COSC2658: Group Project

Instructor & Tutor: Dr. Tri Dang Tran

Team number: *Team 19*

Students:

Dao Kha Tuan (s3877347)

Nguyen Ngoc Minh (s3907086)

Bui Quang An (s3877482)

Nguyen Luu Quoc Bao (s3877698)

Date: *12/12/2022*

1) Overview

The Maze-solving Robot is a problem that asks developers to implement algorithms for the Robot to find the most optimal way to reach the end of the maze. In this problem, the Robot will be placed in a dark maze which makes the Robot cannot identify its position in the maze, which has a maximum size of 1000x1000, and it can move in 4 directions, including "UP", "DOWN", "LEFT", and "RIGHT". In order to solve this problem, our group decided to use the recursive backtracking algorithm, and we implemented it using Java programming language. Maze-solving Robot has 2 main classes: "Maze" and "Robot", and 2 utility classes which are "FileScanner" and "ArrayStack". While the "Maze" class has 2 methods, the "Robot" class has 6 methods, and each of them with the utility classes plays an essential role in implementing the algorithm to the Robot.

2) Data Structure and Algorithm Description

a) Data Structure

Our team used a data structure called "ArrayStack" to store the previous steps that the Robot had made so we could perform a backtracking operation after it encountered a dead end in the Maze. It has a constructor with an argument called "capacity" to set the maximum size it can store. This structure has a "size()" function to get the number of objects it contains, an "isEmpty()" function to check whether it is empty or not, a "push()" function to add a new object to itself, a "pop()" function to remove the last added object, and a "peek()" function to get the last object that was added without removing it. This data structure follows the "Last In – First Out" order.

b) Algorithm

The algorithm we decided to apply to solve this problem is a maze-solving algorithm called "Recursive Backtracking." The basic definition of this algorithm is to follow an initial direction until a wall or obstacle is met. Also, it marks all visited cells while traversing the maze. When a wall or obstacle is encountered, the Robot will step into an unvisited direction. Suppose a dead end is bumped into, and there is no alternative direction. In that case, the Robot will perform a technique called "backtracking," meaning it goes back to the previously visited cell right before and finds all the adjacent cells that have not been visited to explore new paths. It is similar to the "Depth First Search" algorithm but applies the Stack

data structure to store the previous steps instead of recursion to perform this "backtracking" technique.

To apply this algorithm, we have created some essential attributes and methods for the Robot class:

- Constant variables MAX_COL and MAX_ROW to store the maximum number of columns and rows a maze to solve can have, which are 1000 rows and 1000 columns, respectively.
- An array to store all 4 directions that the Robot can follow, which are “UP”, “DOWN”, “LEFT”, “RIGHT.”
- A 2–dimensional array to store all visited cells. This array does not reflect the precise locations of the maze, but it is used to check if the Robot has already visited a cell. As the location of the Robot in the maze is not pre-defined, the maximum number of the maze's rows is 1000, and its columns are 1000, it can be calculated that this 2D array must have at least 2001 rows and 2001 columns, as the Robot may be located at one of the 4 farthest corners of the maze.
- Variables called “currentRow” and “currentCol” store the current row and column where the Robot is located. This coordinate is only the location of the Robot in the 2D array mentioned above, not the actual location in the maze.
- Variable to store the current direction of the Robot called “currentDirection”. The initial value of this variable is “UP”.
- An “ArrayStack” data structure called “steps” stores all the directions the Robot has traveled. It is used to apply the “backtracking” technique. To ensure extra capacities for backing up purposes, the initial space of this data structure is 4 * MAX_COL * MAX_ROW, as each cell in the maze can be visited at most 4 times from 4 directions.
- A function to check if a cell in the maze is visited or not, which is called “isVisited(String direction).” This function will take a “direction” argument, indicating where the cell that needs to be checked is located, and it will return true if the cell is already visited. If the cell has not been visited yet, it will be marked as visited and return false.
- “updateLocation()” is a function that updates the “currentRow” and “currentCol” of the Robot after it visits a cell successfully.

