes of one branch as deep as possible before visiting all other branches in a similar fashion (post-order traversal). **DFS is used for the final project.**
 - Breadth-First Search (BFS): Start from the root node and visits all nodes of current depth before moving to the next depth in the tree.

¹https://en.wikipedia.org/wiki/Tree_traversal

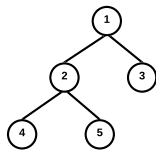
Final Project | Depth-First Search (DFS)²

- Algorithm for traversing or searching tree or graph data structures.
- The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.
- Backtracking: When you are moving forward and there are no more nodes along the current path, you 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.
- DFS does not always generate an optimal solution and will usually not generate the shortest path.

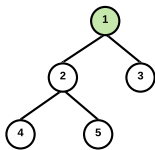
²https://en.wikipedia.org/wiki/Depth-first_search

Final Project | Depth-First Search (DFS)

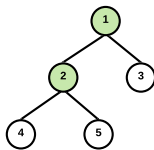
- Goal: Find a path from 1 to 3.
- Instruction: Explore the left branch first and then the right branch.



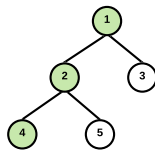
1



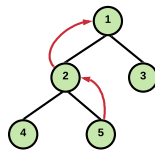
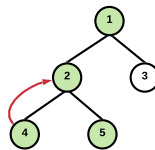
1 → 2



1 → 2 → 4



1 → 2 → 4 → 5



1 → 2 → 4 → 5 → 3

Final Project | Depth-First Search (DFS) | Pseudo Code

- DFS can be implemented using stacks with the last-in-first-out (LIFO) approach.
 1. Pick a starting node and push it into a stack.
 2. Pop a node from top of stack to select the next node to visit (using the search order).
 - If a next node is found then push it into the top of the stack.
 - If a next node is not found, then pop the current node from the stack.
 3. Repeat this process until the stack is empty or the goal is found (whichever comes first).
- Fortunately C++ has a stack data structure that we can use.

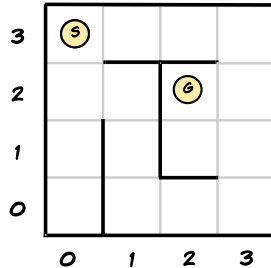
Final Project | Depth-First Search (DFS) | Pseudo Code

- Ensure that the nodes that are visited are marked.
- This will prevent you from visiting the same node more than once.
- If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.
- You can use a boolean array to keep track of visited nodes:

```
int row_number{4};
int col_number{4};
//--all elements of the array set to false
bool visited_node[row_number][col_number]{false};
//--when you visit a node
visited_node[0][2] = true;

if (visited_node[0][2]) {
    std::cout << "node [" << x << ", " << y << "] is visited" << std::endl;
}
```

Final Project | Depth-First Search (DFS) | Example

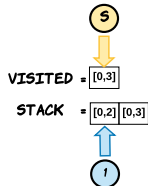
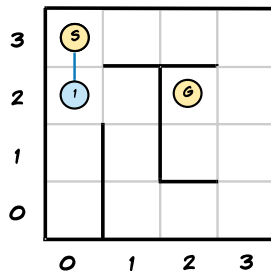


VISITED =

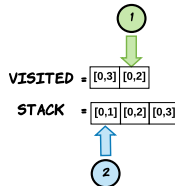
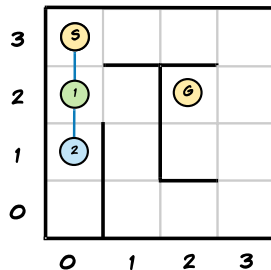
STACK = [0,3]



