**Assignment Report**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Student** | **Analyze requirement** | **Developer** | **Tester** | **Reporter** |
| Nguyễn Ngọc Nhân | X | X |  |  |
| Lý Bảo Huy | X |  |  | X |
| Mai Phước Thành | X |  | X |  |

**Assignment 1: Graph problem**

1. **Requirements**
2. Input file **“*maze\_input.txt”***: Provided a tunnel structure in a maze form
3. Tasks

* **Step 1:** Creating the diagram of the tunnel (diagram problem)
* **Step 2:** Finding the shortest path from a certain station to an exit position. (shortest pathproblem)

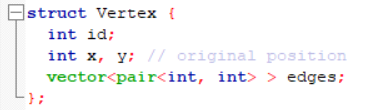
1. Output

Two files: “***diagram.txt”***, and “***shortest\_path.txt”***

* ***Diagram.txt***: it is the result of step 1 and it shows the weighted graph in an adjacency-list format. Each adjacent edge has an attached weight
* ***Shortest\_path.txt***: it is the result of step 2 and it shows the shortest path from a place to an exit position

1. **How to represent a graph by our data structure?**

* Vertex: using Struct in C++. Each Vertex will contain Vertex ID, Vertex Position (a pair of integers X (First) & Y (Second) ) and Edge



* Edge: using Vector in C++. Each Edge has a pair of integers X (First) & Y (Second). With
  + X: ID of the next Vertex of Edge.
  + Y: the length of Edge



* Graph: using Vector in C++ and it is a set of Vertex



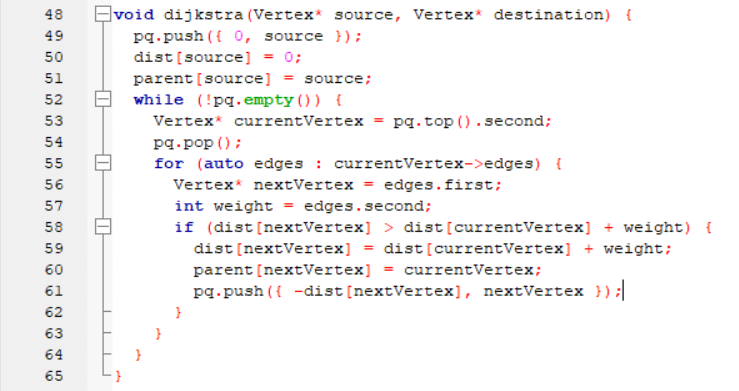
1. **How to construct the graph from the input**

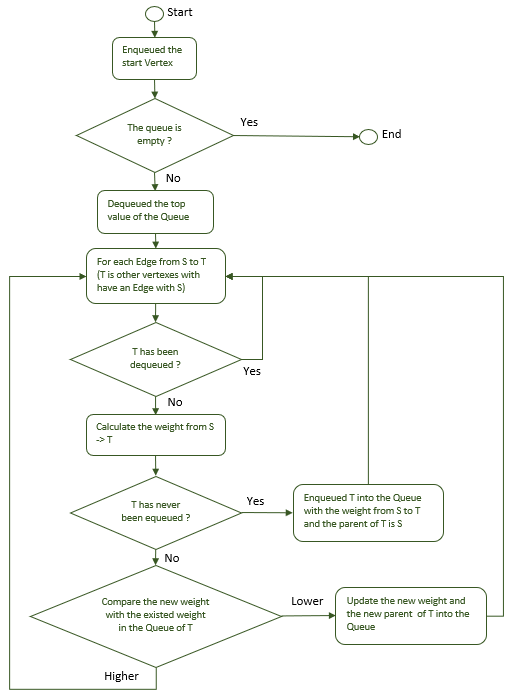
|  |  |
| --- | --- |
| **Input Maze** | **Output Graph** |
| 11 8 #.###### #.###### #......# #.###.## #..#..## #.##.### #....... #.###### #.##.### #......# ######## 17 55 |  |
| **Definition**   * **#** : The wall * **.** : The blank * 11 : the number of rows * 8 : the number of columns * 17 : Position of a station. Here, it is vertex B whose ID is 17 = 2 \* 8 + 1 = (2, 1) * 55 : Exit position. Here, it is vertex I whose ID is 55 = 6 \* 8 + 7 = (6, 7) * A maze cannot contain a square of blanks, such as the following grid   #### #..# #..# #### | |

