LOVELY PROFESSIONAL UNIVERSITY

PROJECT ASSIGNMENT (CA-3)

FOR

FRONT-END WEB DEVELOPMENT (INT219)

Submitted by

Varun Mishra

Registration No : 11803468

ROLL NO. : A22

Programme Name : Btech. CSE

Under the Guidance of

Ms. RUCHITA DUGGAL

School of Computer Science & Engineering

Lovely Professional University, Phagwara

IMPORTANTS LINKS

GitHub Link : https://github.com/LUC4Ri0/Pathfinding-Visualizer

Video Link (My Explanation) : https://drive.google.com/file/d/13CSKSyih5uY8UznnU42OG4GyeQQImiUY/view?usp=sharing

Live Website Link : https://luc4ri0.github.io/Pathfinding-Visualizer/

Motivation:

I have always wanted to build a pathfinding visualizer that animates algorithms. Recently, I have just started my Data structure & Algorithm class as a computer science student. So today, after watching this YouTube video on traversal of graph which is ***BFS & DFS algorithm***, I decided to wait no longer and just start building a simple version of the pathfinding visualization.

Process:

In the process i was able to revisit the following concepts:

1. HTML & BOOTSTRAP
    - NAV-Bar using bootstrap
    - Dropdown using bootstrap
    - Button using bootstrap
    - Flex using bootstrap to indicate the start and target node also to indicate the color of Nodes.
2. CSS
    - Using Normal CSS to attach images in flex.
3. JavaScript
    - QuerySelector & querySelectorAll (for selecting DOM elements)
    - SetTimeout (to create animation effect)
    - CreateElement, getAttribute, setAttribute, appendChild (for modifying the DOM)
    - use math.random function to create random maze generator.
4. Algo
    - I wrote the code based on the video explanation of BFS & DFS algorithm.

Breadth-First Search Shortest Path Algorithm:

Breadth first search is one of the basic and essential searching algorithms on graphs. As a result of how the algorithm works, the path found by breadth first search to any node is the shortest path to that node, i.e the path that contains the smallest number of edges in unweighted graphs. The algorithm works in **O(N + M)** time, where n is the number of vertices and m is the number of edges.

The algorithm takes as input an unweighted graph and the id of the source vertex s. The input graph can be directed or undirected, it does not matter to the algorithm. The algorithm can be understood as a fire spreading on the graph: at the zeroth step only the source s is on fire. At each step, the fire burning at each vertex spreads to all of its neighbors. In one iteration of the algorithm, the "ring of fire" is expanded in width by one unit (hence the name of the algorithm).

More precisely, the algorithm can be stated as follows: Create a queue q which will contain the vertices to be processed and a Boolean array used[ ] which indicates for each vertex, if it has been lit (or visited) or not. Initially, push the source s to the queue and set used[s] = true, and for all other

vertices v set used[v] = false. Then, loop until the queue is empty and in each iteration, pop a vertex from the front of the queue. Iterate through all the edges going out of this vertex and if some of these edges go to vertices that are not already lit, set them on fire and place them in the queue. As a result, when the queue is empty, the "ring of fire" contains all vertices reachable from the source s, with each vertex reached in the shortest possible way. You can also calculate the lengths of the shortest paths (which just requires maintaining an array of path lengths d[]) as well as save information to restore all of these shortest paths (for this, it is necessary to maintain an array of "parents" p[], which stores for each vertex the vertex from which we reached it).

Steps to find Shortest Route in BSF:

- ✔ Step 1: SET STATUS = 1 (ready state) for each node in G
- ✔ Step 2: Enqueue the starting node A and set its STATUS = 2 (wating 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 STAUS = 1) ans set their STATUS = 2 (wating state) [END OF LOOP]
- ✔ Step 6: EXIT

Depth-First Search Shortest Path Algorithm:

Depth first search is one of the main graph algorithms. Depth First Search finds the lexicographical first path in the graph from a source vertex u to each vertex. Depth First Search will also find the shortest paths in a tree (because there only exists one simple path), but on general graphs this is not the case.

The algorithm works in **O(M + N)** time where n is the number of vertices and m is the number of edges.