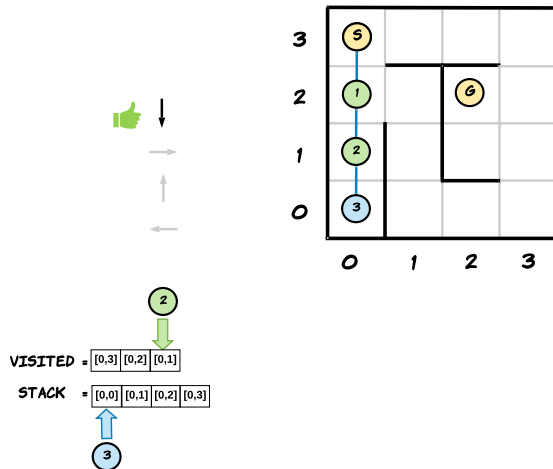
Final Project | Depth-First Search (DFS) | Example



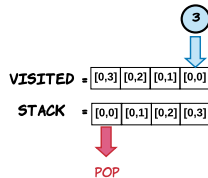
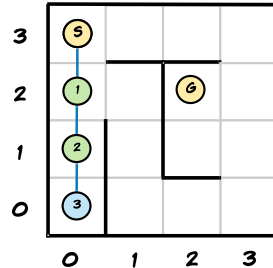
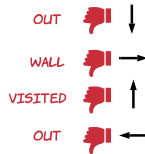
Final Project | Depth-First Search (DFS) | Example



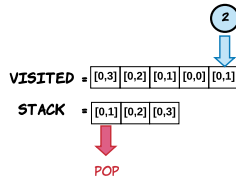
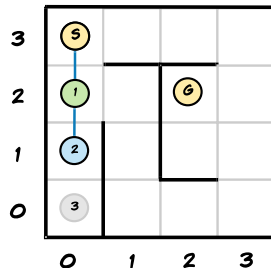
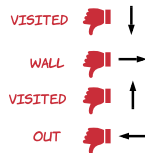
Final Project | Depth-First Search (DFS) | Example



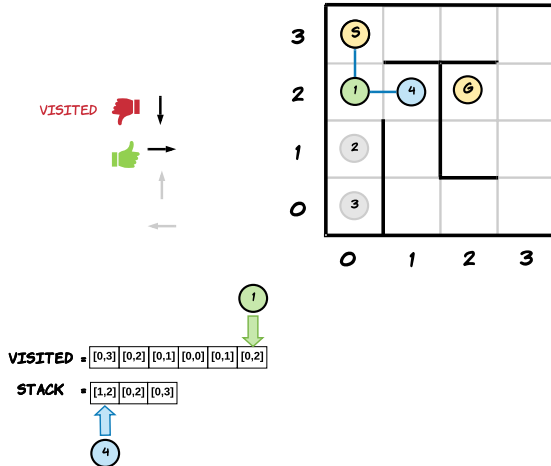
Final Project | Depth-First Search (DFS) | Example



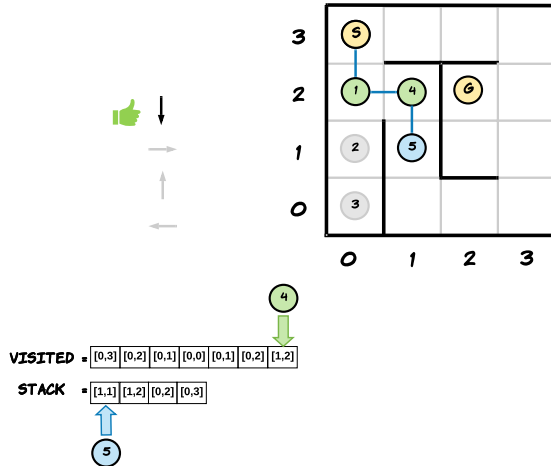
Final Project | Depth-First Search (DFS) | Example



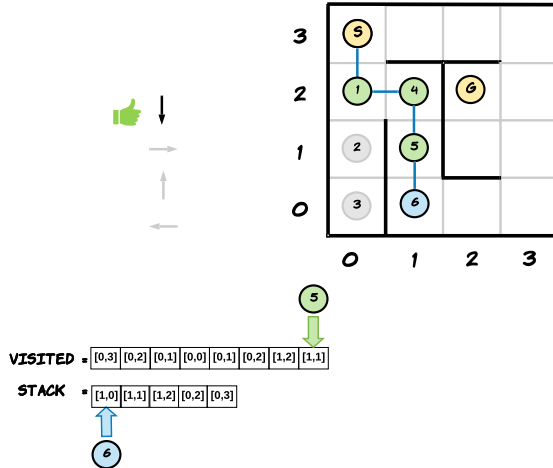
Final Project | Depth-First Search (DFS) | Example



