**“*Towards Global Technological Excellence”***

A Minor Project Report on

**OPTIPATH**

**(Path Finding Visualizer)**

**For the Degree of**

**Bachelor of Technology in**

**Computer Science and Engineering**

By:

Sakshi V. Bele (20005039)

Mrunal M. Deshmukh (20005040)

Shifanaaz H. Sheikh(20005041)

Aishwarya S. Dhanvijay(20005055)

**Under the Guidance of**

**Prof. Dr. P. N. Chatur**

**Prof. Dr. K. A. WAGHMARE**



**Department of Computer Science and Engineering**

**Government College of Engineering, Amravati**

**(An autonomous Institute of Government of Maharashtra)**

**Maharashtra State, India**

**2022 - 2023**

# **CERTIFICATE**

This is to certify that the Minor Project Report entitled **“OptiPath”**, which is being submitted in partial fulfillment of the requirements for the degree of **‘Bachelor of Technology’** in **‘Computer Science and Engineering’** of Shri. Sant Gadge Baba Amravati University, Amravati is the result of the Work and Contribution by **‘Miss. Sakshi V. Bele, Miss. Mrunal M. Deshmukh, Miss. Shifanaaz H. Sheikh, Miss. Aishwarya S. Dhanvijay’** under my supervision and guidance. The work embodied in this report has not formed earlier for the basis of the award of any degree or compatible certificate or similar title of this for any other diploma/examining body or university to the best of knowledge and belief.

|  |  |
| --- | --- |
| **Prof. Dr. P. N. Chatur** | **Prof.Dr. K. A. Waghmare** |
| Guide | Guide |
| Department of Computer | Department of Computer |
| Science & Engineering | Science & Engineering |

**Prof. Dr. A. V. Deorankar**

Head

Department of Computer

Science & Engineering

**DECLARATION**

I hereby declare that I have formed, completed and written the Minor Project Report entitled **“OptiPath”**. It has not previously been submitted for the basis of the award of any degree or diploma or other similar title of this for any other diploma / examining body or university.

**Sakshi V. Bele** **(20005039)**

**Mrunal M. Deshmukh (20005040)**

**Shifanaaz H. Sheikh** **(20005041)**

**Aishwarya S. Dhanvijay (20005055)**

**Place:** Amravati

**Date:-**

**Abstract**

**OptiPath** aims to develop a software tool that allows users to visually explore and compare different path finding algorithms in graphs. This report presents a comprehensive analysis of the Path Finding Visualizer project, focusing on the implementation and evaluation of several popular path finding algorithms. The algorithms included in this study are Dijkstra's algorithm, A\* algorithm, Breadth-First Search (BFS), and Depth-First Search (DFS). The project utilizes a user-friendly graphical interface to create an interactive environment where users can define custom graph layouts and obstacles. The visualization component allows users to observe the step-by-step execution of the chosen algorithms, providing insights into their workings and performance characteristics.

Contents

1. Introduction
   1. Introduction
   2. Aim
   3. Objectives
   4. Motivation
2. Methodology
   1. Dijkstra's Algorithm
   2. A\* Search
   3. BFS (Breadth First Search)
   4. DFS (Depth First Search)
3. Maze & Patterns
   1. Recursive Division
   2. Basic Random Maze
   3. Simple Stair Pattern
4. Functionalities
5. Flowchart
6. Result
7. Conclusion

7.1 Conclusion

7.2 Future Scope

References

**1.INTRODUCTION**

**1.1 INTRODUCTION**

The Pathfinding Visualizer project is a web-based application designed to provide user s with an interactive and visual representation of various pathfinding algorithms. It serves as a powerful educational tool for understanding the principles and mechanics behind different pathfinding techniques. By offering a hands-on experience, users can explore and analyze how algorithms navigate through complex environments to find the shortest path between two points.

Pathfinding is a fundamental problem in computer science and has numerous real-world applications. From robotic navigation to GPS routing, efficient pathfinding algorithms are crucial for optimizing routes, planning movements, and minimizing costs in various scenarios. The Pathfinding Visualizer project aims to simplify these complex algorithms and make them accessible to users of all levels of programming and algorithmic knowledge.

By combining interactive elements, intuitive visualization, and educational content, the Pathfinding Visualizer provides an engaging environment where users can experiment, learn, and gain a deeper understanding of pathfinding algorithms. Whether you're a student, a developer, or simply curious about the inner workings of these algorithms, the Pathfinding Visualizer offers an immersive learning experience.

The application showcases popular pathfinding algorithms, including Dijkstra's algorithm, A\* algorithm, Breadth-First Search (BFS), and Depth-First Search (DFS). These algorithms demonstrate different strategies for exploring and searching the graph, considering factors such as node distances, heuristics, and exploration patterns. Users can customize the grid, add obstacles, set start and end points, and observe how the algorithms adapt and find the optimal path in real-time.

