ALGORITHM VISUALIZER

TITLE:

To create a website where one can visualize and understand various types of Algorithms.

ABOUT WEBSITE:

Our project enables animating sorting algorithms and path finding algorithms as a learning aid for classroom instruction. A web-based animation tool is created to visualize three common sorting algo-rithms: Bubble Sort, Insertion Sort, and Merge Sort and common path finding algorithms like Dijkstra's Algorithm, BFS and DFS. The animation tool would represent data as a bar-graph and after selecting a data-ordering and algorithm, the user can run an automated animation or step through it at their own pace.The results and responses can be recorded and analyzed in our project will help us improve our website and make it more student friendly.

Sorting Visualizer allows users to visualize sorting algorithms in action with the help of dynamic array sizes and effective animations.

THEORY:

SORTING TECHNIQUES:

Sorting is a very classic problem of reordering items (that can be compared, eg integers, floating-point numbers, strings, etc) of an array (or a list) in a certain order (increasing, non-decreasing, decreasing, non- increasing, lexicographical, etc).
There are many different sorting algorithms, each has its own advantages and limitations.
Sorting is commonly used as the introductory problem in various Computer Science classes to showcase a range of algorithmic ideas.
Sorting problem has a variety of interesting algorithmic solutions that embody many Computer Science ideas:

1. Comparison versus Non-Comparison based strategies,
2. Iterative versus Recursive implementation,
3. Divide-and-Conquer paradigm,
4. Best / Worst / Average-case Time Complexity analysis,
5. Randomized Algorithms etc.

PATHFINDING VISUALIZER:

Pathfinding algorithms are usually an attempt to solve the shortest path problem in graph theory. They try to find the best path given a starting point and ending point based on some predefined criteria.

Path finding algorithms are important because they are used in applications like google maps, satellite navigation systems, routing packets over the internet. The usage of pathfinding algorithms isn't just limited to navigation systems. The overarching idea can be applied to other applications as well.

ALGORITHMS AND CODE SNIPPETS:

Merge Sort Helper Function:

```
function doMerge(
  mainArray,
  startIdx,
  middleIdx,
  endIdx,
  auxiliaryArray,
  animations
) {
  let k = startIdx;
  let i = startIdx;
  let j = middleIdx + 1;
  while (i <= middleIdx && j <= endIdx) {
    // These are the values that we're comparing; we push them once
    // to change their color.
    animations.push([i, j]);
    // These are the values that we're comparing; we push them a second
    // time to revert their color.
    animations.push([i, j]);
    if (auxiliaryArray[i] <= auxiliaryArray[j]) {
      // We overwrite the value at index k in the original array with the
      // value at index i in the auxiliary array.
      animations.push([k, auxiliaryArray[i]]);
      mainArray[k++] = auxiliaryArray[i++];
```

```
    } else {
      // We overwrite the value at index k in the original array with the
      // value at index j in the auxiliary array.
      animations.push([k, auxiliaryArray[j]]);
      mainArray[k++] = auxiliaryArray[j++];
    }
  }
  while (i <= middleIdx) {
    // These are the values that we're comparing; we push them once
    // to change their color.
    animations.push([i, i]);
    // These are the values that we're comparing; we push them a second
    // time to revert their color.
    animations.push([i, i]);
    // We overwrite the value at index k in the original array with the
    // value at index i in the auxiliary array.
    animations.push([k, auxiliaryArray[i]]);
    mainArray[k++] = auxiliaryArray[i++];
  }
  while (j <= endIdx) {
    // These are the values that we're comparing; we push them once
    // to change their color.
    animations.push([j, j]);
    // These are the values that we're comparing; we push them a second
    // time to revert their color.
    animations.push([j, j]);
    // We overwrite the value at index k in the original array with the
    // value at index j in the auxiliary array.
    animations.push([k, auxiliaryArray[j]]);
    mainArray[k++] = auxiliaryArray[j++];
  }
}
```

Merge Sort Helper Function:

```
// Performs Dijkstra's algorithm; returns *all* nodes in the order
// in which they were visited. Also makes nodes point back to their
```

```
// previous node, effectively allowing us to compute the shortest path
// by backtracking from the finish node.
export function dijkstra(grid, startNode, finishNode) {
  const visitedNodesInOrder = [];
  let flagwalls = false;
  let flagwallf = false;
  let flagweights = false;
  let flagweightf = false;
  if (startNode.isWall) flagwalls = true;
  if (finishNode.isWall) flagwallf = true;
  if (startNode.isWeight) flagweights = true;
  if (finishNode.isWeight) flagweightf = true;
  startNode.isWall = false;
  finishNode.isWall = false;
  startNode.isWeight = false;
  finishNode.isWeight = false;
  startNode.distance = 0;
  const unvisitedNodes = getAllNodes(grid);
  while (!!unvisitedNodes.length) {
    sortNodesByDistance(unvisitedNodes);
    const closestNode = unvisitedNodes.shift();
    // If we encounter a wall, we skip it.
    if (closestNode.isWall) continue;
    // If the closest node is at a distance of infinity,
    // we must be trapped and should therefore stop.
    if (closestNode.distance === Infinity) return visitedNodesInOrder;
    closestNode.isVisited = true;
    visitedNodesInOrder.push(closestNode);
    if (closestNode === finishNode) {
      if (flagwalls) startNode.isWall = true;
      if (flagwallf) finishNode.isWall = true;
      if (flagweights) startNode.isWeight = true;
      if (flagweightf) finishNode.isWeight = true;
      return visitedNodesInOrder;
    }
    updateUnvisitedNeighbors(closestNode, grid);
```

```
  }
}

function sortNodesByDistance(unvisitedNodes) {
  unvisitedNodes.sort((nodeA, nodeB) => nodeA.distance - nodeB.distance);
}

function updateUnvisitedNeighbors(node, grid) {
  const unvisitedNeighbors = getUnvisitedNeighbors(node, grid);
  for (const neighbor of unvisitedNeighbors) {
    if (neighbor.isWeight) {
      neighbor.distance = node.distance + 15;
    } else neighbor.distance = node.distance + 1;
    neighbor.previousNode = node;
  }
}

function getUnvisitedNeighbors(node, grid) {
  const neighbors = [];
  const { col, row } = node;
  if (row > 0) neighbors.push(grid[row - 1][col]);
  if (row < grid.length - 1) neighbors.push(grid[row + 1][col]);
  if (col > 0) neighbors.push(grid[row][col - 1]);
  if (col < grid[0].length - 1) neighbors.push(grid[row][col + 1]);
  return neighbors.filter((neighbor) => !neighbor.isVisited);
}

function getAllNodes(grid) {
  const nodes = [];
  for (const row of grid) {
    for (const node of row) {
      nodes.push(node);
    }
  }
  return nodes;
}
```

```
// Backtracks from the finishNode to find the shortest path.
// Only works when called *after* the dijkstra method above.
export function getNodesInShortestPathOrderDijkstra(finishNode) {
  const nodesInShortestPathOrder = [];
  let currentNode = finishNode;
  while (currentNode !== null) {
    nodesInShortestPathOrder.unshift(currentNode);
    currentNode = currentNode.previousNode;
  }
  return nodesInShortestPathOrder;
}
```

Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

1)Pick a starting node and push all its adjacent nodes into a stack.

2)Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

3)Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

Breadth First Search (BFS)

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer.
2. Move to the next layer and do the same for this layer until we reach the target node.

TECHNOLOGIES USED:

The project is bootstrapped with Create React App.

1)**REACT:**

React is a JavaScript library that aims to simplify development of visual interfaces. Developed at Facebook and released to the world in 2013, it drives some of the most widely used code in the world, powering Facebook and Instagram among many, many other software companies.

2)**HTML:**

HTML is a markup language which is used by the browser to manipulate text, images and other content to display it in required format.

3)**CSS:**

CSS. Stands for "Cascading Style Sheet." Cascading style sheets are used to format the layout of Web pages. They can be used to define text styles, table sizes, and other aspects of Web pages that previously could only be defined in a page's HTML.

4)**JAVASCRIPT:**

JavaScript is a programming language that can be included on web pages to make them more interactive. JavaScript is a client side, interpreted, object oriented, high level scripting language

AVAILABLE SCRIPTS:

In the project directory, you can run:

npm start

Runs the app in the development mode.

Open http://localhost:3000 to view it in the browser.

The page will reload if you make edits.

You will also see any lint errors in the console.

npm test

Launches the test runner in the interactive watch mode.

See the section about running tests for more information.

npm run build

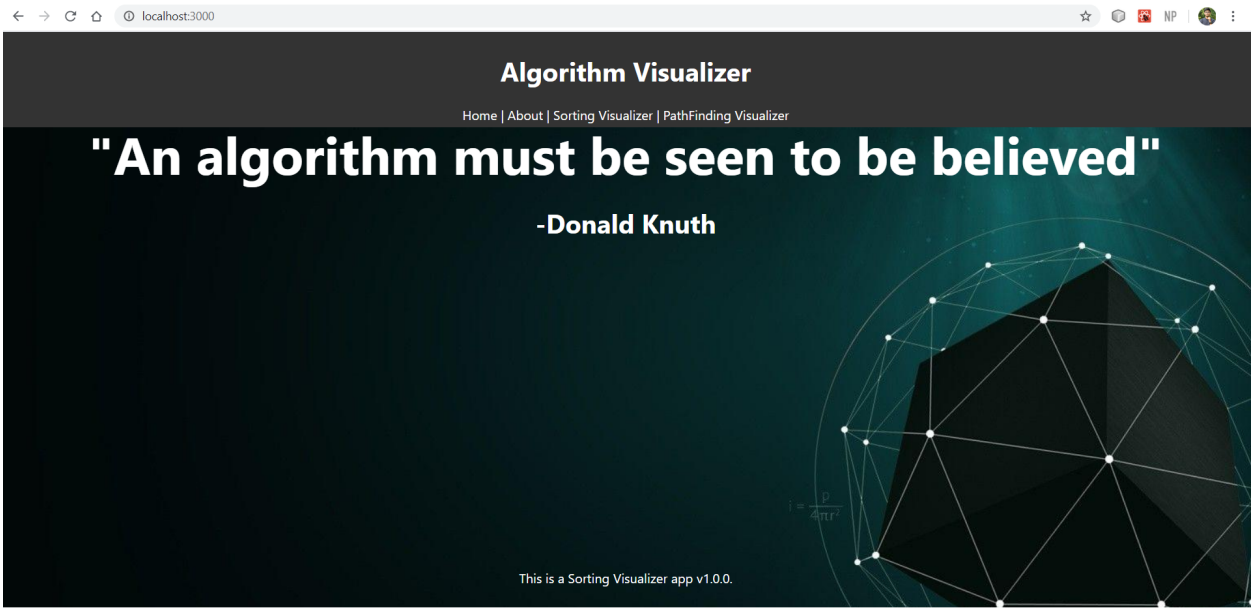
Builds the app for production to the build folder.

It correctly bundles React in production mode and optimizes the build for the best performance.

