12/8/2022

RMIT University

Technical Report

Instructor: Tri Dang Tran

COSC2658: Data Structure & Algorithm

Group 15: Dinh, Minh, & Jinho

Contents

[**I.** **Introduction** 2](#_Toc121671268)

[**II.** **High-Level Design** 2](#_Toc121671269)

[**a.** **Software Architecture** 2](#_Toc121671270)

[**b.** **Class Diagram** 3](#_Toc121671271)

[**III.** **Data Structure** 4](#_Toc121671272)

[**a.** **Array** 4](#_Toc121671273)

[**b.** **2D Array** 5](#_Toc121671274)

[**c.** **Queue** 5](#_Toc121671275)

[**d.** **Linked List** 5](#_Toc121671276)

[**IV.** **Algorithm** 6](#_Toc121671277)

[**Breadth First Search (BFS)** 6](#_Toc121671278)

[**Queuing** 6](#_Toc121671279)

[**Get Path** 7](#_Toc121671280)

[**Validate Position** 7](#_Toc121671281)

[**Go** 7](#_Toc121671282)

[**V.** **Complexity Analysis** 7](#_Toc121671283)

[Breadth First Search Complexity 7](#_Toc121671284)

[**Queueing** 8](#_Toc121671285)

[**Get Path** 9](#_Toc121671286)

[**Validate Position** 9](#_Toc121671287)

[**Go** 9](#_Toc121671288)

[**VI.** **Evaluation** 9](#_Toc121671289)

[**Efficiency** 9](#_Toc121671290)

[**Correctness** 9](#_Toc121671291)

# **Introduction**

This report is going to discuss our program’s design with software architecture and class diagram. Furthermore, we will discuss the data structures that were used to construct the program, as well as the algorithms applied to solve the problem. Finally, we will evaluate the time complexity for each algorithm, as well as their correctness for solving the problem.

# **High-Level Design**

## **Software Architecture**

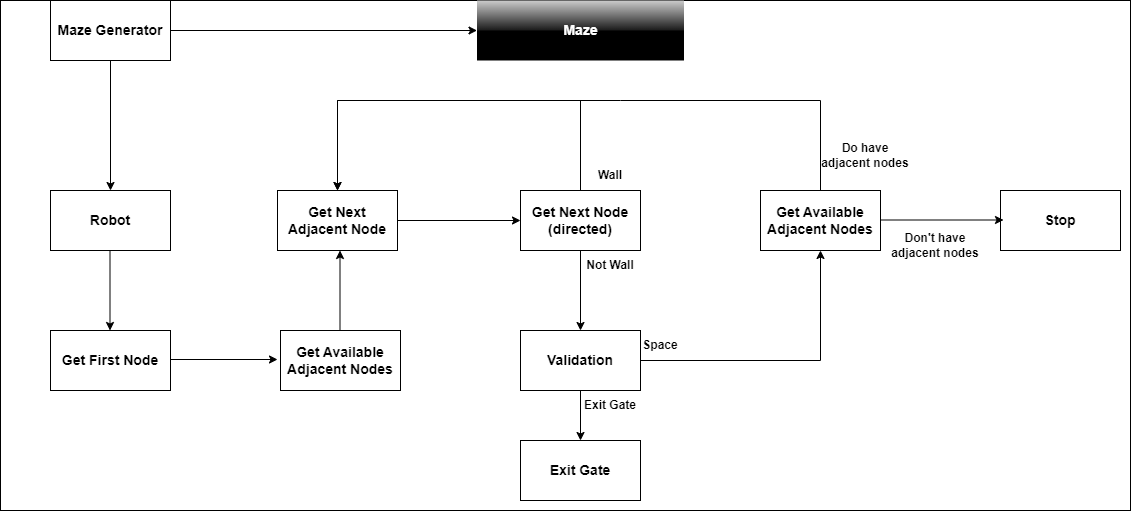


Figure 1. Software Architecture

In this project, the system will generate a maze using 2D arrays along with a robot, an escape gate, and obstacles such as walls. By applying appropriate algorithms, the robot will sequentially explore all the available areas surround it until it meets either dead-end or the exit gate. When a dead-end is met, the robot will return to its previous location and it will explore other areas that have not been visited yet. The steps will be repeated until the robot reaches the exit gate. Nonetheless, if all areas have been explored and the gate is not found, then the maze has no available escape route and the robot is trapped inside forever.