Overall, the Pathfinding Visualizer project offers a user-friendly, visually appealing, and educational platform for exploring and understanding pathfinding algorithms. It combines the power of interactive visualization, customizable grid environments, and informative content to provide an immersive learning experience for users interested in the fascinating world of pathfinding and algorithmic problem-solving.

**1.2 AIM:**

The Pathfinding Visualizer project aims to provide an interactive and educational platform for users to visualize and understand pathfinding algorithms. It offers a user-friendly interface for customization and real-time observation of the pathfinding process. The project showcases popular algorithms, highlights their applications, and provides educational content to enhance the learning experience.

**1.3 OBJECTIVES:**

1. Develop an intuitive and user-friendly interface for pathfinding visualization.
2. Implement and showcase popular pathfinding algorithms such as Dijkstra's algorithm, A\* algorithm, Breadth-First Search (BFS), and Depth-First Search (DFS).
3. Allow users to customize the grid, including adding barriers and setting start and end points.
4. Provide real-time visualization of the pathfinding process, highlighting the visited nodes and the final shortest path.
5. Offer educational value by explaining the concepts behind each algorithm and their applications.

**1.4 MOTIVATION:**

The motivation behind the Pathfinding Visualizer project is to simplify and demystify pathfinding algorithms, while providing an engaging and accessible platform for users. Created with inspiration from Clement Mihailescu, the project aims to make learning and understanding pathfinding algorithms intuitive and enjoyable. By offering visualizations, customization options, and educational content, it empowers users to explore, experiment, and gain practical knowledge in the field of pathfinding.

**2.METHODOLOGY**

* 1. **Dijkstra’s Algorithm**

**Dijkstra's Algorithm** is a Graph algorithm **that finds the shortest path** from a source vertex to all other vertices in the Graph (single source shortest path). It is a type of Greedy Algorithm that only works on Weighted Graphs having positive weights. The time complexity of Dijkstra's Algorithm is **O(V2)** with the help of the adjacency matrix representation of the graph. This time complexity can be reduced to **O((V + E) log V)** with the help of an adjacency list representation of the graph, where **V** is the number of vertices and **E** is the number of edges in the graph.

**Fundamentals of Dijkstra’s Algorithm:**

1. Dijkstra's Algorithm begins at the node we select (the source node), and it examines the graph to find the shortest path between that node and all the other nodes in the graph.
2. The Algorithm keeps records of the presently acknowledged shortest distance from each node to the source node, and it updates these values if it finds any shorter path.
3. Once the Algorithm has retrieved the shortest path between the source and another node, that node is marked as 'visited' and included in the path.
4. The procedure continues until all the nodes in the graph have been included in the path. In this manner, we have a path connecting the source node to all other nodes, following the shortest possible path to reach each node.

**Pseudocode for Dijkstra's Algorithm:**

1. We have to maintain a record of the path distance of every node. Therefore, we can store the path distance of each node in an array of size n, where n is the total number of nodes.
2. Moreover, we want to retrieve the shortest path along with the length of that path. To overcome this problem, we will map each node to the node that last updated its path length.
3. Once the algorithm is complete, we can backtrack the destination node to the source node to retrieve the path.
4. We can use a minimum Priority Queue to retrieve the node with the least path distance in an efficient way.

**Time and Space Complexity of Dijkstra's Algorithm:**

The Time Complexity of Dijkstra's Algorithm is O(E log V), where E is the number of edges and V is the number of vertices.

The Space Complexity of Dijkstra's Algorithm is O(V), where V is the number of vertices.

**Application:**

1. Finding the shortest path in transportation networks.
2. Routing packets in computer networks.
3. Pathfinding in video games and simulations.
4. Network analysis and optimization.
   1. **A\* Search**

**A\* Search Algorithm** is one of the best and popular technique used in path-finding and graph traversals. A\* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

**Fundamentals of A\* Search Algorithm:**

1. Graph Representation:

A\* works on a graph, which can be represented as nodes and edges. Each node represents a location or state, and the edges represent the connections between nodes. The graph can be either directed or undirected, and the edges can have non-negative weights or costs.

1. Heuristic Function (h(n)):

A\* uses a heuristic function to estimate the cost or distance from each node to the goal. The heuristic function should be admissible, meaning it never overestimates the actual cost. It provides an informed estimate of how close a node is to the goal, guiding the search towards the most promising nodes.

1. Evaluation Function (f(n)):

The evaluation function is used to determine the priority of each node in the search. It combines the cost of reaching a node from the start (g(n)) and the estimated cost to reach the goal (h(n)). The evaluation function is defined as f(n) = g(n) + h(n). A\* selects the node with the lowest f(n) value for expansion.

**Pseudo code of A\* Search Algorithm:**

1. Initialize the algorithm:

* Create an open set, initially containing the start node.
* Create a closed set, initially empty.
* Assign a g-score of 0 to the start node.
* Assign an h-score to the start node, estimating the distance to the destination.