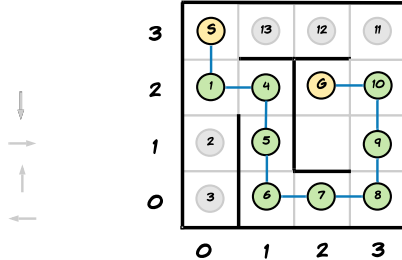
Final Project | Depth-First Search (DFS) | Example



Final Project | Depth-First Search (DFS) | Example




Final Project | Depth-First Search (DFS) | Example

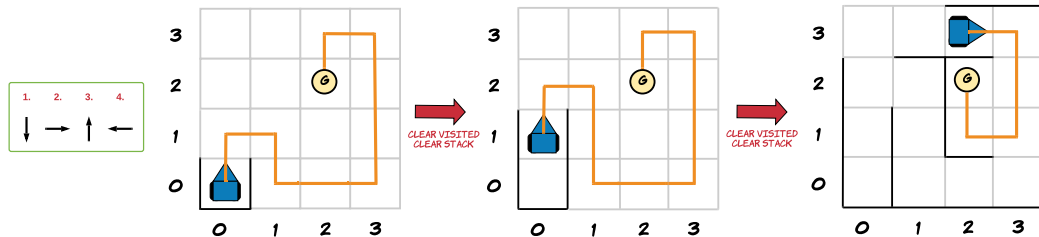


VISITED = [0,3] [0,2] [0,1] [0,0] [1,2] [1,1] [1,0] [2,0] [3,0] [3,1] [3,2] [3,3] [2,3] [1,3]

STACK = [2,2] [3,2] [3,1] [3,0] [2,0] [1,0] [1,1] [1,2] [0,2] [0,3]

SOLUTION = [0,3] [0,2] [1,2] [1,1] [1,0] [2,0] [3,0] [3,1] [3,2] [2,2]  REVERSE

Final Project | Depth-First Search (DFS) | Example

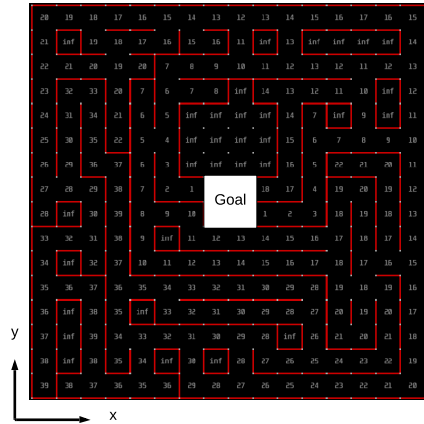


1. Compute a path from S to G using the DFS algorithm.
2. Move the robot and discover the walls.
3. Recompute a path each time the original path is blocked by a wall.
4. The current position becomes the start position (S) and the goal position (G) stays the same.

Final Project | Simulator

- In the final project:
 - All mazes have the same dimensions 16×16 cells.
 - All mazes have walls around their perimeter.
 - The robot has no knowledge of walls (even the ones around the perimeter) beforehand. Walls are discovered only when the robot drives through cells that have walls.
 - The coordinate system for all the mazes is displayed in the next slide.
 - $x \in [0, 15]$ and $y \in [0, 15]$
 - Any of the 4 cells that are in the center of the maze constitutes the goal the robot needs to reach (see next slide). The coordinates of the goal cells are consistent among mazes: $(7, 7)$, $(7, 8)$, $(8, 7)$, $(8, 8)$
 - When the simulation starts, the robot will always start at position $(0, 0)$ and will always face north.

Final Project | Simulator



Final Project | Simulator

- After generating a path using the DFS algorithm you need to move the robot along this path.
- To do so we will use an open source project, the micromouse simulator, a small C++ QT simulator that allows you to visualize path planning algorithms.
- You need to communicate with the simulator to send commands and read status. More information on the communication with the simulator is provided in an external pdf file [809Y_FinalProject_Fall2020.pdf](#).