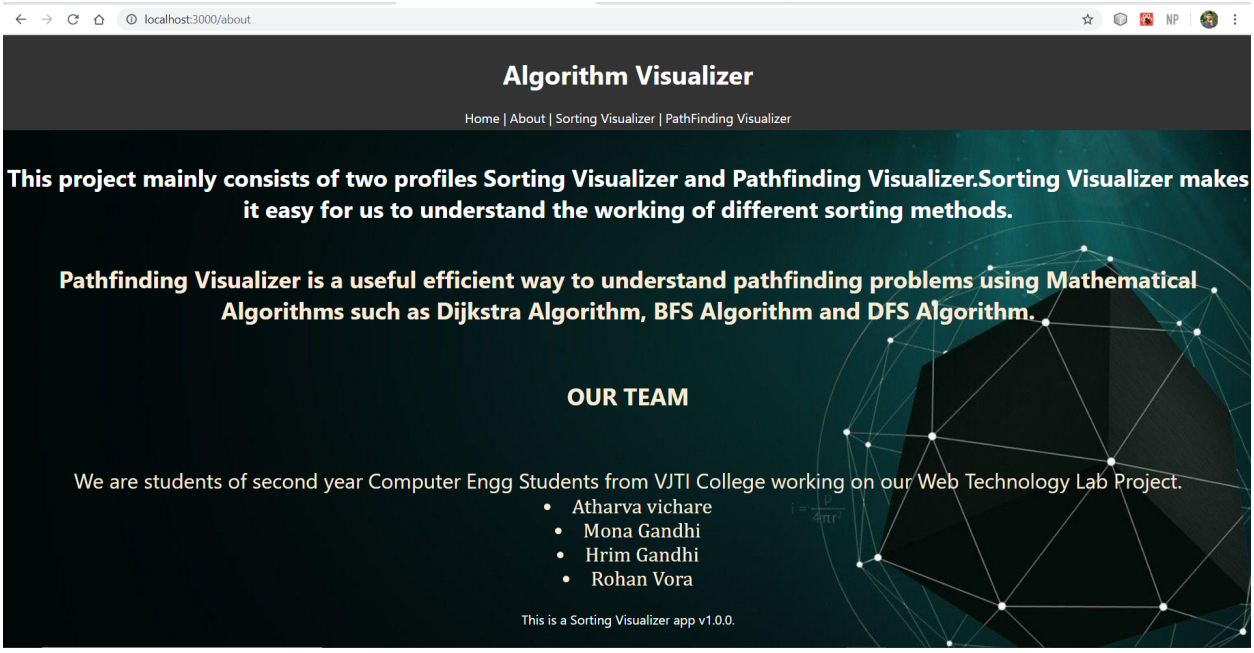
The build is minified and the filenames include the hashes.

SNAPSHOTS OF WEBSITE:

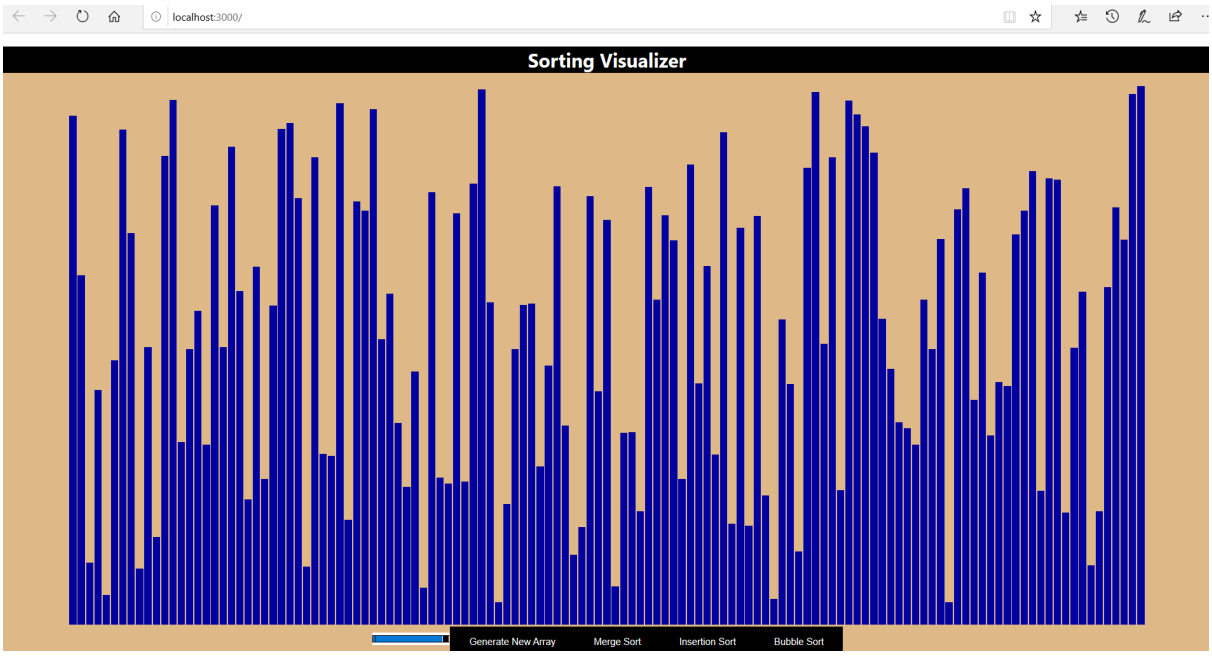
1) Home Page



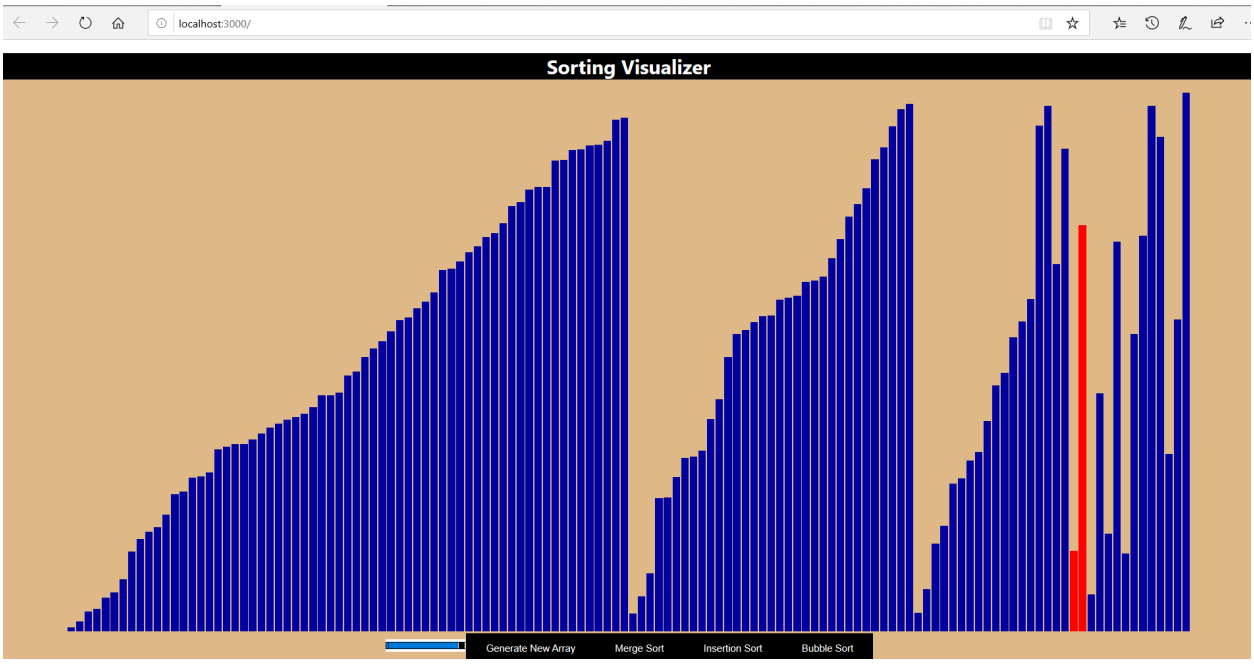
2) About Page



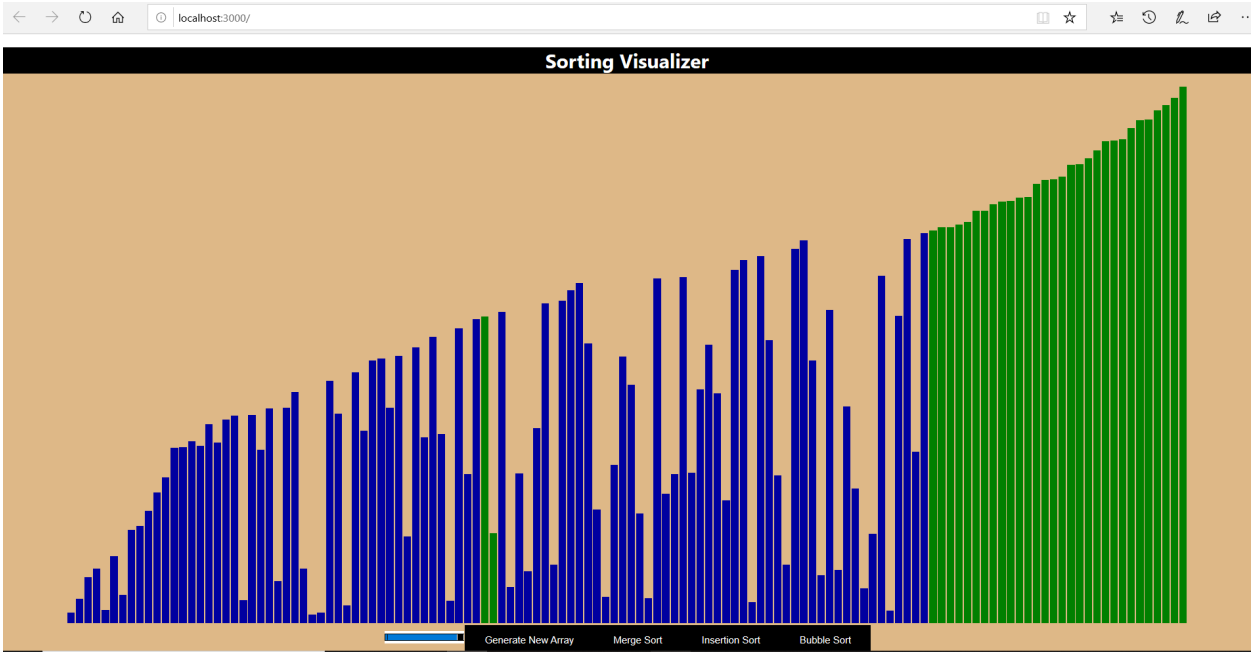
3) Sorting Visualizer



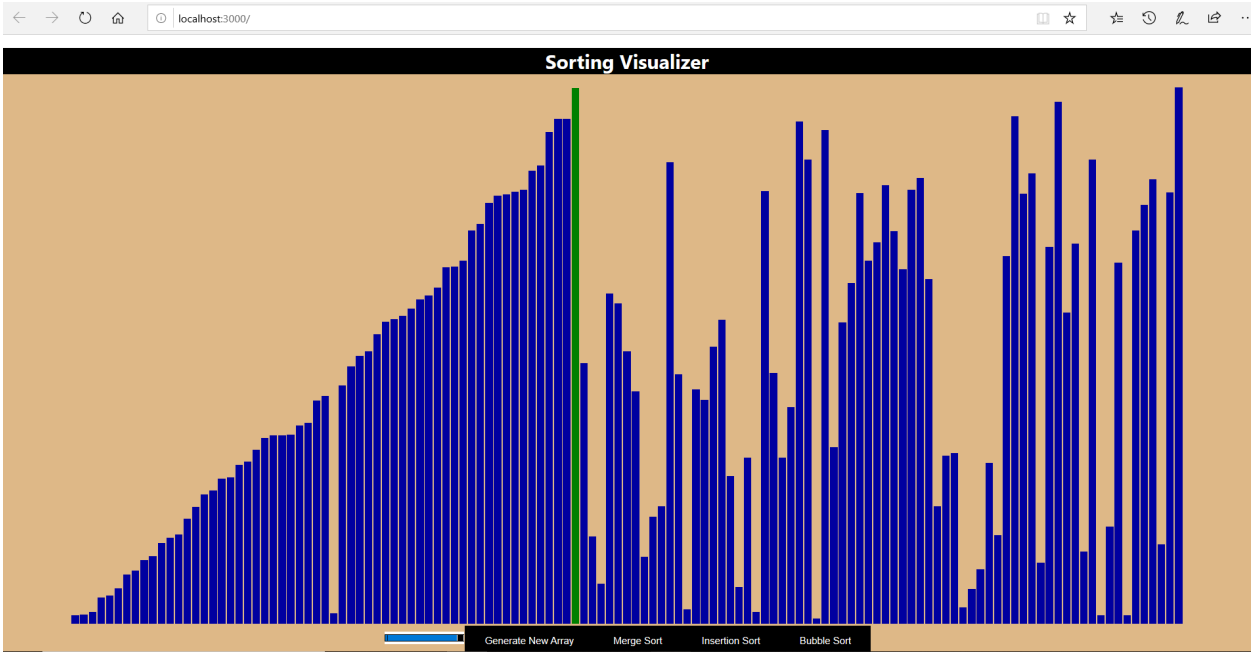
a) Merge Sort