1. While the open set is not empty:
   * Select the node with the lowest f-score (sum of g-score and h-score) from the open set. This node becomes the current node.
   * If the current node is the destination, the path has been found. Stop the algorithm and trace back the path.
   * Move the current node from the open set to the closed set to mark it as visited.
2. Explore neighboring nodes:
   * For each neighbor of the current node:
   * If the neighbor is in the closed set, skip it.
   * Calculate the tentative g-score for the neighbor by adding the cost of the current node to the neighbor's g-score.
   * If the neighbor is not in the open set or the new g-score is lower than its existing g-score:
   * Update the neighbor's g-score to the new, lower value.
   * Calculate the h-score for the neighbor, estimating the remaining distance to the destination.
   * Set the neighbor's parent to the current node.
   * If the neighbor is not in the open set, add it.
3. If the open set becomes empty before reaching the destination, there is no path available.
4. If the destination is reached, trace back the path from the destination node to the start node by following the parent pointers.

**Time and Space Complexity of A\* Search Algorithm:**

Considering a graph, it may take us to travel all the edge to reach the destination cell from the source cell [For example, consider a graph where source and destination nodes are connected by a series of edges, like – 0(source) –>1 –> 2 –> 3 (target). So the worse case time complexity is O(E), where E is the number of edges in the graph of vertices.

**Application:**

1. Pathfinding in Video Game
2. Puzzle Solving
3. Network Routing
4. GPS Navigation Systems
   1. **BFS (Breadth First Search)**

**Breadth First Search (BFS) algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.**

**Fundamentals of BFS Algorithm:**

1. Queue: BFS uses a queue data structure to keep track of the vertices to be visited. The queue follows the First-In-First-Out (FIFO) principle, meaning the vertex that enters the queue first is the first one to be visited.
2. Visited Marking: Similar to DFS, BFS also requires marking visited vertices to avoid revisiting them. This can be done using a visited array or a visited set.
3. Starting Vertex: The algorithm begins by selecting a starting vertex from which to explore the graph. This vertex is marked as visited and enqueued into the queue.
4. Exploring Adjacent Vertices: At each step, the algorithm dequeues a vertex from the front of the queue and explores its adjacent vertices. If an adjacent vertex has not been visited, it is marked as visited and enqueued into the queue.

**Pseudocode for Dijkstra's Algorithm:**

The steps involved in the BFS algorithm to explore a graph are given as follows

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

(waiting state)

[END OF LOOP]

Step 6: EXIT

**Time and Space Complexity of BFS Algorithm:**

Time complexity of BFS depends upon the data structure used to represent the graph. The time complexity of BFS algorithm is O(V+E), since in the worst case, BFS algorithm explores every node and edge. In a graph, the number of vertices is O(V), whereas the number of edges is O(E).

The space complexity of BFS can be expressed as O(V), where V is the number of vertices

**Application:**

1. Shortest path and distance calculation.
2. Connectivity
3. Finding minimum spanning tree
4. Social Network analysis
5. Puzzle solving.
   1. **DFS (Depth First Search)**

**DFS algorithm in the data structure. It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.**

**Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.**

**Fundamentals of DFS Algorithm:**

1. Stack or Recursion: DFS can be implemented using either a stack data structure or recursion. The stack-based approach is more commonly used in iterative implementations, while the recursive approach is simpler to understand.

2. Visited Marking: To avoid visiting the same vertex multiple times and potentially entering an infinite loop, it is important to mark the visited vertices. This can be done by maintaining a visited array or a visited set.

3. Starting Vertex: The algorithm begins by selecting a starting vertex from which to explore the graph. This vertex is marked as visited and added to the stack or passed as an argument in the recursive implementation.

4. Traversing Adjacent Vertices: At each step, the algorithm selects an unvisited adjacent vertex of the current vertex and explores it. This process continues until there are no unvisited adjacent vertices left.

5. Backtracking: When a vertex has no unvisited adjacent vertices, the algorithm backtracks to the previous vertex to explore other unvisited branches. In the stack-based implementation, this is done by popping the top vertex from the stack. In the recursive implementation, the function call stack handles the backtracking automatically.

**Pseudocode for DFS Algorithm:**

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

**Time and Space Complexity of BFS Algorithm:**

The time complexity of the DFS algorithm is O(V+E), where V is the number of vertices and E is the number of edges in the graph.

The space complexity of the DFS algorithm is O(V)

**Application:**

1. Graph Traversal
2. Cycle Detection
3. Topological sorting
4. Backtracking
5. Solving mazes and Puzzles

**3.Maze and Patterns**

A **maze** is a complex network of paths or corridors, often with dead ends and twists, designed as a puzzle or challenge. It typically consists of interconnected passages through which one or more individuals must navigate to reach a specific goal or endpoint. Mazes can be represented in various forms, such as grids, graphs, or physical structures.