## **Class Diagram**

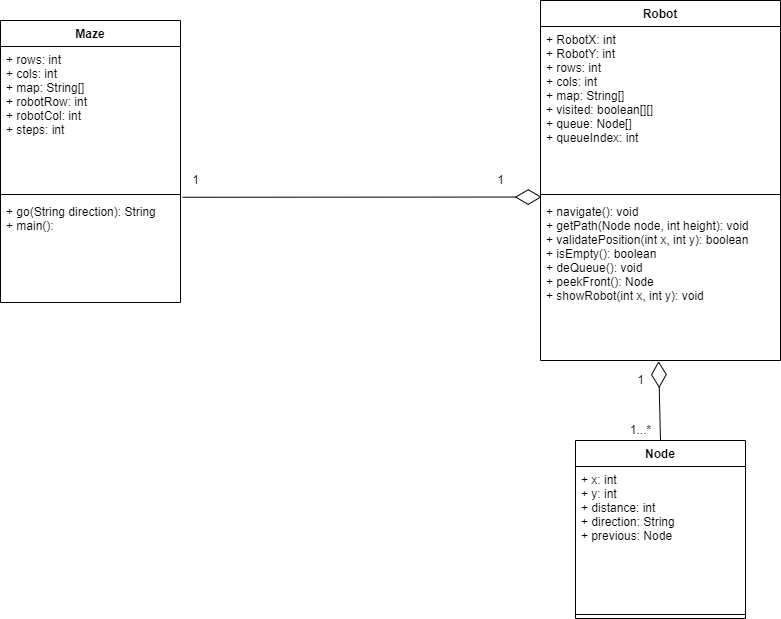


Figure 2. Class Diagram

In this project, we used 3 classes to build the program. They share these following relationship:

* Maze – Robot: Aggregation is their relationship because they can independently exist. However, robot holds the information of a maze because a maze object is initiated in robot class.
* Robot – Node: Aggregation is their relationship because they can independently exist. However, robot holds the information of many nodes to keep track of the path it has been through.

For each class, there are some methods and variables that have their own purpose. The following table will briefly summarize the intention behind all variables and methods implemented in the project:

|  |  |  |
| --- | --- | --- |
| **Class** | **Function** | **Variable** |
| **Maze** | Go(): This function will take in the next directions of the robot. It will update the robot’s position accordingly to the algorithm’s result | Rows: Represents the width of the maze |
| Main(): This function will start the program | Cols: Represents the height of the maze |
| Map: Combination of rows and columns |
| robotCol: X coordinate of the robot in the map (the row it belongs) |
| robotRow: Y coordinate of the robot in the map (the column it belongs) |
| Steps: The nodes that have been visited before the robot reaches the gate |
| **Robot** | Navigate(): Initiate the maze, robot, and all relavant information. | RobotX: X coordinate of the robot in the map (the row it belongs) |
| getPath(): Print out the shortest path from the start point to the exit gate if there is at least one path | RobotY: Y coordinate of the robot in the map (the column it belongs) |
| validatePosition(): Check to see if the robot reaches the border of the maze to make sure it doesn’t go out of bound | Rows: Represents the width of the maze |
| isEmpty(): Check to make sure the queue that stores all the visiting nodes is not empty. If the queue is empty then the robot can’t escape. | Cols: Represents the height of the maze |
| deQueue(): Discard visited nodes from the queue to give rooms for unvisited ones. | Map: Combination of rows and columns |
| peekFront(): Return the unvisited node in the queue | Visited: This array will marked visited nodes on the map |
| showRobot(): Print out the current map and robot’s position for testing purpose | Queue: This array will store the unvisited nodes which will be visiting next by the robot |
| queueIndex: This keeps track of the current size of the queue |
| **Node** |  | X: X coordinate |
| Y: Y coordinate |
| Distance: The height so far, meaning it is the actual length of the shortest path from the start to the gate. |
| Direction: Indicate the next node to visit |
| Previous: Indicate the last node in the linked list. This is used to track the path so far on a specific node |

# **Data Structure**

## **Array**

In this project, we have used array multiple times to store the Object or data that share the same structure. The following are the important data that were saved using array structure:

* Map: We have used a String array to store the map (followed the given maze initiation java file)
* Path: Store the parent nodes from the gate-node (reversed linked list) to print out the shortest path from the start point to exit gate. This is the best way to properly print out the shortest path without using two pointers that point to start and end points to keep track of the path.

## **2D Array**

In this project, we used one 2D-boolean-array for marking visited nodes to make sure the robot doesn’t visit the same node more than once. This helps to reduce the running time since the robot won’t check a wall or a visited node twice. 2D array has a constant access time so it’s very fast to mark/unmark visited nodes. Since the size of the maze won’t change once the program begins, there is no insertion/deletion to alter the map. Therefore, 2D array is the best data structure for keeping visited nodes since it has a fixed size.

Since the map is a String array which each character represents either a wall or a space (also a 2D-array), it is very logical to create another 2D array to keep track of visited nodes and walls. It consumes more space, but it helps to increase the ease of accessing visited nodes for tracking current position and future paths.