- “checkNextStep(Maze maze)” is a function that checks whether the Robot can take a step to an adjacent cell. This function takes the instance of Maze as an argument. First, it will check if a cell is visited with the “isVisited(String direction)” function mentioned above. If it is not visited, it will call the “go()” function of the Maze instance to check whether it can go to the next cell in the current direction. If it reaches the exit, it will return the result immediately. But if it approaches the next non-exit cell, it will call the “updateLocation()” function to update the current location while adding the current direction to the “steps” stack and returning the result.

  If the current direction only leads to a wall or obstacle, the Robot checks other directions instead and repeats the same process as above. If no direction can lead to an unvisited cell, then the function will return false.
- There is also a function to perform the backtracking process, called “backtrack(Maze maze)”, and it takes a Maze instance as an argument. This function gets the current direction from the “steps” stack to identify the opposite direction while removing the current path instruction from the stack. After the Robot follows the opposite direction to perform the backtracking process, the “updateLocation()” function is called to update the current location of the Robot.
- The “navigate()” function is the **main function** of the Robot class. First, it creates an instance of the Maze and marks the initial current location of the Robot as it visits. Then, it will call the “checkNextStep(Maze maze)” function for the Robot to move to any possible cell it can step on. While the exit point has not been reached, when the Robot meets a dead end, it calls the “backtrack(Maze maze)” function to perform the backtracking process to explore any cells that it has not visited before.

3) Complexity Analysis

Let's assign the width of the maze is **m**, and the height of the maze is **n**.

isVisited method:

```
private boolean isVisited(String direction) {                          Complexity
        int checkedRow = currentRow, checkedCol = currentCol;             {1}
        switch (direction) {                                              {1}
            case "UP" -> checkedRow = currentRow - 1;                     {1}
            case "DOWN" -> checkedRow = currentRow + 1;                   {1}
            case "LEFT" -> checkedCol = currentCol - 1;                   {1}
            case "RIGHT" -> checkedCol = currentCol + 1;                  {1}
        }

        if (visited[checkedRow][checkedCol] == 1) return true;            {1}
        visited[checkedRow][checkedCol] = 1;                              {1}
        return false;                                                     {1}
    }
```

Figure 1: This figure shows the full code and the complexity of the isVisited method.

updateLocation method:

```
private void updateLocation(String direction) {          Complexity
        switch (direction) {                                {1}
            case "UP" -> currentRow--;                      {1}
            case "DOWN" -> currentRow++;                    {1}
            case "LEFT" -> currentCol--;                    {1}
            case "RIGHT" -> currentCol++;                   {1}
        }
    }
```

Figure 2: This figure shows the full code and the complexity of the updateLocation method

In the isVisited and updateLocation methods, there is no loop iteration. Therefore, the total time complexity of 2 of the above methods is at constant time, which is known as **O(1)**.

checkNextStep method:

```
private String checkNextStep(Maze maze) {                          Complexity
        String result;                                                {1}
        if (!isVisited(currentDirection)) {                           {1}
            result = maze.go(currentDirection);                       {1}
            if (result.equals("win")) {                               {1}
                return result;                                        {1}
            }
            if (result.equals("true")) {                              {1}
                updateLocation(currentDirection);                     {1}
                steps.push(currentDirection);                         {1}
                return result;                                        {1}
            }
        }

        for (String direction : directions) {                         {1}
            if (!direction.equals(currentDirection)) {                {1}
                if (isVisited(direction)) continue;                   {1}
                result = maze.go(direction);                          {1}
                if (result.equals("true")) {                          {1}
                    currentDirection = direction;                     {1}
                    updateLocation(currentDirection);                 {1}
                    steps.push(direction);                            {1}
                    return result;                                    {1}
                }

                if (result.equals("win")) return result;              {1}
            }
        }

        return "false";                                               {1}
    }
```

Figure 3: This figure shows the full code and the complexity of the checkNextStep method