**Patterns**, on the other hand, refer to repetitive or recurring designs, arrangements, or sequences. Patterns can be found in nature, mathematics, art, language, and various other domains. They can be simple or intricate, and they often exhibit regularity or predictability. Patterns can be visual, auditory, or conceptual, and they play an essential role in understanding and organizing the world around us.

* 1. **Recursive Division**

Recursive Division is a maze generation algorithm that creates intricate and visually appealing maze patterns. It operates by dividing a rectangular grid recursively into smaller sub-regions and then creating passages within those sub-regions.

The algorithm follows these general steps:

1. Start with a rectangular grid representing the entire maze.

2. Choose a random wall within the grid to create an opening.

3. Divide the remaining region into two sub-regions by adding a wall perpendicular to the chosen wall.

4. Create a passage in one of the walls within the sub-regions, except where the division wall was added.

5. Recursively repeat steps 2-4 for each sub-region, dividing it further until the desired complexity is reached.

6. Continue dividing and creating passages until no further division is possible, resulting in the completion of the maze.

* 1. **Basic Random Maze**

A basic random maze is a maze generated using a randomization algorithm without any specific patterns or constraints. It is created by randomly placing walls or obstacles within a grid structure, forming a complex network of paths and dead ends.

Approach to generating a basic random maze:

1. Create a grid: Start with a grid structure where each cell represents a potential space in the maze. The grid can be represented as a 2D array or any other suitable data structure.

2. Initialize the grid: Initially, mark all cells as walls or obstacles, representing the boundaries of the maze. This means that every cell is unreachable.

3. Select a starting point: Choose a random cell within the grid as the starting point of the maze. This will serve as the entry point.

4. Carve paths: Use a randomization algorithm, such as the Randomized Prim's Algorithm or Recursive Backtracking Algorithm, to carve paths through the maze.

5. Select an endpoint: Once all the paths are carved, mark a random cell within the maze as the endpoint or exit point.

The result of this process is a basic random maze with a maze entrance, paths, dead ends, and an exit point. The maze can be represented visually, with walls represented by filled cells and paths represented by empty cells.

* 1. **Basic Random Maze**

A simple stair pattern is a specific pattern often used in maze generation algorithms to create mazes with a staircase-like structure. It involves placing walls or obstacles in a repeated stair-like pattern, resulting in a maze that resembles a series of steps.

Here's a basic description of how a simple stair pattern can be generated:

1. Create a grid: Start with a grid structure where each cell represents a potential space in the maze. The grid can be represented as a 2D array or any other suitable data structure.

2. Initialize the grid: Initially, mark all cells as walls or obstacles, representing the boundaries of the maze. This means that every cell is unreachable.

3. Define the stair pattern: Determine the desired size and shape of the stairs. The pattern can be adjusted based on the specific requirements and aesthetics of the maze. Typically, the stairs are designed to ascend or descend in a diagonal or straight manner.

4. Place stairs: Starting from a specific corner or edge of the grid, begin placing the stairs according to the defined pattern. Typically, this involves a repeated sequence of steps where walls are removed to create the stair-like structure.

5. Optional: Add additional features: Once the stair pattern is created, you can add additional elements to the maze, such as dead ends, loops, or other obstacles to increase complexity or challenge.

The result of this process is a maze with a staircase-like structure, where the stairs are created by removing walls or obstacles in a defined pattern. The specific design and complexity of the stair pattern can be customized based on the desired outcome.

**4.Functionalities**

The functionalities of an optimal path visualizer minor project typically involve generating and visualizing the optimal path between two points in a graph or maze. Here are some common functionalities that can be included:

1. Graph or Maze Input: Provide an interface for users to input or create a graph or maze. This can be done by allowing users to define the size, shape, and obstacles of the graph or by importing pre-defined graphs or mazes.

2. Start and End Points Selection: Allow users to select the start and end points within the graph or maze. This determines the path that needs to be found.

3. Pathfinding Algorithms: Implement different pathfinding algorithms, such as Dijkstra's algorithm, A\* search, or the Breadth-First Search (BFS) algorithm, to find the optimal path between the start and end points. These algorithms compute the shortest or most efficient path based on a given heuristic or cost function.

4. Path Visualization: Display the graph or maze and visualize the optimal path found by the pathfinding algorithm. This can be done by highlighting the path or marking it in a distinct color to differentiate it from other areas.