## **Queue**

In this project, we used a queue (a special array type) for registering next adjacent/neighbor nodes of current visiting nodes (Breadth First Search approach) for visiting. When a node is being visited, all its neighbors/adjacent nodes are pushed on to the queue, and the visited node will be marked as visited (avoiding future visits on the same visited node) and discarded to let the robot visits next node in the queue.

Since the robot doesn’t know the size of the map, the nodes that will be added to the data structure can be very large. Therefore, we needed a queue to manage flexible amount of nodes efficiently.

Since nodes are visited consecutively, the node that is visited first should be marked as visited first, and deleted from the structure. Hence, the queue data structure also suits best for this purpose since it can perform deletion and insertion at constant time.

## **Linked List**

In this project, we used a Node class to implement our Linked List structure. The node has x and y coordinates that match the exact location on the map (2D array).

Each node object will store its parent node. When the robot reaches the exit gate, the last node (with x and y coordinates match the gate on the map) will be returned to trace back the path from the end to the start. Linked list helps to link consecutive nodes at constant time. Since we are only traverse through the linked path, getting the shortest path will only be the height of the tree.

This helps to reduce the implementation of another array to keep track of visited nodes (which will be huge in size and have high time complexity to insert/delete), or the implementation of two or more pointers to track the start, middle, and end of the path, thus, saving a lot of memory space for other purposes.

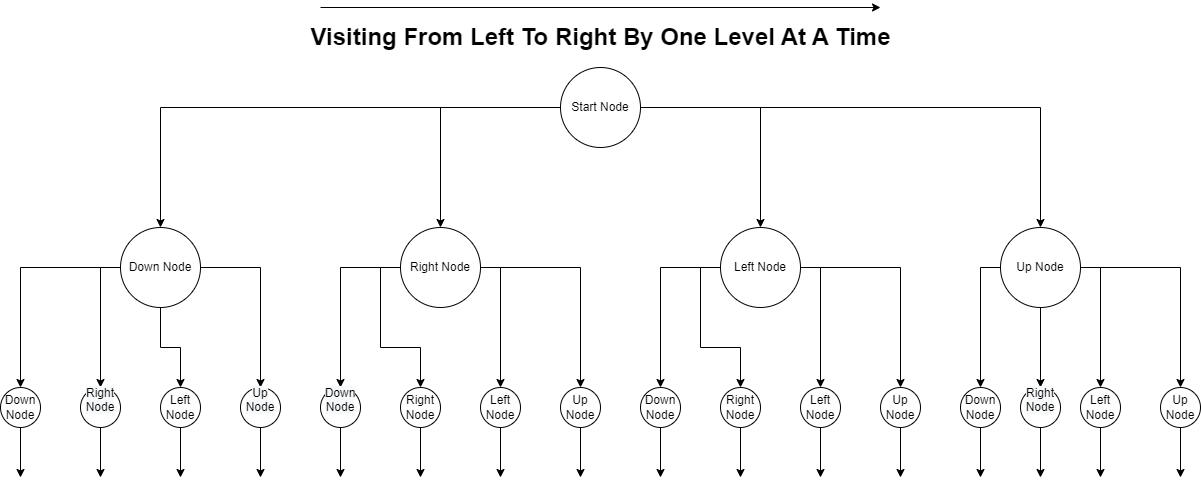
# **Algorithm**

## **Breadth First Search (BFS)**

In this project, to optimize the pathing of the robot, we have used the breadth first search algorithm. The algorithm can be described as below:

1. The robot is initiated at a specific location in the maze
2. The robot marks the node it stays at as visited
3. The robot queue up current node’s adjacent nodes (UP, DOWN, RIGHT, LEFT if not wall)
4. The robot proceeds to visit the next node in the queue (UP,DOWN,RIGHT,LEFT)
5. Repeat step 2 - > step 4 until either the robot reaches the end gate (win) or all nodes are visited (lose).

Using breadth first search ensures all possible path are uniformly visited (visiting one node at a time from left to right), thus, the final path when the robot reaches the gate is guarantee to be the shortest path. Moreover, breadth first search also avoids non-optimal paths such as when the robot follows one direction.



**Figure 3. BFS Illustration**

The breadth first search consumes a lot of memory spaces since the queue can grow exponentially. However, since the requirement is aiming for the shortest path, it’s ideal to uniformly visit each possible path.

## **Queuing**

In this project, to optimize insertion/deletion of adjacent nodes of current node for visiting, we used the queue data structure with its enqueue method that insert the new node to the end of the queue with constant time. Since the queue structure is an array, accessing the first element of the queue is also a constant time. However, to delete the visited node at the front of the queue, we need to shift all elements, there for it is a linear method. The algorithm can be described as below:

1. The current node of the robot is enqueue
2. The available adjacent nodes (not walls and not visited yet) are queued up after
3. Dequeue the current visiting node from the queue (first element of the queue)
4. Repeat step 1 - > step 3 until the queue is empty

The queueing algorithm works simultaneously with the breadth first search algorithm, making sure we have a fast time complexity.