In this checkNextStep method, there is a loop iteration of a fixed array called “directions”. This array contains only 4 elements: UP, DOWN, LEFT, and RIGHT. Therefore, when combined with all other 0(1) executions, the total time complexity of the above method is **0(1)**.

backtrack method:

```
private void backtrack(Maze maze) {                    Complexity
    currentDirection = steps.peek();                     {1}
    steps.pop();                                         {1}
    String backDirection = "";                           {1}
    switch (currentDirection) {                          {1}
        case "UP" -> backDirection = "DOWN";             {1}
        case "DOWN" -> backDirection = "UP";             {1}
        case "LEFT" -> backDirection = "RIGHT";          {1}
        case "RIGHT" -> backDirection = "LEFT";          {1}
    }
    maze.go(backDirection);                              {1}
    updateLocation(backDirection);                       {1}
}
```

Figure 4: This figure shows the full code and the complexity of the backtrack method

Within the backtrack method, there is no loop iteration. Therefore, the total time complexity of the above method is **0(1)** (constant time).

navigate method:

```
public void navigate() {                               Complexity
    Maze maze = new Maze();                              {1}
    visited[currentRow][currentCol] = 1;                 {1}
    String result = checkNextStep(maze);                 {1}
    while (!result.equals("win")) {                    {2 * m * n}
        result = checkNextStep(maze);                    {1}
        if (result.equals("false")) {                    {1}
            backtrack(maze);                             {1}
        }
    }
}
```

Figure 5: This figure shows the full code and the complexity of the navigate method

Regarding this navigate method, in the while loop, there are 2 actions in which the robot checks the next step and returns a boolean result. If the result is false, the robot has to go back. The process of going into dead-ends and backtracking returns not “win” results. Therefore, in the worst-case scenario, every cell of the maze has to be looped twice. As a result, the total complexity of the above method is **0(2 * m * n)**

Total Time and Space Complexity:

In the case where the exit point is located at the most difficult place to find, the robot has to step through every cell and backtrack every cell of the maze with the exclusion of the 4-wall boundaries. It can be represented by:

$$\boldsymbol{2(m-2)(n-2)}$$

Also, the exit point can be placed on the maze's wall as well. Then we must exclude the 4 points in the 4 corners where it is impossible to reach. The total cells that can be stepped on the wall are calculated by the circumference of the maze minus 4 corners:

$$2 \times (m + n) - 4$$

Therefore, **in the worst-case scenario**, excluding all the constant time executions, we have the time complexity of:

$$2(m-2)(n-2) + 2(m+n) - 4$$

$$\Rightarrow 2(mn - 2m - 2n + 4) + 2m + 2n - 4$$

$$\Rightarrow 2mn - 4m - 4n + 8 + 2m + 2n - 4$$

$$\Rightarrow 2mn - 2m - 2n + 4$$

By not taking constant numbers into account, the above calculation has shown that **the worst time complexity** of the algorithm will always take **less than 0(2*m*n)**. Moreover, **the worst space complexity** is also less than **0(2*m*n)** as it only takes one time to stack every cell in the maze into a stack structure, and this process excludes the cells of the maze's wall.