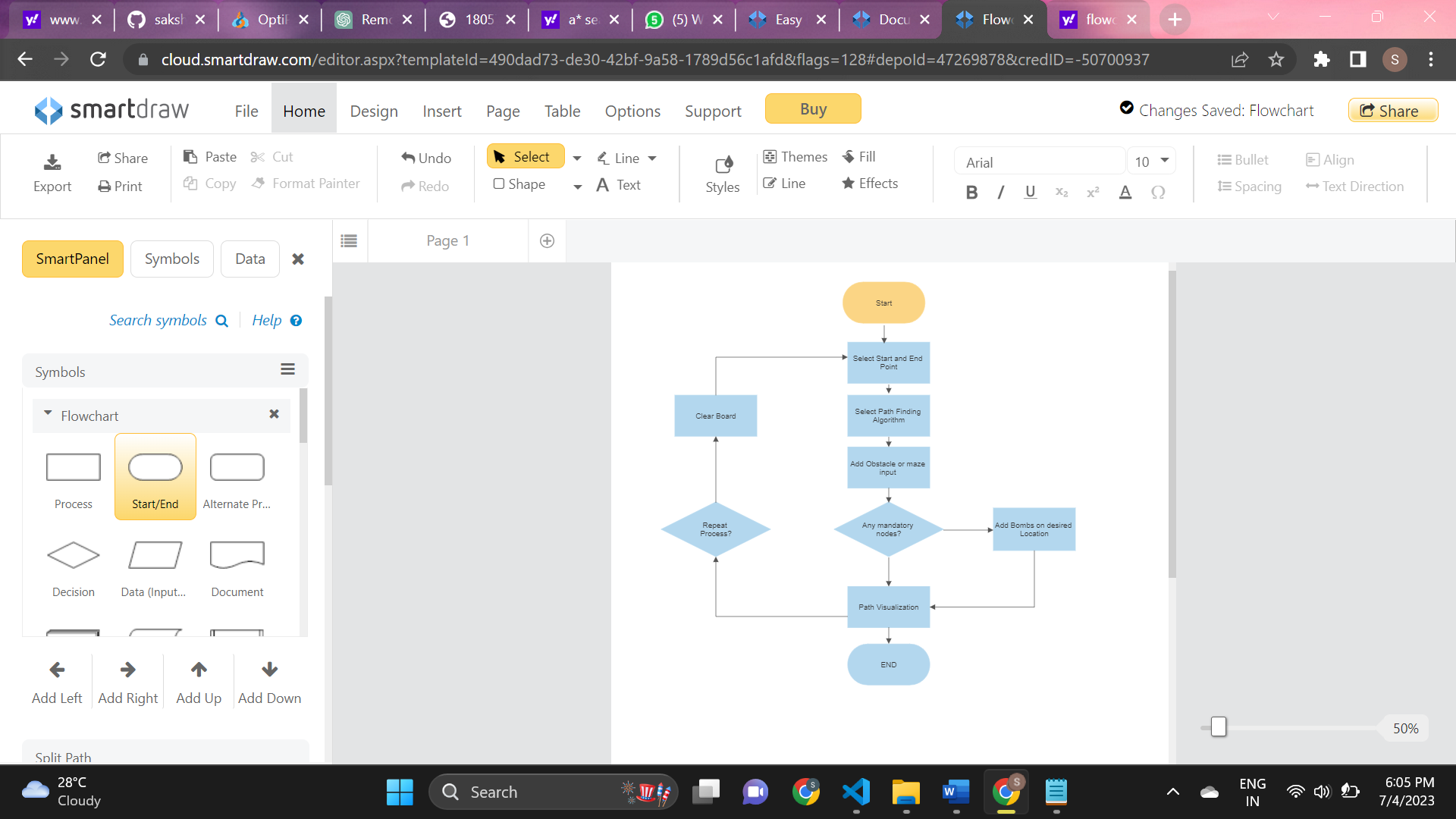
5. Real-Time or Step-by-Step Visualization: Provide an option to visualize the pathfinding algorithm in real-time or step-by-step. Real-time visualization shows the entire process from start to end, while step-by-step visualization allows users to observe the algorithm's progress at each iteration or step.

6. Maze Generation: Offer the ability to generate random mazes for users to test the pathfinding algorithms. This can include different maze generation algorithms like recursive backtracking or randomized Prim's algorithm.

7. Obstacle Editing: Allow users to modify or edit the obstacles within the graph or maze. They can add or remove obstacles to test different scenarios and observe how it affects the optimal path.

These functionalities collectively create an interactive and informative environment for users to understand and visualize the optimal pathfinding process in a graph or maze.