## **Get Path**

The algorithm will use the last node (exit gate) to traverse back on the path of the robot. Since each node contains its parent node (previous location) and the path’s length so far, the robot is capable of printing out the path by saving all nodes to an array (known path’s length). This array will then be used to print out the path that the robot traversed through from its start point to the end gate.

## **Validate Position**

The algorithm will use the map’s row and column attributes to indicate the border where the robot can’t cross. The algorithm is designed to stop the robot moving further (out of bound) if it reaches one of the four borders of the map.

## **Go**

This algorithm will check the next node based on provided direction by the robot to decide the robot’s next action. There are three possible actions for the robot to follow:

* Escape: The next node is the exit gate
* Move: The robot will go to the next node based on the given direction
* Stay: The robot will stay at current position if the next node is a wall or border

# **Complexity Analysis**

## Breadth First Search Complexity

Since the map is constructed using an array of String

String[] map = new String[n];

We consider it as an adjacency matrix because each character inside a String element of the array is a node with x is the String element index in the array, and y is the character index in the String element. Therefore, we have the map’s format in 2D array represents as:

Char[][] map = new char[i][j];

With i is the number of rows (Length of a String element), and j is the number of columns (Length of the map array).

Let’s call each node is a vertex (V) that the robot needs to visit and validate if that vertex is the exit gate. The BFS algorithms start with a vertex, and it needs to traverse through all the remaining vertices to validate if other vertices aren’t visited yet.

Throughout the whole program,

To analyze the time complexity of breath first search, we have three different scenarios:

|  |  |  |
| --- | --- | --- |
| **Case** | **Rationale** | **Time Complexity** |
| **Worst Case** | The robot locates top left (map[0][0]) and exit gate locates bottom right (map[i][j]),  The robot will need to visit all vertices multiple times for each visiting vertex since one is for marking visited, and many times are to check if it’s already visited.  This is the deepest level of visiting (max height of the tree), where the gate is at last char of the String element, or last String index of the array | **O(V \* V) = O (V2)**  With V is the vertices in the map |
| **Average Case** | The robot needs to traverse through half the vertices (V/2) | **O (V/2 \* V/2) = O(V2 / 4)**  **= O (V2)**  With V is the vertices in the map |
| **Best Case** | The exit gate locates next to the robot’s start position | **O(1)** |

Since we must only consider the worst case, we have the time complexity for BFS is: O(V2) **[1]**

## **Queueing**

To analyze the time complexity of queueing methods, we have listed each method’s time complexity below:

|  |  |  |
| --- | --- | --- |
| Method | Rationale | Time Complexity |
| Enqueue | Add the next neighbor node of the current visiting node to the queue  The neighbor node must be a space and not visited  All cases share the same time complexity | O(1) |
| Dequeue | Remove the first element in the queue and shift rest elements to the left  All cases share the same time complexity | O(n)  With n being the number of elements in the queue |
| PeekFront | Return the first element in the queue for processing  All cases share the same time complexity | O(1) |
| isEmpty | Check if the queue is empty  All cases share the same time complexity | O(1): Queue is empty or the first element exists |

Finally, we have the time complexity for queueing is: O(1) + O(n) + O(1) + O(1) = O(n) **[2]**

## **Get Path**

Inside this algorithm, we have the following operators:

If + while loop + constant + constant + for loop + constant = O(1) + O(n) + O(1) + O(1) +O(n)+O(1)

The most significant time complexity operators are while loop and for loop.

Finally, we have time complexity = O(n) + O(n) = O(n) with n is the length of the linked list **[3]**

## **Validate Position**

This algorithm only consists of an “if” statement, therefore we have:

Time Complexity = O(1) **[4]**

## **Go**

Inside this algorithm, we have the following operators:

If + If + if = O(1) + O(1) + O(1)

Some “If” statements have multiple “else if”, however, these “else if” consists of only constant operations.

Therefore, we have time complexity = O(1) + O(1) = O(1) = O(1) **[5]**

# **Evaluation**

## **Efficiency**

From [1], [2], [3], [4], and [5], the program will take at most the O(V2) to send the robot to the exit gate. The program will grow exponentially as the input size grows (as rows and columns increase).

## **Correctness**

With BFS, we are guaranteed the exit gate will always be found, and the robot will always take the shortest path to reach the gate. This is guaranteed because BFS explore potential paths uniformly (one node increase at a time), as long as there is an exit gate.