b) Bubble Sort



c) Insertion Sort

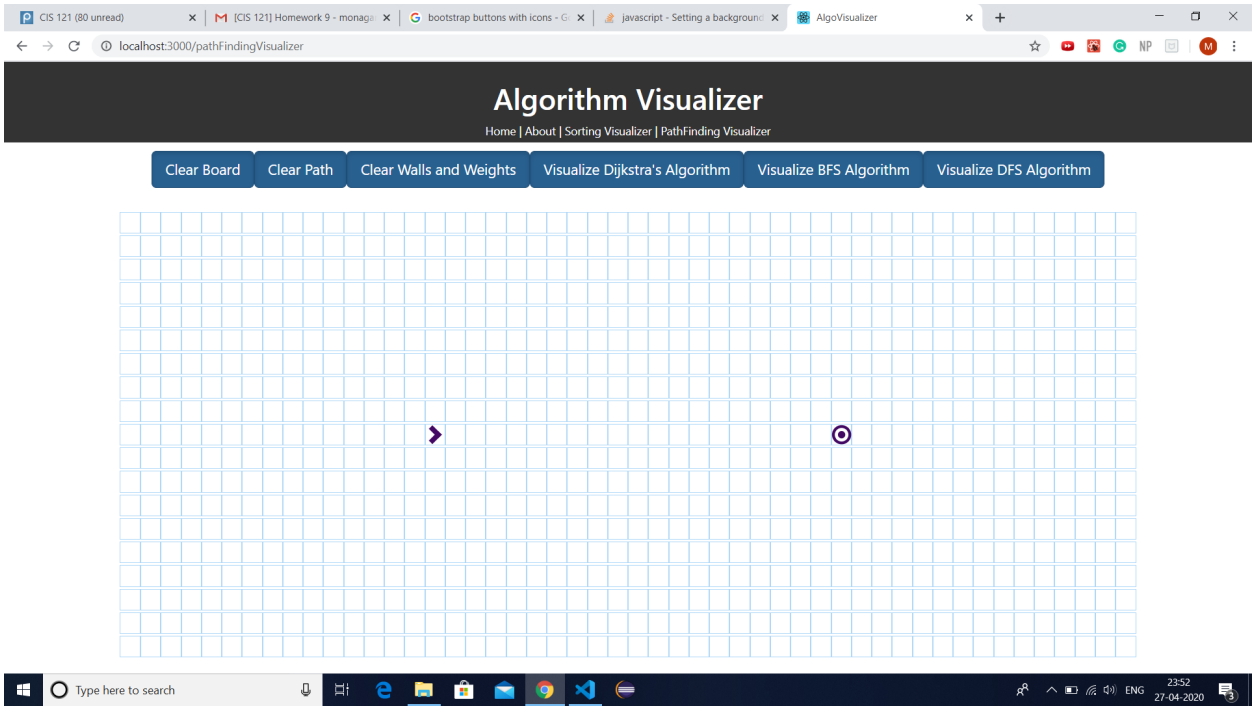


4) PathFinding Visualizer

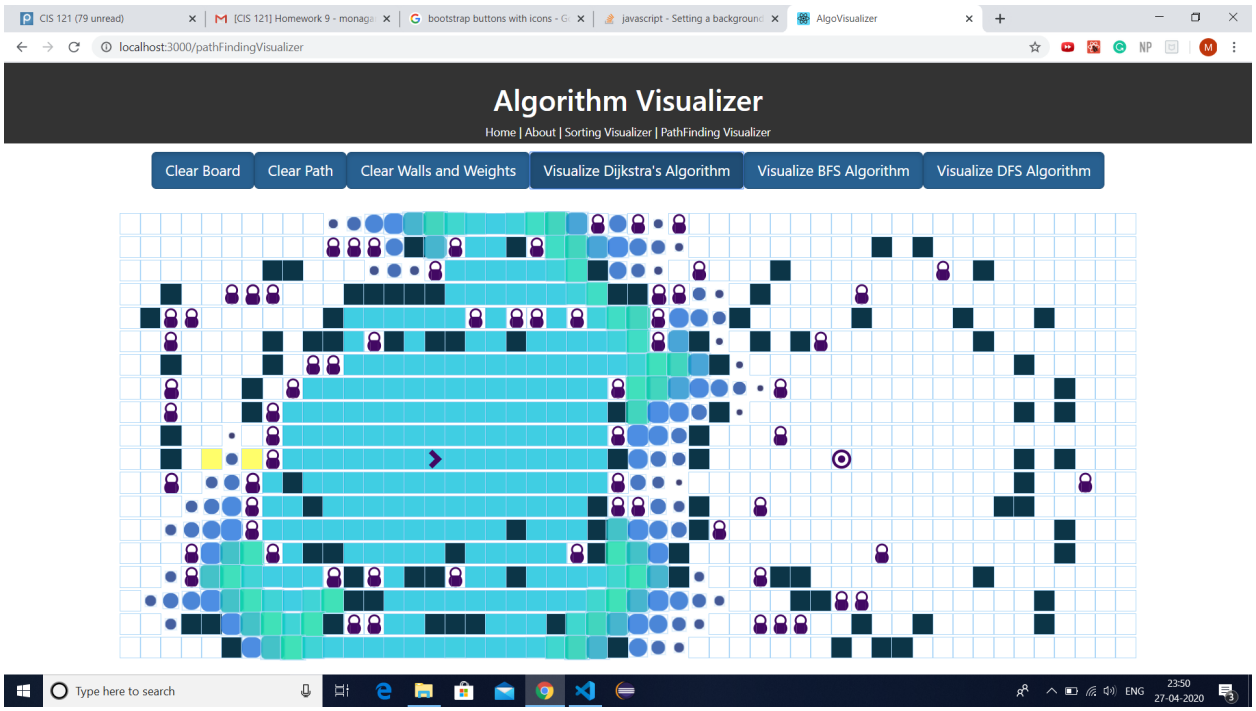
Walls:Walls are impenetrable, meaning that a path cannot cross through them.