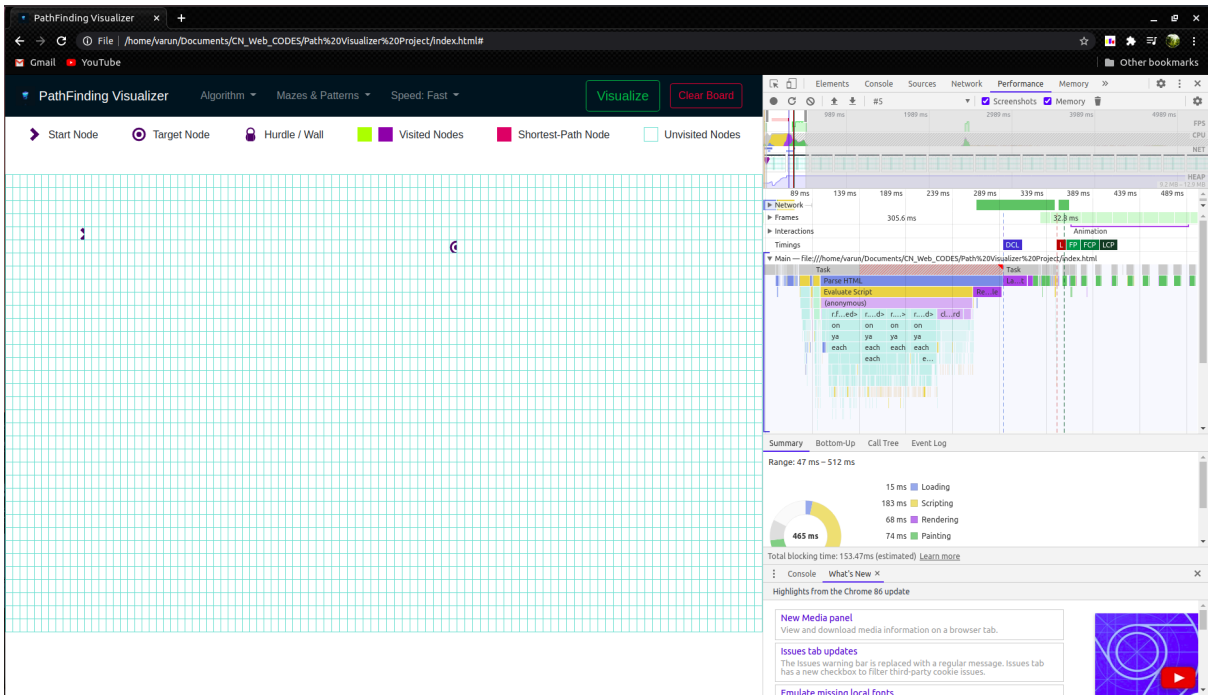
The idea behind DFS is to go as deep into the graph as possible, and backtrack once you are at a vertex without any unvisited adjacent vertices.It is very easy to describe / implement the algorithm recursively: We start the search at one vertex. After visiting a vertex, we further perform a DFS for each adjacent vertex that we haven't visited before. This way we visit all vertices that are reachable from the starting vertex.

Steps to find Shortest Route in BSF:

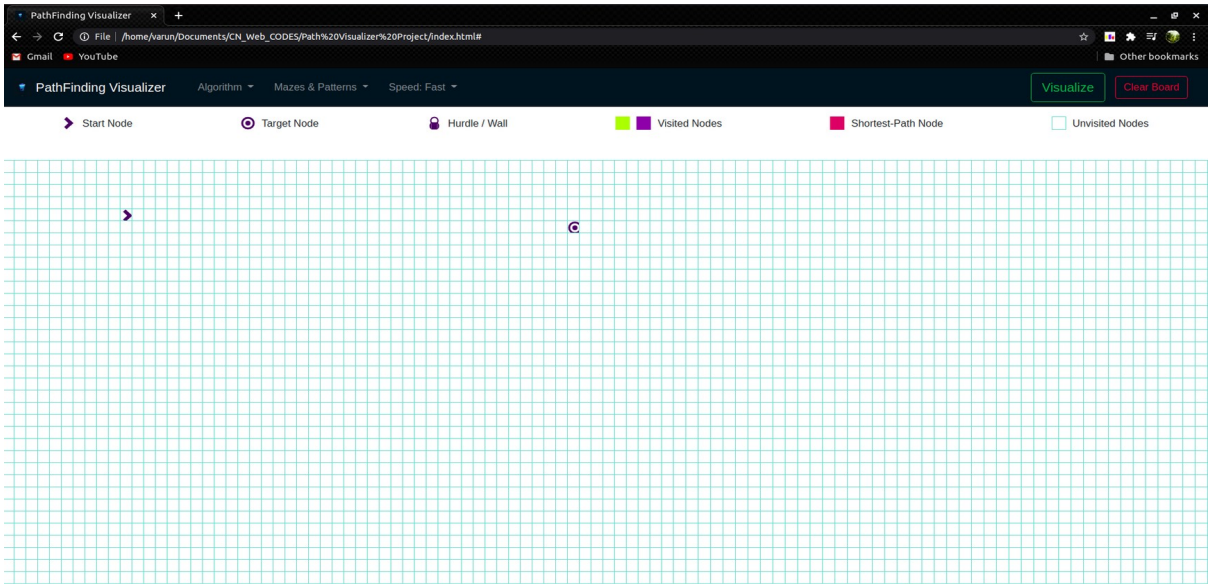
- ✔ Step 1: SET STATUS = 1 (ready state) for each node in G
- ✔ Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- ✔ Step 3: Repeat Steps 4 and 5 until STACK is empty
- ✔ Step 4: POP the top node N. Process it and set its STATUS = 3 (processed state)

