



COSC2658 Group Project

Lecturer: Tri Dang Tran

Group 21

Duong Phu Dong	S3836528
Ho Quoc Thai	S3927025
Do Hoang Quan	S3800978
Phan Trong Nguyen	S3927189

Table of Contents

1. Overview	2
A. Overview	2
B. High-Level Design	3
2. Data Structure & Algorithm	4
A. Data structure	4
B. Algorithms overview	4
C. Classes declaration	4
D. Algorithms' working description	5
3. Complexity Analysis	7
A. Pseudocode of Display the robot's memory:	7
4. Evaluation	8

1. Overview

A. Overview

Since the final maze will use an unknown design with the goal being either at the outer walls or inside open spaces, the safest way to ensure the robot always reaches the goal is to make it trace every available space while fulfilling the requirements:

- Don't walk into the same wall twice
- Don't backtrack unnecessarily

To achieve this, we gave the robot:

- **A “memory” using a 2D array to record where it has visited and where any wall is**

This 2D array structure stores every visited position using their X and Y coordinates relating to the robot's starting position at $X=0$ and $Y=0$ and marks if that position is either “visited,” “wall,” or “dead-end”. The robot continuously checks its “memory” for valid moves (not walking into walls, visiting positions) before calling the Maze's “go()” function to ensure efficiency. Using this “memory”, the robot will prioritize new and unvisited positions, never walk into the same wall twice (even if it travels to the other side of the wall), and only backtrack when necessary to get out of a dead-end or a fully traced open space.

- **A stack to store the directions of all moves made.**

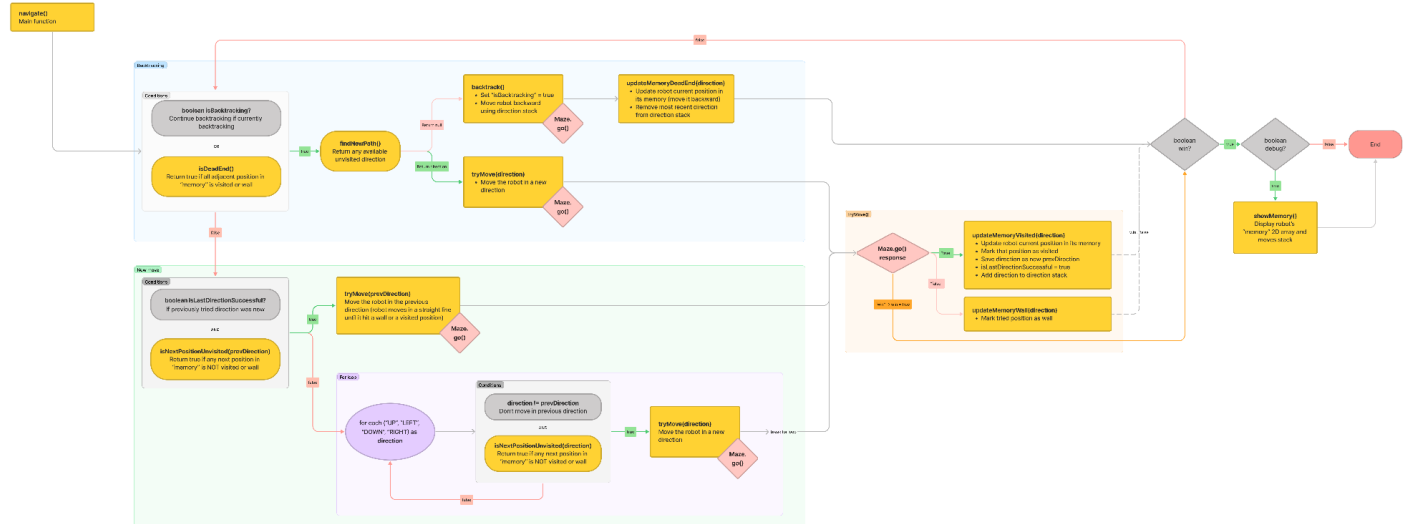
This stack is used during backtracking to ensure the robot moves back in the reverse position while checking at every possible step to find new, unvisited positions. The robot will “pop” the most recent direction of the stack for every backtrack step. This way, the robot will always make it out of any dead-end and open space until it reaches its original position.