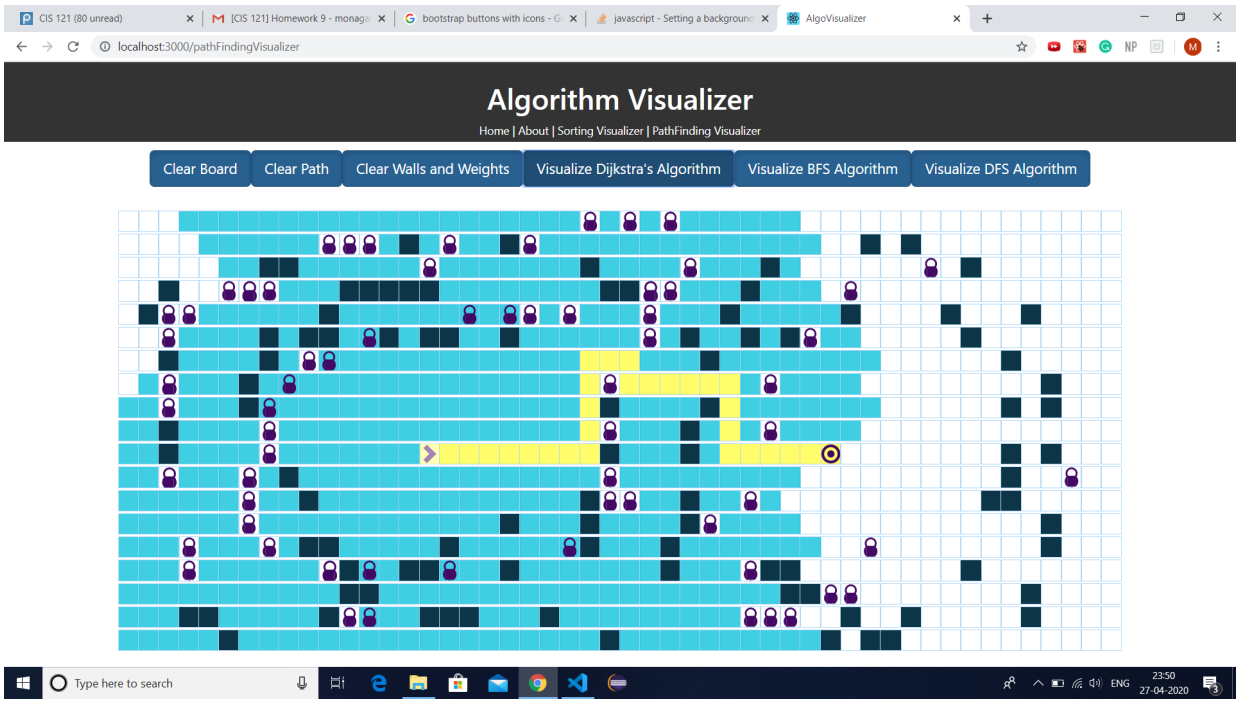
Weights:Weights, however, are not impassable. They are simply more "costly" to move through. In this application, moving through a weight node has a "cost" of 15.