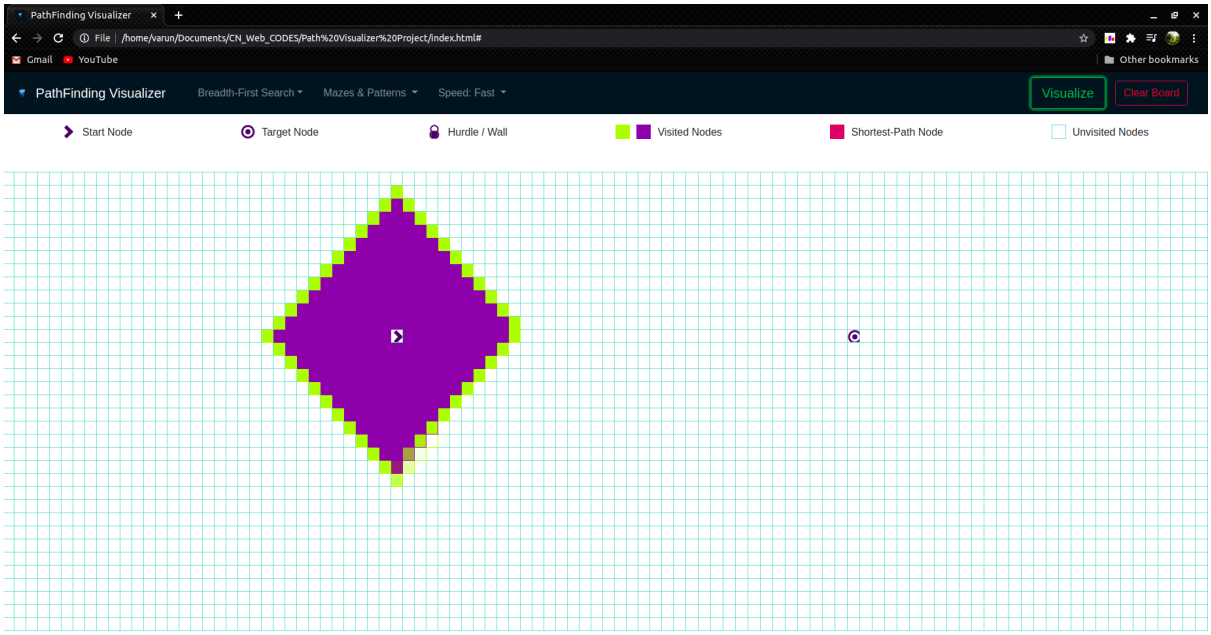
- ✔ Step 5: PUSH on the stack all the neighbours of N that are in the ready state (whose STAUS = 1) ans set their STATUS = 2 (wating state) [END OF LOOP]
- ✔ Step 6: EXIT