This algorithm ensures the robot never makes unnecessary moves and always finds the goal no matter where it is.

B. High-Level Design

© 2000 by Pearson Education, Inc.

Group 21 Robot High Level Diagram



[View this high-level diagram in detail](#)

Figure 1: High level diagram

Classes are marked in yellow boxes.

2. Data Structure & Algorithm

A. Data structure

We use two primary data structures to record the robot's moves:

1. **2D array**
2. **Stack**

B. Algorithms overview

We have decided to implement Depth-First Search to trace the optimal path to exit the maze. Our convention for the maze's characteristics is as follows:

- Unvisited cells are marked as “.”
- A visited cell is marked as “v”.
- Dead cells (cells that are visited but do not provide optimal path) are marked as “d”.

Also, we decided that the priority of our moves is Up (U), Left (L), Down (D), and Right (R). The robot will follow this order as long as the available path is valid.

In this project, we use the Stack data type to utilize its Last In First Out characteristic to store the path. In short, if the cell is a dead cell, it will pop that cell from the stack and then return to the cell that has possible moves. The final stack stores the path of our maze (a detailed explanation will be provided in the following sessions).

The algorithm is quite simple: From the start, we trace the next possible cell until we can find the exit (noted as “x”). The movement to the next cell depends on the order of the moves that we have specified previously. At first, the cell is marked “v” so that we can avoid the visited cell from being duplicated. That cell is also stored in the stack to form the maze's path. If the “v” cell has no other possible move, whether in the middle of all other visited cells or hits the wall and cannot turn left or right, that cell will be marked as “d”. Then, we will trace back to the last cell marked as “v” in our stack to try another path from it. The final path contains all the “v” cells from the start to “x.”

C. Classes declaration

We declare three classes that help us handle the maze tracing:

- “Stack” class: store the path of our maze. It includes methods a stack may have:
 - Push: Add a new element.
 - Pop: Delete the latest element from the stack.

- Peek: check for the most recent element.
 - Size: Calculate our stack size to provide moves taken to exit.
 - RetraceAllmoves: to export all elements of the stack.
- “Maze” class: This class contains the characteristics of the maze
 - “map” array: this will store the map's array, its size of rows and columns, and cell characteristics.
 - “go” method: check if there exists any new move. If yes, the step counter will increase.
- “Robot” class: Stores the position of the robot and methods to handle the movement of the robot:
 - Final variables U, D, L, R, true, false, v, d, w: These variables will determine the characteristics of the cell and the current direction (For example, if a cell A moves UP to cell B, the current direction is U).
 - “memory” 2-dimension array: to store the visited cell of the robot.
 - String “prevDirection,” Boolean “isLastDirectionSuccessful” and “isBacktracking”, Stack “allMoves”, Boolean “win”: features of the robot to trace the path, store the cells that build the optimal path, and the winning condition of the maze.
 - “Robot” constructor: to initialize the robot's characteristics.
 - “navigate” method: with many sub-methods (listed in table 1), this method handles the movement of our robot, checks the possible cell, and traces

D. Algorithms’ working description

At first, we initialize the maze’s information to the “Maze” class. This Maze requires a 2D array to store, so we use a String-type “map” to store it (since the string itself is an array already). Each index of the map will store one row, and the string indices represent the column. Also, the starting position is declared in this class, and the step counter. After the information is loaded, we will pass it to the constructor **Robot** to initialize the robot’s attribute for each Maze case. The robot’s initial position and

information about the maze are stored in a Maze-type **maze**, declared in the **Robot** class.

At first, the method **isNextPositionUnvisited** (lines 101-104) will check if the cell has been visited (or iterated) using the method **getNextPositionFromMemory** (lines 204-220). **getNextPositionFromMemory** tells the robot the direction based on the priority list of directions. If the cell is not visited, it will proceed that cell into **tryMove** (lines 109-124) method. This method will check whether the cell is a wall or an empty cell. Two methods, **updateMemoryWall** (lines 146-155) & **updateMemoryVisited** (lines 129-141), will check the cell. If a wall is hit, **updateMemoryWall** will mark the cell as "w," or it will be pushed into the stack **allMoves** (line 30) declared in class **Robot**.