* Firstly, we use 2D array of integer to store the input graph (called Maze[][]), if the value is “.”, then the value in Maze[][] is 1
* Secondly, for each element of Maze[][] :
  + If the value equals to 0, we will skip it and move to the next element because it is a wall
  + After that, we will check four elements next to the current element (left side, right side, up side & down side), if the value is a blank, then it is a possible way to move. In case the numbers of way are not equal to two or this element is the start/end point, we can consider it is one Vertex and store it into Graph
  + Then we will explore the Graph and capture the length between two Vertices

1. **The algorithms used to fulfill the project requirement.**

* To solve the project requirement, we need to generate the graph from the inputted maze from step 1 by executing **build-graph.cpp**. Then we use Dijkstra algorithm to find the shortest path from the start point to the end point by executing **shortest-path.cpp**
* Input: **Diagram.txt** (the graph which was built in step 1)
* Output: **Shortest\_Path.txt** (the shortest path and the distance of this path)





1. **Source code**



1. **Test cases :**



**Assignment 2: Tic-tac-toe**

1. **How do we represent a state**

For Tic-tac-toe game, we will have 2 players: Player and Computer, and the state is each time Computer or Player making a move to play.

1. **How do we evaluate the static score of a state**

* Firstly, we define that the heuristic score is the constant value and it equals to 50
* Secondly, we will check if the next move is the winning move or not. For TicTacToe, we have 8 winning moves. For each turn, we will recall the Minimax Algorithm to the leaf of tree game then check and return the score:
  + If computer has a winning move, the EvaluateBoard function will return the heuristic score’s value,
  + If player has a winning move, the EvaluateBoard function will return the negative heuristic score’s value,

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

* If there is no possible move to play, the EvaluateBoard function will return score 0

1. **The strategy to choose the next move**

To build this game, we use Heuristic Search and Minimax algorithm.

* **Step 1:** for the first move, because the middle position has the most winning moves, we will check if it is Computer’s turn and it is empty, Computer will always play this position.
* **Step 2:** for each time the Player played, for each empty position (which can be played by computer), we use Minimax Algorithm and Heuristic Search to check and calculate the score for this position
  + If this is the leaf (Win or Lose) or there is no more move to play, we will return the score which is calculated by EvaluateBoard function.
  + Else, the Minimax Algorithm will check for the next empty position.

private int MinimaxAlgo(Grid board, int depth, bool isMax)

{

int score = EvaluateBoard(board); // calculate score for this position

if (score == \_heuristic)

{

return score - depth;

}

if (score == -\_heuristic)

{

return score + depth;

}

if (ContainEmptySquare(board) == false) // Check for empty position to play

{

return 0;

}

int best;

if (isMax) // if this is the Computer move

{

best = -1000;

foreach (Button button in board.Children)

{

if (button.Content.ToString() == string.Empty)

{

button.Content = Computer;

// Get maximum score between each move

best = Math.Max(best, MinimaxAlgo(board, depth + 1, !isMax));

button.Content = string.Empty;

}

}

}

Else // if this is the Opponent - Player move

{

best = 1000;

foreach (Button button in board.Children)

{

if (button.Content.ToString() == string.Empty)

{

button.Content = Player;

// Get minimum score between each move

best = Math.Min(best, MinimaxAlgo(board, depth + 1, !isMax));

button.Content = string.Empty;

}

}

}

return best;

}

* **Step 3:** Compare the heuristic score between each possible move and choice the best one to play.