ScreenShots

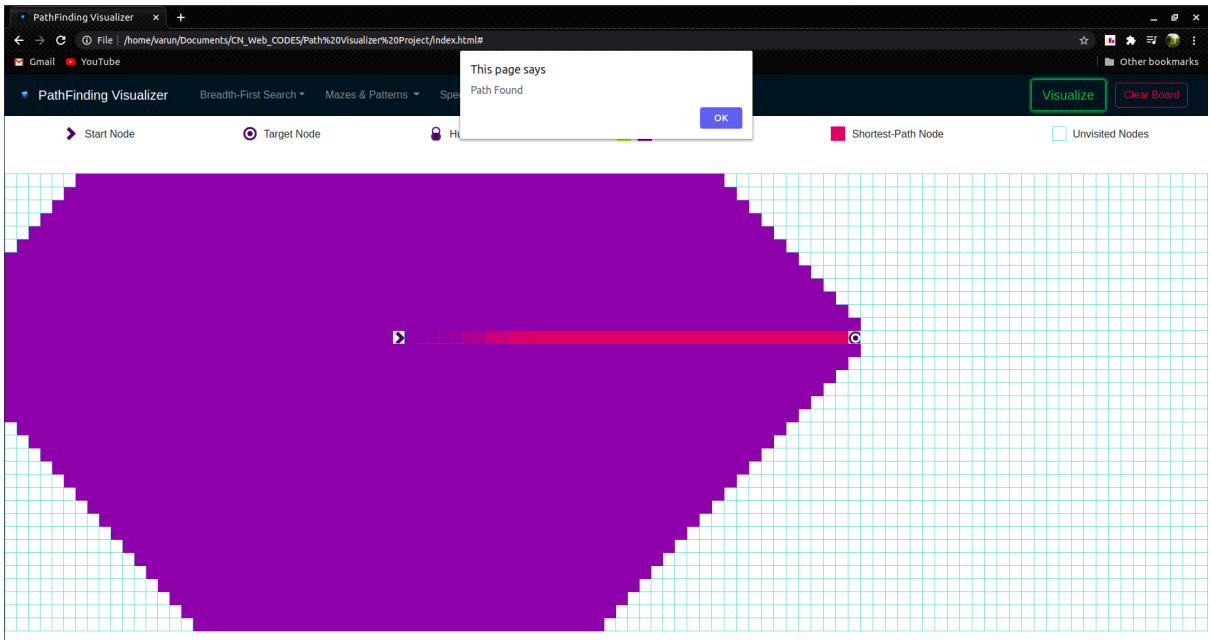


ACTUAL VIEW OF PROJECT

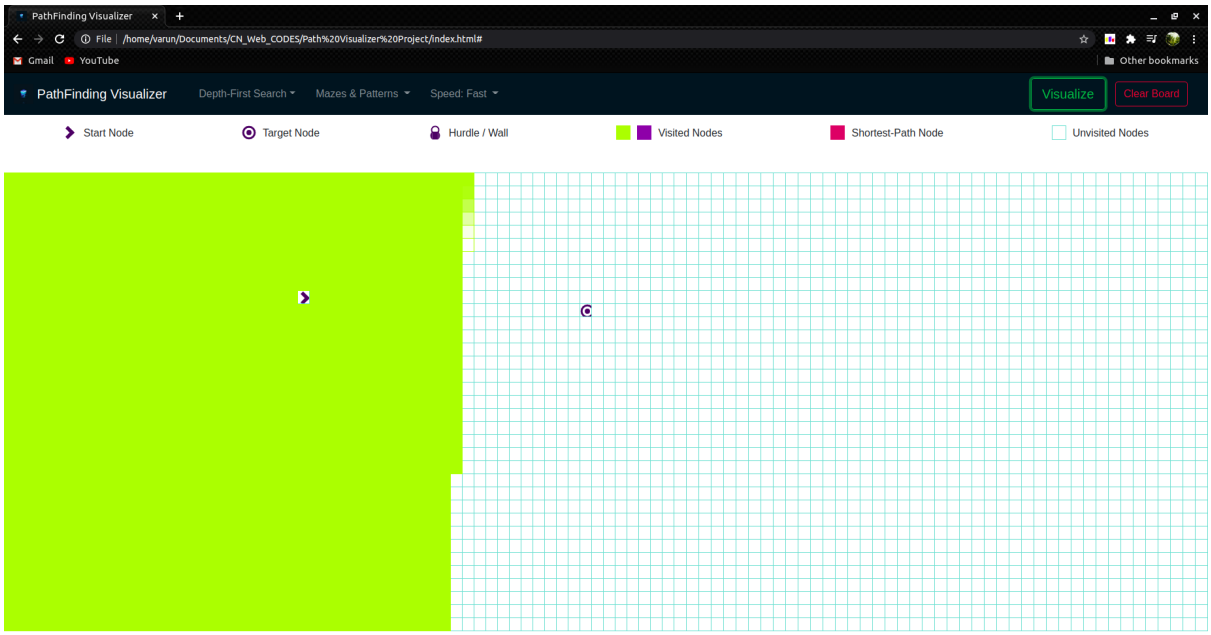




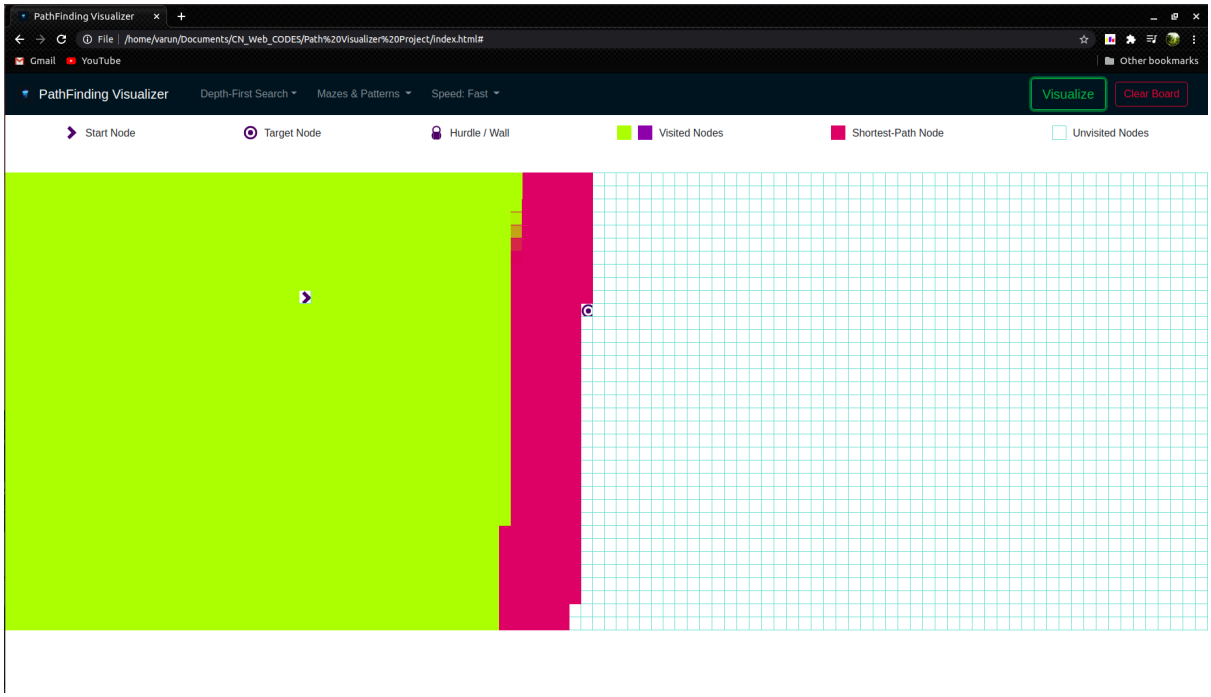
BFS SEARCH

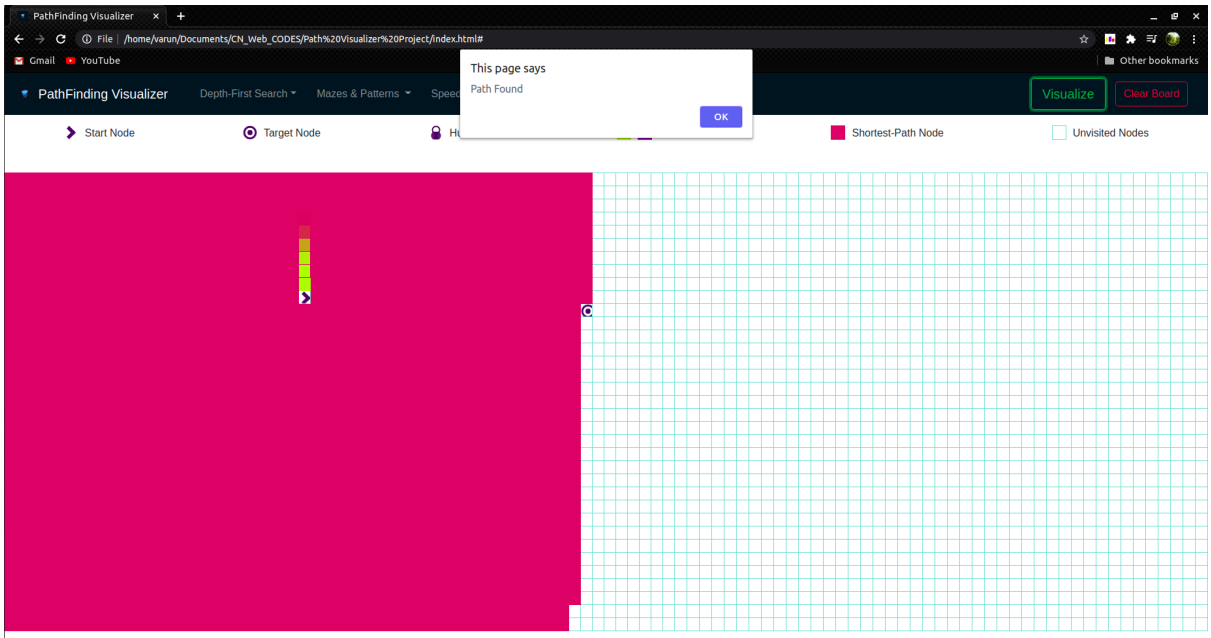


PATH FOUND IN BFS SEARCH


PathFinding Visualizer
File | /home/varun/Documents/CN_Web_CODES/Path%20Visualizer%20Project/index.html#
Gmail
YouTube
Other bookmarks
PathFinding Visualizer
Depth-First Search
Mazes & Patterns
Speed: Fast
Visualize
Clear Board
Start Node
Target Node
Hurdle / Wall
Visited Nodes
Shortest-Path Node
Unvisited Nodes

DFS SEARCH


PathFinding Visualizer
File | /home/varun/Documents/CN_Web_CODES/Path%20Visualizer%20Project/index.html#
Gmail
YouTube
Other bookmarks
PathFinding Visualizer
Depth-First Search
Mazes & Patterns
Speed: Fast
Visualize
Clear Board
Start Node
Target Node
Hurdle / Wall
Visited Nodes
Shortest-Path Node
Unvisited Nodes



PATH FOUND IN DFS SEARCH

RANDOM MAZE GENERATION

Snipped JavaScript Code

```
/*****************************Mouse Function*****************************/
$("td").mousedown(function () {
    var index = $("td").index(this);
    var startCellIndex = (startCell[0] * (totalCols)) + startCell[1];
    var endCellIndex = (endCell[0] * (totalCols)) + endCell[1];
    if (!inProgress) {
        if (justFinished && !inProgress) {
            clearBoard(keepWalls = true);