4) Evaluation

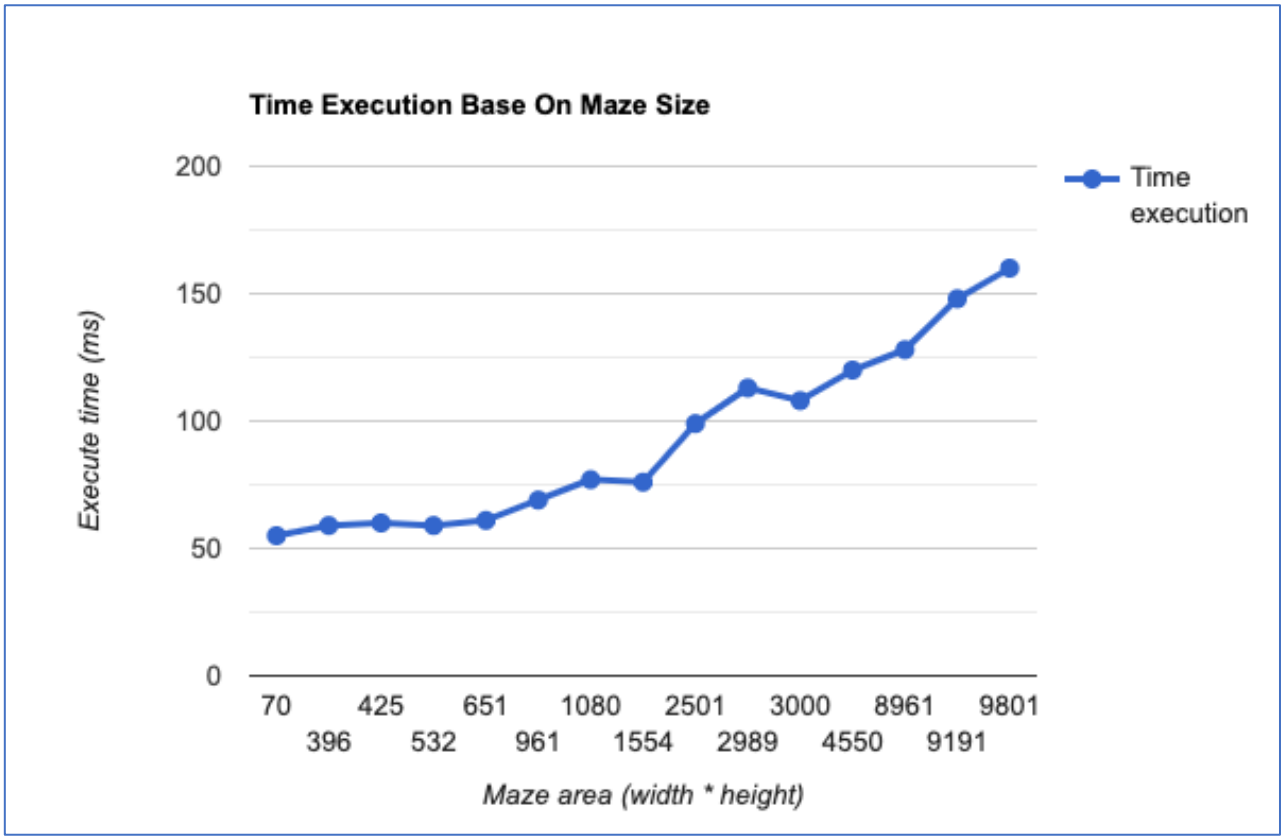
To test the correctness and efficiency of the system, we created 16 test cases to evaluate the results and the steps taken. *Below is the table of results.*

Maze file	Row	Column	Expected	Result	Step to solve	Execution time (ms)
maze.txt	101	91	true	true	5727	148
maze1.txt	21	31	true	true	647	61
maze2.txt	42	37	true	true	1648	99
maze3.txt	30	100	true	true	2759	120
maze4.txt	17	25	true	true	370	60
maze5.txt	7	10	true	true	50	55
maze6.txt	31	31	true	true	479	69
maze7.txt	81	121	true	true	5064	160
maze8.txt	41	61	true	true	2412	113
maze9.txt	65	70	true	true	4500	120
maze10.txt	19	28	true	true	564	59
maze11.txt	21	61	true	true	1291	76
maze12.txt	87	103	true	true	9504	128
maze13.txt	27	40	true	true	1373	77
maze14.txt	49	61	true	true	3028	108
maze15.txt	12	33	true	true	249	60

Table 1: *Correctness and step evaluation table*

Note. All mazes are store in "src/data package"

Generally, the Robot can solve all the mazes with moderate steps. On average, the number of steps is directly proportional to the size of the maze. In conclusion, the Robot can solve the maze with a reasonable number of steps. Below is the time execution base on the maze area line graph. We have created numerous mazes with the Robot at the top left corner and the exit gate (X) at the bottom right corner. As can be seen from the graph, the time complexity is relatively linear.



Graph 1: *Time Execution Based on Maze Size Graph*