**5.FLOWCHART**

****

Description:

1. Start

2. Selecting Start and End Points

- Guide the user on selecting the start and end points within the graph or maze

- Provide instructions on how to specify these points effectively

3. Pathfinding Algorithms

- Introduce the available pathfinding algorithms in the project

- Explain the characteristics and differences of each algorithm

- Guide the user on choosing the appropriate algorithm for their needs

4. Graph or Maze Input

- Explain the methods for inputting or creating the graph or maze

- Describe how to define the size, shape, and obstacles of the graph or maze

5. Add Obstacles

- Provide details on any additional features, such as maze generation or obstacle editing

- Explain how to utilize these features to customize the graph or maze

6. Path Visualization

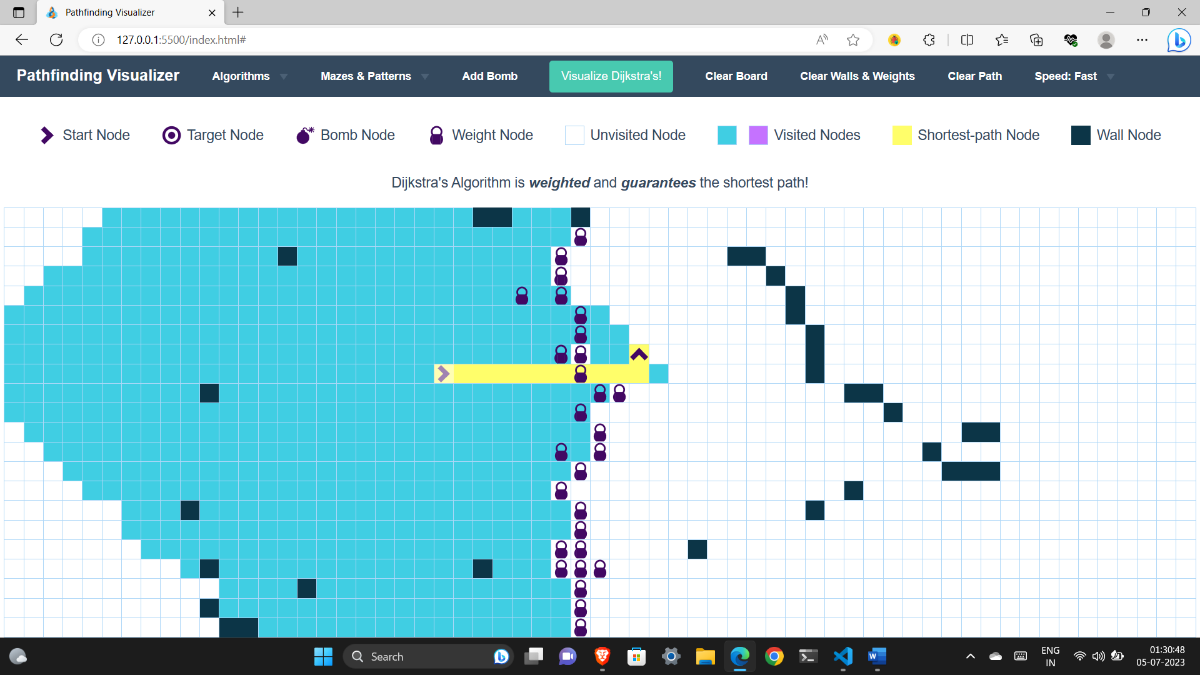
- Explain how the optimal path will be displayed in the application