            justFinished = false;
        }
        if (index == startCellIndex) {
            movingStart = true;
        } else if (index == endCellIndex) {
            movingEnd = true;
        } else {
            createWalls = true;
        }
    }
});
$("td").mouseup(function () {
    createWalls = false;
    movingStart = false;
    movingEnd = false;
});

$("td").mouseenter(function () {
```

```
    if (!createWalls && !movingStart && !movingEnd) { return; }
    var index = $("td").index(this);
    var startCellIndex = (startCell[0] * (totalCols)) + startCell[1];
    var endCellIndex = (endCell[0] * (totalCols)) + endCell[1];
    if (!inProgress) {
        if (justFinished) {
            clearBoard(keepWalls = true);
            justFinished = false;
        }
        if (movingStart && index != endCellIndex) {
            moveStartOrEnd(startCellIndex, index, "start");
        } else if (movingEnd && index != startCellIndex) {
            moveStartOrEnd(endCellIndex, index, "end");
        } else if (index != startCellIndex && index != endCellIndex) {
            $(this).toggleClass("wall");
        }
    }
});

$("td").click(function () {
    var index = $("td").index(this);
    var startCellIndex = (startCell[0] * (totalCols)) + startCell[1];
    var endCellIndex = (endCell[0] * (totalCols)) + endCell[1];