a) On loading the page and after clear board

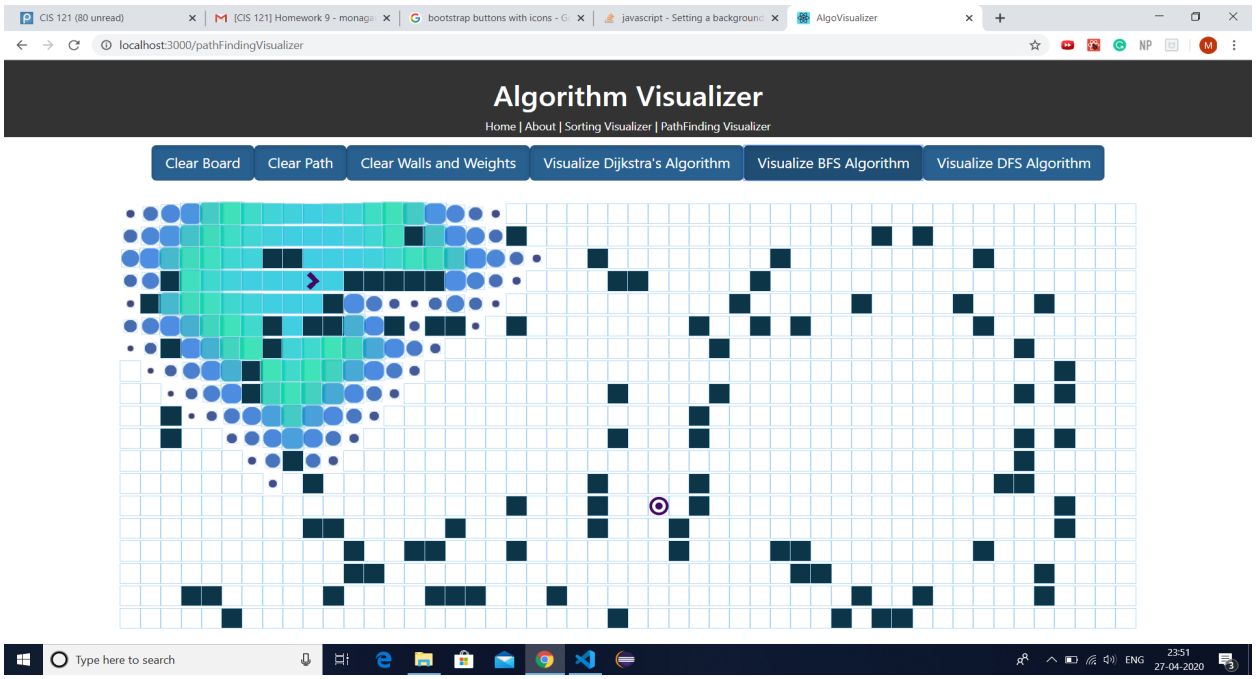


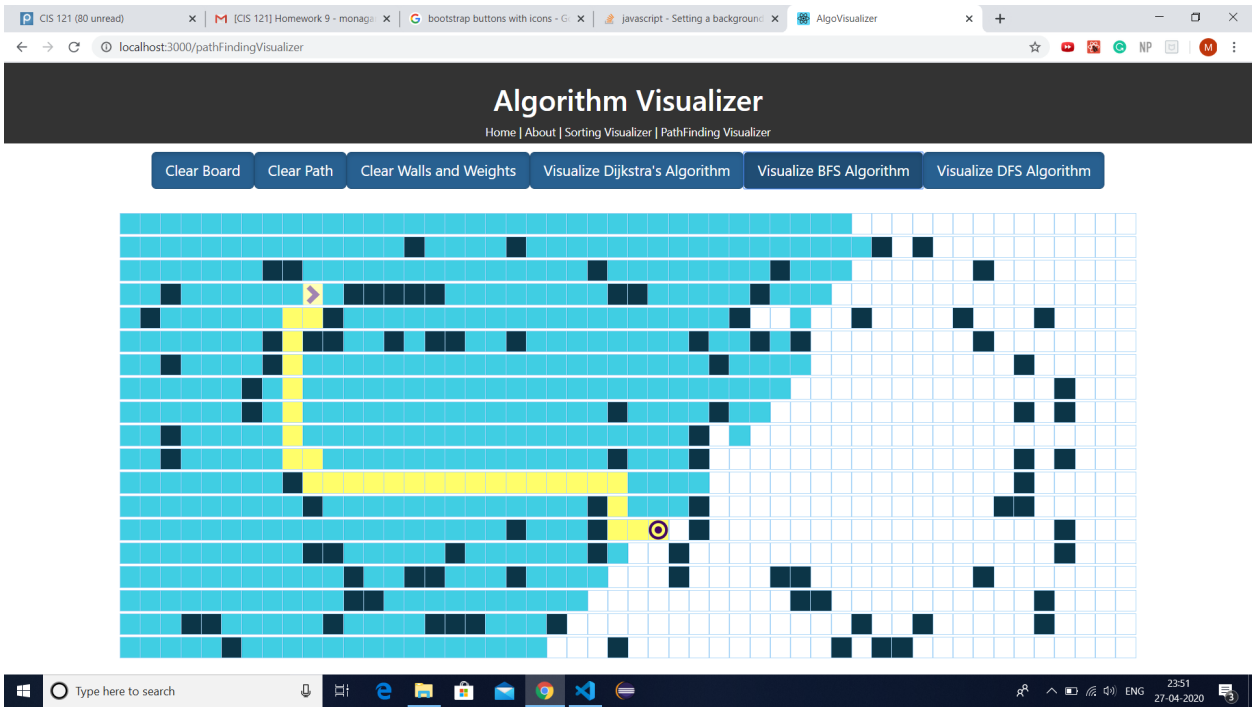
b) Path Finding by Dijkstra's Algorithm