- Describe any distinct visual markers or colors used to represent the path

7.End

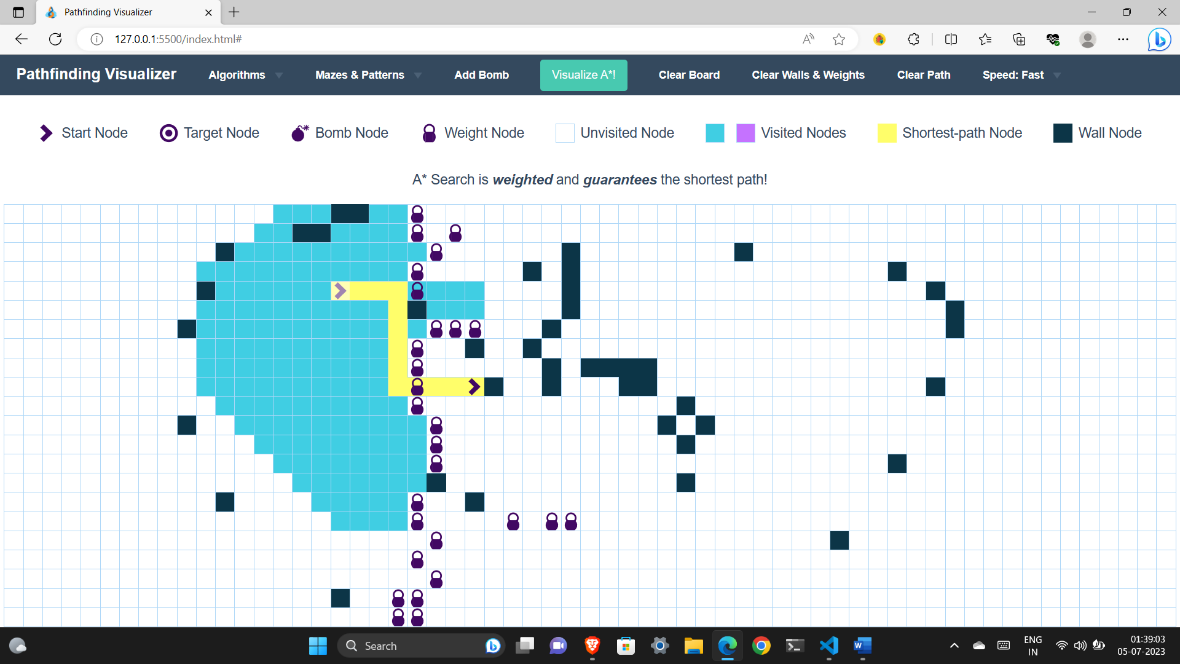
**6.Result**

**PATH FINDING VISUALIZATION FOR DIJKSTRA'S ALGORITHM :**



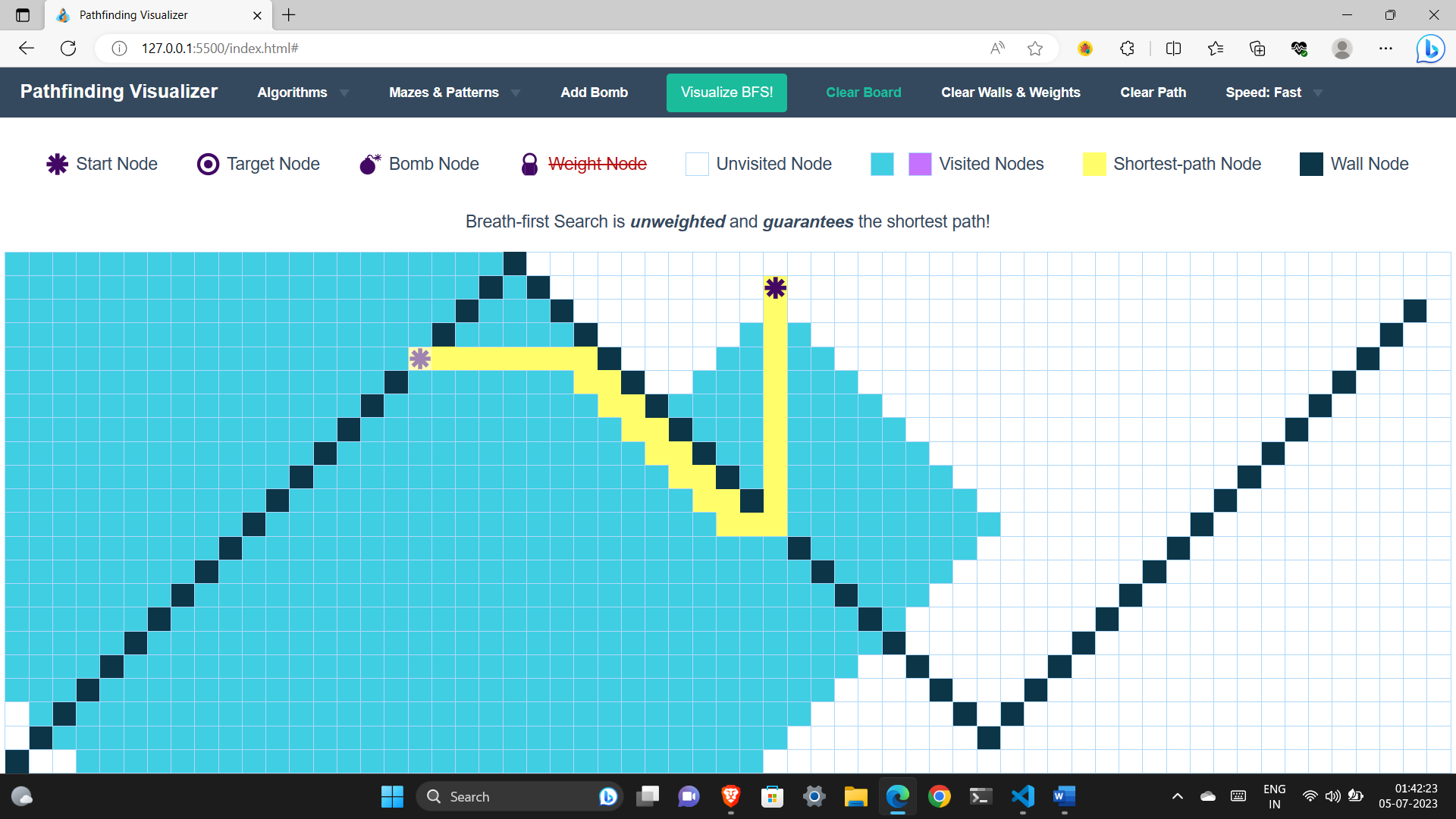
* + Path finding visualizer gives the path for Dijkstra's algorithm which is optimal and shortest by considering all the obstacles of walls and weights.
  + Dijkstra's algorithm is weighted and guarantees the shortest path.