    if ((inProgress == false) && !(index == startCellIndex) && !(index == endCellIndex)) {
        if (justFinished) {
            clearBoard(keepWalls = true);
            justFinished = false;
        }
        $(this).toggleClass("wall");
```

```
    }
});

$("body").mouseup(function () {
    createWalls = false;
    movingStart = false;
    movingEnd = false;
});
/**********************************Buttons**********************************/
$("#startBtn").click(function () {
    if (algorithm == null) { return; }
    if (inProgress) { update("wait"); return; }
    traverseGraph(algorithm);
});

$("#clearBtn").click(function () {
    if (inProgress) { update("wait"); return; }
    clearBoard(keepWalls = false);
});
/*****************************Nav-Bar Menus*****************************/
$("#algorithms .dropdown-item").click(function () {
    if (inProgress) { update("wait"); return; }
    algorithm = $(this).text();

    updatealgo();
});

$("#speed .dropdown-item").click(function () {
```

```
    if (inProgress) { update("wait"); return; }
    animationSpeed = $(this).text();
    updateSpeedDisplay();
});

$("#mazes .dropdown-item").click(function () {
    if (inProgress) { update("wait"); return; }
    maze = $(this).text();
    if (maze == "Random Maze Generation") {
        randomMaze();
    }
});
```