All of the possible will be traced until it reaches a dead-end cell. This cell will be surrounded by visited cells or walls. The boolean-type method **isDeadEnd** (lines 159-164) will tell if all of the adjacent cells of the current one are impossible to proceed with. Then, this method will return the value, which the Robot can extract from **getNextPositionFromMemory**. ____ Then, the method **backtrack** (lines 169-174) will move the robot backward from the dead-end cell. To do this, we declare **oppositeOf** (lines 223-239) to check for the directions that lead to the current cell. The data for this method can be extracted using **allMoves.peek()**. The robot will move backward using that direction, and the cell will be marked as "d." The back-tracking process happens until a cell has possible moves (no dead-end found). Once again, the process works continuously until the robot reaches "X" - it will be notified as "win." All these steps occur in a do-while loop **Navigate** (lines 55-99).

To print out the maze information, directions, steps, dead-ends, and walls, we declare a method **showMemory** (lines 244-273). For the steps stored in **allMoves**, we declare a method **retraceAllMoves** (lines 334-338) by printing the current head in **allMoves** of each iteration through the loop. To take the number of steps to exit the maze, we have a **go** (line 377-385) method to increase the step counter.

3. Complexity Analysis

A. Pseudocode of Display the robot's memory:

```
function showMemory()
{
    Print (Map);
    Print (\t);
    for i starts at 0 , i -> memory length, i increases:
    {
        Print (i + "\t")
    }
    Print ();
    for i starts at 0 , i -> memory length, i increases:
    {
        Print (i + "\t")
        for(array char[] chars : memory ) {
            if ( chars at position i != '\0' {
                Print (chars at position i + "\t")
            } else
            {
                Print (i + "\t")
            }
        }
    }
    Print ();
}
```

1. "Memory" array search - **O(1)**: because we always know the X, Y of the robot, and every time we search, it **ONLY** searches for 4 points around, so we always know what the index to search is, so it's always a constant time.
2. "Memory" array insert - **O(1)**: the same reason as above (it also only inserts around the X and Y of the robot that we always know)
3. "Memory" array deletion - N/A
4. The robot's number of steps: the maximum will be twice the number of steps (not double the map size). All points can go; worse, it will **go twice**, average **only once**.
5. "allMoves" stack insert, pop - **O(1)**: insert and pop only to the nearest place.
6. "allMoves" delete - N/A.

4. Evaluation

For the project to be successful, it needs to fulfill all the requirements which were given at the beginning:

- Do not walk into a wall twice
- No unnecessary backtracking
- Be able to keep track of visited positions
- Must stop whenever it reaches the final position in the maze

The most appropriate way to test the correctness and also complexity of the program is by testing the program through different mazes. By that, we could prove the big-O that we have calculated above.

To test the correctness of the program, we would use a dataset that we could quickly check, allowing us to check if our program can perform the algorithm correctly. For the program's complexity, we would check with several datasets of different sizes, by which we could further prove if the theoretical calculation is correct.

To further prove the correctness and complexity of the program, we will test the program with a sample maze of 10x10:

The robot will be placed at the position with row =8 and column=5 as in the picture. As in our program, the robot will go in a specific direction that we have specified, which in our case is in order of: "UP, DOWN, LEFT, RIGHT". And if the robot hits the wall, it will change direction until it has a new direction. With every robot move, we also store that position into an array, which the Robot can use for backtracking if it collides with a dead end or is stuck in a loop.

```
map[0] = ". . . . . . . . . .";
map[1] = ". . . x | . . . .";
map[2] = ". . . . . . . . . .";
map[3] = ". . . . . . . . . .";
map[4] = ". . . . . . . . . .";
map[5] = ". . . . . . . . . .";
map[6] = ". . . . . . . . . .";
map[7] = ". . . . . . . . . .";
map[8] = ". . . . . . . . . .";
map[9] = ". . . . . . . . . .";
```

Figure 2: 10x10 map

We can ensure that the Robot will check every possible space, and the stored locations will help prevent the robot from moving through one position twice. The robot will stop as soon as it gets to the “win” mark of the maze.