**PATH FINDING VISUALIZATION FOR A\* ALGORITHM :**



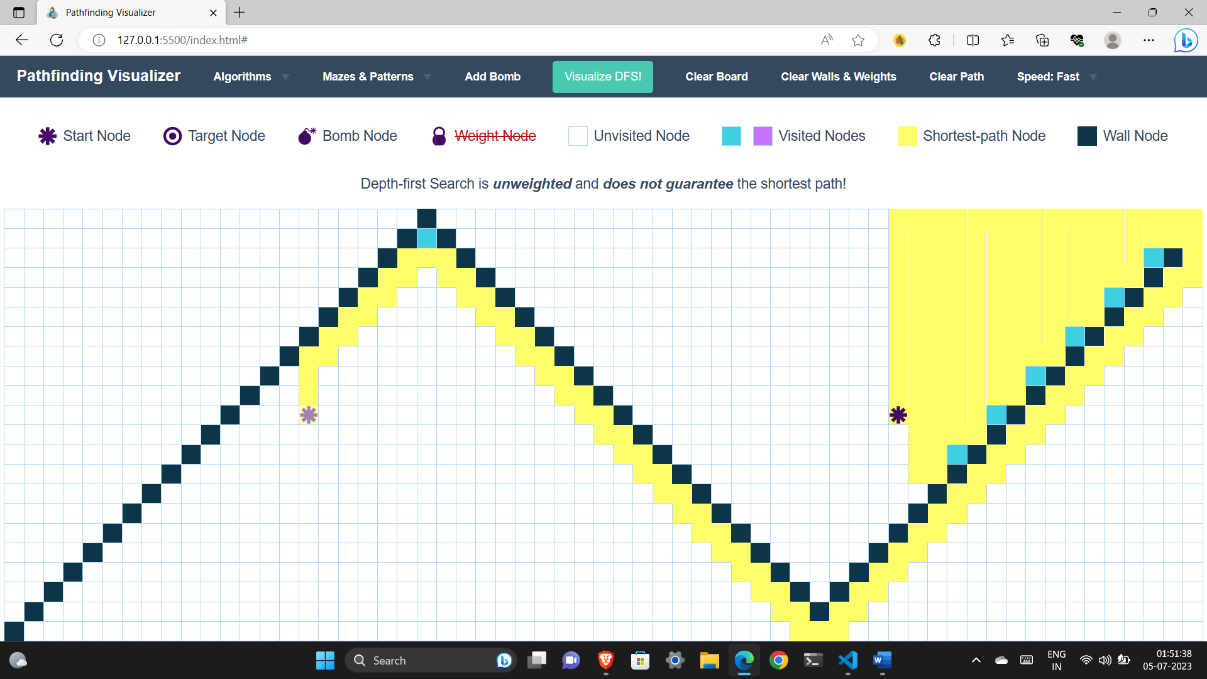
* + Path finding visualizer gives the path for A\* algorithm which shortest by considering all the obstacles of walls and weights.
  + A\* algorithm is weighted and guarantees the shortest path.

**PATH FINDING VISUALIZATION FOR BFS ALGORITHM :**



* + Path finding visualizer gives the path for BFS algorithm which shortest by considering all the obstacles of walls.
  + A\* algorithm is unweighted and guarantees the shortest path

**PATH FINDING VISUALIZATION FOR DFS ALGORITHM :**



* + Path finding visualizer gives the path for DFS algorithm which is may or may not be shortest .
  + DFS algorithm is unweighted and does not guarantees the shortest path.

**7.Conclusion**

**7.1 Conclusion**

In conclusion, an optimal path visualizer minor project provides a valuable tool for understanding and visualizing the process of finding the optimal path between two points in a graph or maze. By implementing various functionalities such as graph or maze input, start and end point selection, pathfinding algorithms, path visualization, real-time or step-by-step visualization, maze generation, obstacle editing, and user interactions, the project offers an interactive and informative environment.

The project enables users to input or create graphs or mazes, select start and end points, and apply different pathfinding algorithms to find the optimal path. The visualization aspect allows users to see the computed path and observe how it traverses through the graph or maze. Real-time or step-by-step visualization options offer flexibility in understanding the algorithm's progress.

Additional features like maze generation and obstacle editing provide users with the ability to test different scenarios and observe how the presence or absence of obstacles impacts the optimal path.

Overall, an optimal path visualizer minor project is a valuable tool for learning and experimenting with pathfinding algorithms, providing a practical and interactive way to understand the concepts of optimal pathfinding in a visually appealing manner.

**7.2 Future Scope**

1. Advanced Algorithms:

2. Real-Time Visualization:

3. Customization Options:

4. Multi-Agent Pathfinding:

5. Integration with Real-World Applications:

**References**

[1]. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

[2]. https://www.geeksforgeeks.org/a-search-algorithm/

[3]. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[4]. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/