/**********Functions**********/

```
// For updation of start at new location of both start and end
function moveStartOrEnd(prevIndex, newIndex, startOrEnd) {
    var newCellY = newIndex % totalCols;
    var newCellX = Math.floor((newIndex - newCellY) / totalCols);
    if (startOrEnd == "start") {
        startCell = [newCellX, newCellY];
    } else {
        endCell = [newCellX, newCellY];
    }
    clearBoard(keepWalls = true);
    return;
}

function updateSpeedDisplay() {
    if (animationSpeed == "Slow") {
```

```
        $(".Visualizerspeed").text("Speed: Slow");
    }
    else if (animationSpeed == "Average") {
        $(".Visualizerspeed").text("Speed: Average");
    }
    else if (animationSpeed == "Fast") {
        $(".Visualizerspeed").text("Speed: Fast");
    }
    return;
}
/*****************************Algorithm function************************/
function updatealgo() {
    if (algorithm == "Breadth-First Search") {
        $(".algo").text("Breadth-First Search");
    } else if (algorithm == "Depth-First Search") {
        $(".algo").text("Depth-First Search");
    }
    return;
}


async function traverseGraph(algorithm) {
    inProgress = true;
    clearBoard(keepWalls = true);
    var pathFound = executeAlgo();


    await animateCells();
    if (pathFound) {
        alert("Path Found")
    } else {
```

```
        alert("Path Not Found");
    }
    inProgress = false;
    justFinished = true;
}

function executeAlgo() {
    if (algorithm == "Depth-First Search") {
        var visited = createVisited();
        var pathFound = DFS(startCell[0], startCell[1], visited);
    } else if (algorithm == "Breadth-First Search") {
        var pathFound = BFS();
    }
    return pathFound;
}
/*****************************visited for BFS & DFS************************/
function createVisited() {
    var visited = [];
    var cells = $("#tableContainer").find("td");
    for (var i = 0; i < totalRows; i++) {
        var row = [];
        for (var j = 0; j < totalCols; j++) {
            if (cellIsAWall(i, j, cells)) {
                row.push(true);
            } else {
                row.push(false);
            }
        }
        visited.push(row);
```

```
    }
    return visited;
}

function cellIsAWall(i, j, cells) {
    var cellNum = (i * (totalCols)) + j;
    return $(cells[cellNum]).hasClass("wall");
}
/************************Depth-first search function************************/
function DFS(i, j, visited) {
    if (i == endCell[0] && j == endCell[1]) {
        cellsToAnimate.push([[i, j], "success"]);
        return true;
    }
    visited[i][j] = true;
    cellsToAnimate.push([[i, j], "searching"]);
    var neighbors = getNeighbors(i, j);
    for (var k = 0; k < neighbors.length; k++) {
        var m = neighbors[k][0];
        var n = neighbors[k][1];
        if (!visited[m][n]) {
            var pathFound = DFS(m, n, visited);
            if (pathFound) {
                cellsToAnimate.push([[i, j], "success"]);
                return true;
            }
        }
    }
    cellsToAnimate.push([[i, j], "visited"]);
```

```
    return false;
}

/**********************Breadth-first search function**************************/
function BFS() {
    var pathFound = false;
    var myQueue = new Queue();
    var prev = createPrev();
    var visited = createVisited();
    myQueue.enqueue(startCell);
    cellsToAnimate.push(startCell, "searching");
    visited[startCell[0]][startCell[1]] = true;
    while (!myQueue.empty()) {
        var cell = myQueue.dequeue();
        var r = cell[0];
        var c = cell[1];
        cellsToAnimate.push([cell, "visited"]);
        if (r == endCell[0] && c == endCell[1]) {
            pathFound = true;
            break;
        }
        // Put neighboring cells in queue
        var neighbors = getNeighbors(r, c);
        for (var k = 0; k < neighbors.length; k++) {
            var m = neighbors[k][0];
            var n = neighbors[k][1];
            if (visited[m][n]) { continue; }
            visited[m][n] = true;
            prev[m][n] = [r, c];
```

```
            cellsToAnimate.push([neighbors[k], "searching"]);
            myQueue.enqueue(neighbors[k]);
        }
    }
    while (!myQueue.empty()) {
        var cell = myQueue.dequeue();
        var r = cell[0];
        var c = cell[1];
        cellsToAnimate.push([cell, "visited"]);
    }
    if (pathFound) {
        var r = endCell[0];
        var c = endCell[1];
        cellsToAnimate.push([[r, c], "success"]);
        while (prev[r][c] != null) {
            var prevCell = prev[r][c];
            r = prevCell[0];
            c = prevCell[1];
            cellsToAnimate.push([[r, c], "success"]);
        }
    }
    return pathFound;
}
```