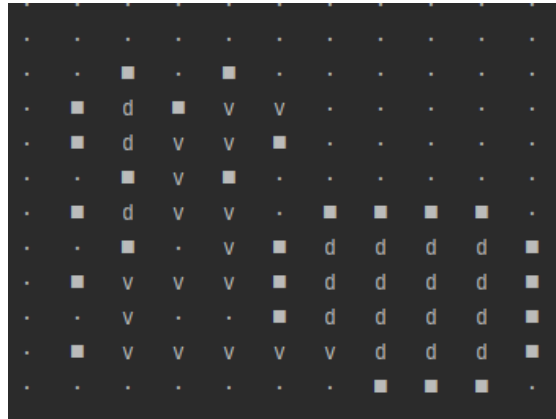


Figure 3: Robot moving map in 10x10 map

Steps to reach the Exit gate 79

Figure 4: Robot steps counter for 10x10 map

For the correctness of the algorithm, we could inspect that the robot at its initial start point has gone in the specified direction. If it meets a dead end or has no other space to visit, it will backtrack out and mark those areas as “d” and then continue using different directions with the same step until it reaches the final destination.

```
Retrace to origin  
RIGHT <- UP <- RIGHT <- UP <- UP <- LEFT <- UP <- UP <- RIGHT <- RIGHT <- UP <- UP <- LEFT <- LEFT <- LEFT <- LEFT <- null
```

Figure 5: Robot's retrace

The program's complexity has two aspects: space and time. Using the theory calculation conducted in the 3rd section of the report, we can compare it with the testing result, which is approximately equal to the theoretical analysis.

```
map[0]=".....";
map[1]=".....x";
map[2]=".....";
map[3]=".....";
map[4]=".....";
map[5]=".....";
map[6]=".....";
map[7]=".....";
map[8]=".....";
map[9]=".....";
map[10]=".....";
map[11]=".....";
map[12]=".....";
map[13]=".....";
map[14]=".....";
map[15]=".....";
map[16]=".....";
map[17]=".....";
map[18]=".....";
map[19]=".....";
map[20]=".....";
map[21]=".....";
map[22]=".....";
map[23]=".....";
map[24]=".....";
map[25]=".....";
map[26]=".....";
map[27]=".....";
map[28]=".....";
map[29]=".....";
```

Figure 6: 30x30 map

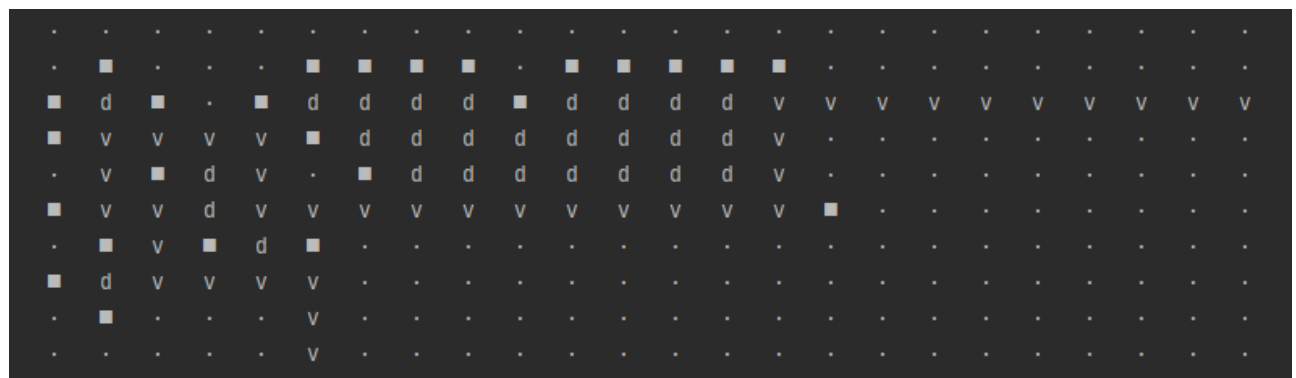
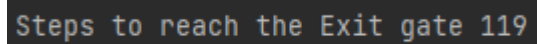


Figure 7: Robot's moving map in 30x30 map



Steps to reach the Exit gate 119

Figure 8: Robot's steps counter for 30x30 map

Furthermore, we continue to test run our program at a larger scale map of 30x30 for the complexity evaluation as in the figures above. We could prove that our complexity has been calculated correctly by evaluating the steps needed with the theoretical calculation.