Final Project | Simulator

- A list of non-exhaustive steps for this project are:
 1. Start the micromouse simulator and load a maze.
 2. Set up the simulator to find your C++ files.
 3. Run the simulation:
 - 3.1 Load maze in the simulator.
 - 3.2 Create a 16×16 maze in your program.
 - 3.3 Get robot location in your program.
 - 3.4 Get robot direction in your program.
 - 3.5 Compute path (DFS) from current robot location to goal location.
 - 3.6 Move the robot based on the path generated.
 - Update maze with new discovered walls.
 - Repeat steps 3.3 – 3.6 until goal is found (all the mazes used have at least one solution path).

Final Project | Simulator | Installation

- Create a directory on your desktop:

```
mkdir ~/Desktop/FinalProject
```

```
cd ~/Desktop/FinalProject
```

- Clone the micromouse simulator in the directory you just created:

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

- Clone a set of maze files:

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

- Clone a C++ program which simply turns the robot left and right:

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

Final Project | Simulator | Installation

- Compile the simulator:

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

```
cd mms/src
```

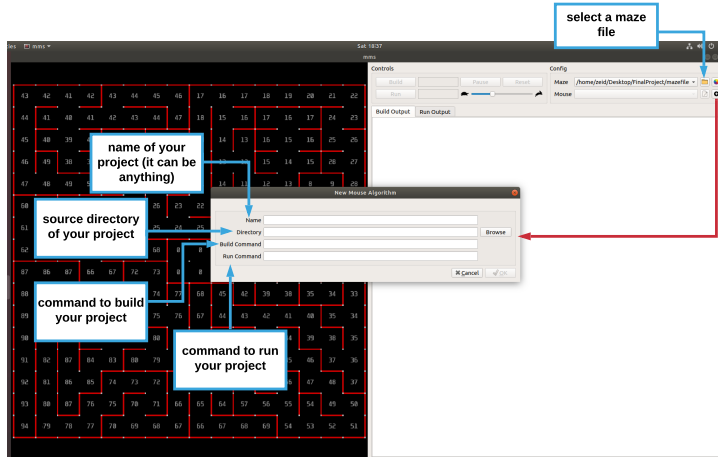
```
qmake && make
```

- Run the simulator:

```
cd ~/Desktop/FinalProject/mms
```

```
bin/mms
```

Final Project | Simulator | Installation



Final Project | Simulator | Interfacing with the Simulator

- The interaction between your code and the simulator is done through an API class.
- This class and method definitions can be found in the mms-cpp directory.

```
int mazeWidth();/--return the width of the maze
int mazeHeight();/--return the height of the maze
bool wallFront();/--true if there is a wall in front of the robot, else false
bool wallRight();/--true if there is a wall to the right of the robot, else false
bool wallLeft();/--true if there is a wall to the left of the robot, else false
void moveForward();/--move the robot forward by one cell
void turnRight();/--turn the robot 90 degrees to the right
void turnLeft();/--turn the robot 90 degrees to the left
void setWall(int x, int y, char direction);/--display a wall in maze
void clearWall(int x, int y, char direction);/--clear a wall from maze
void setColor(int x, int y, char color);/--set the color of a cell
void clearColor(int x, int y);/--clear the color of a cell
void clearAllColor();/--clear the color of all cells
void setText(int x, int y, const std::string& text);/--set the text of a cell
void clearText(int x, int y);/--clear the text of a cell
void clearAllText();/--clear the text of all cells
bool wasReset();/--true if the reset button was pressed, else false
void ackReset();/--reset simulation
```

Final Project | Simulator | Interfacing with the Simulator

- An example of use of some of these methods can be found in **Main.cpp** in the *mms-cpp* directory.
- You need to include the API class definition and implementation in your project.
- Open **Main.cpp** to take a look.

Next Class | 11/12

- **Lecture 11** – The Robot Operating System (ROS) – Part I.
 - You will need a Linux system for this one!
- Stay safe!