/********Random Maze Generation function**********/**

```
async function randomMaze() {
    inProgress = true;
    clearBoard(keepWalls = false);
    var visited = createVisited();
    var walls = makeWalls();
```

```
var cells = [startCell, endCell];
walls[startCell[0]][startCell[1]] = false;
walls[endCell[0]][endCell[1]] = false;
visited[startCell[0]][startCell[1]] = true;
visited[endCell[0]][endCell[1]] = true;
while (cells.length > 0) {
    var random = Math.floor(Math.random() * cells.length);
    var randomCell = cells[random];
    cells[random] = cells[cells.length - 1];
    cells.pop();
    var neighbors = getNeighbors(randomCell[0], randomCell[1]);
    if (neighborsThatAreWalls(neighbors, walls) < 2) { continue; }
    walls[randomCell[0]][randomCell[1]] = false;
    for (var k = 0; k < neighbors.length; k++) {
        var i = neighbors[k][0];
        var j = neighbors[k][1];
        if (visited[i][j]) { continue; }
        visited[i][j] = true;
        cells.push([i, j]);
    }
}
var cells = $("#tableContainer").find("td");
for (var i = 0; i < totalRows; i++) {
    for (var j = 0; j < totalCols; j++) {
        if (i == 0 || i == (totalRows - 1) || j == 0 || j == (totalCols - 1) || walls[i][j]) {
            cellsToAnimate.push([[i, j], "wall"]);
        }
    }
}
```

/**********Clear Board function**********/

```
function clearBoard(keepWalls) {
    var cells = $("#tableContainer").find("td");
    var startCellIndex = (startCell[0] * (totalCols)) + startCell[1];
    var endCellIndex = (endCell[0] * (totalCols)) + endCell[1];
    for (var i = 0; i < cells.length; i++) {
        isWall = $(cells[i]).hasClass("wall");
        $(cells[i]).removeClass();
        if (i == startCellIndex) {
            $(cells[i]).addClass("start");
        } else if (i == endCellIndex) {
            $(cells[i]).addClass("end");
        } else if (keepWalls && isWall) {
            $(cells[i]).addClass("wall");
        }
    }
}

// Ending statements
clearBoard();
```

My Steps in Brief:

1. Starting with the chosen start and target node, update the algorithm which would be used and visualize for the shortest path.
2. Random maze generator is used to generate the huddel in the grid sheet.
3. Select speed for visiualize time interval.

Further Improvements

There are many things to be done for this. Just to name a few:

- Add more algoithms like A* algorithm, Dijkstra’s algotrithms etc.
- Add an tutorial box to learn how to operate.

```
thank you ();
```