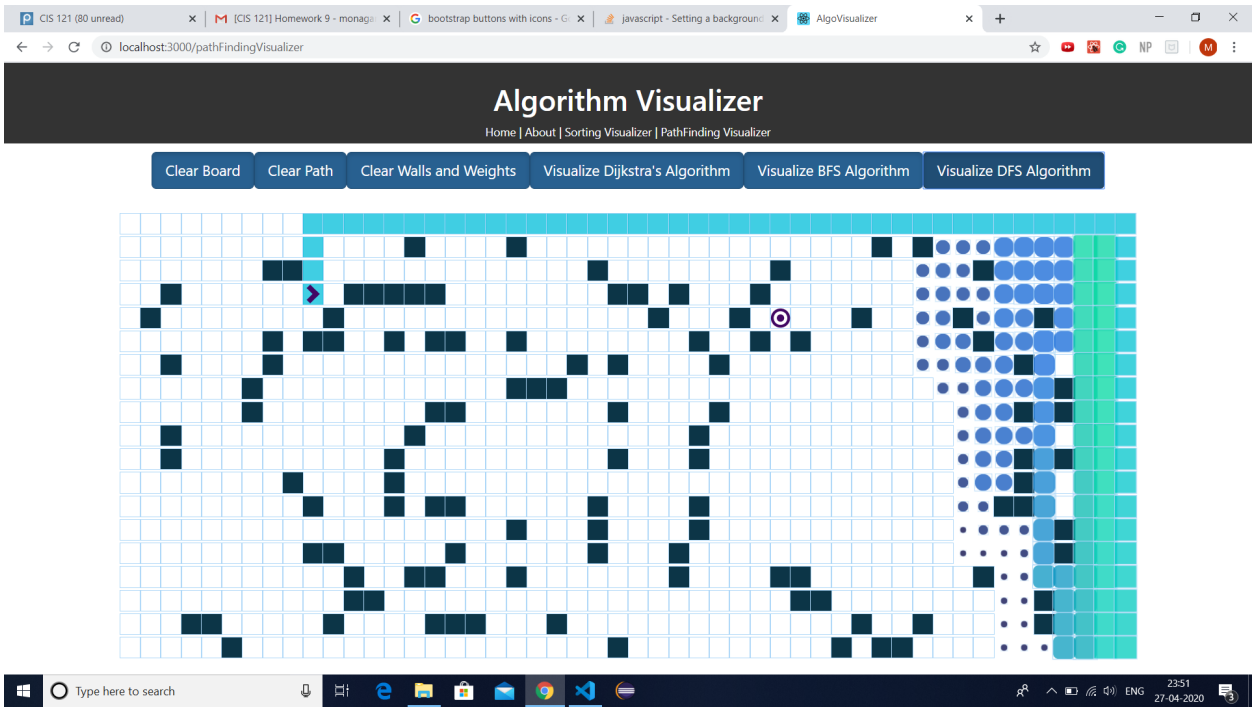


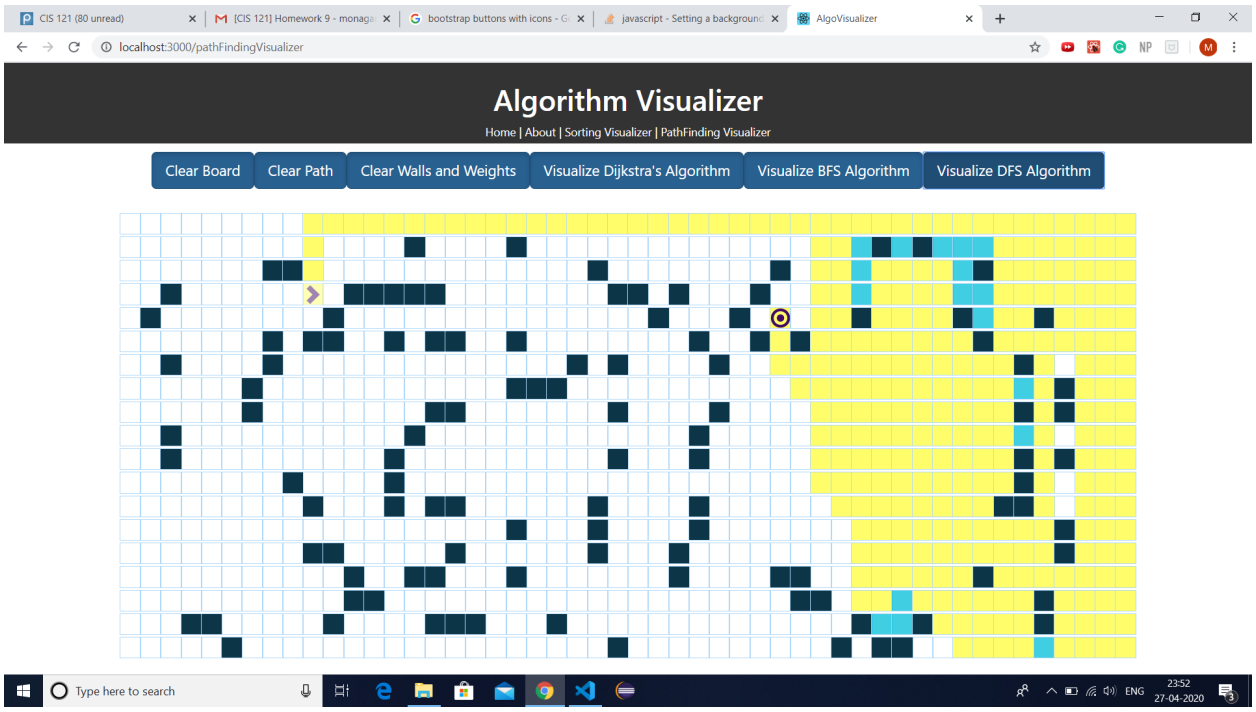
c) Visualizing BFS ALgorithm



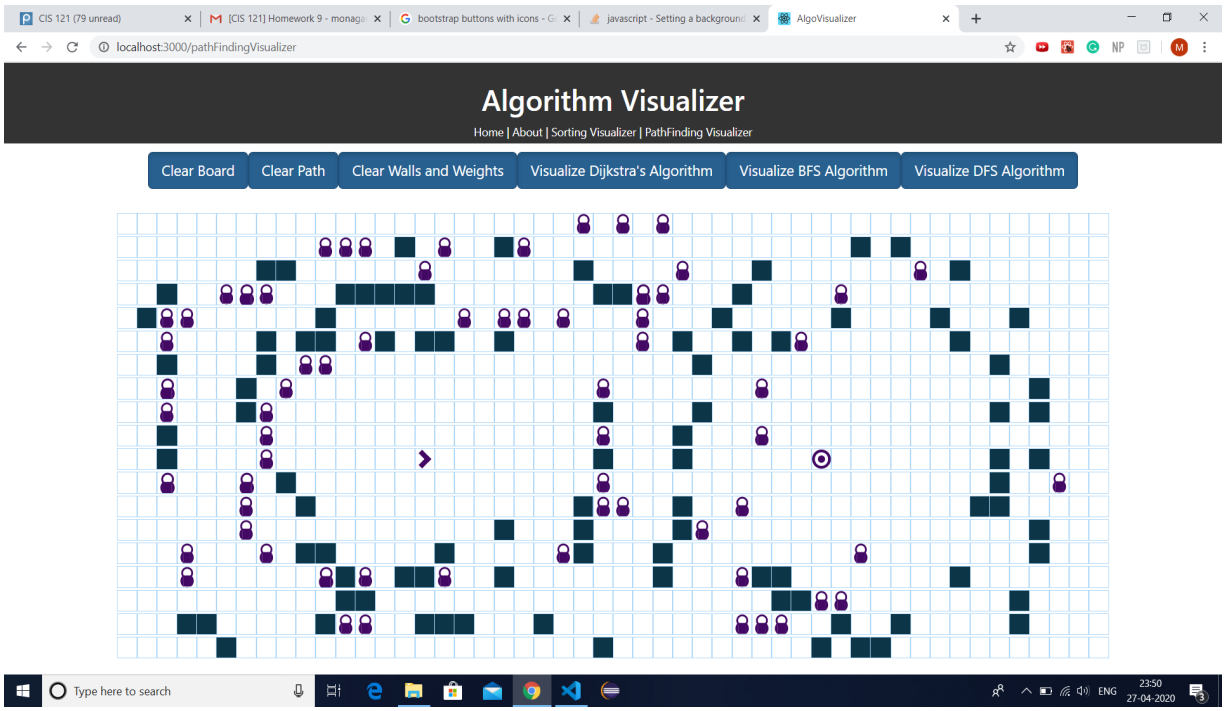


d) Visualizing DFS Algorithm

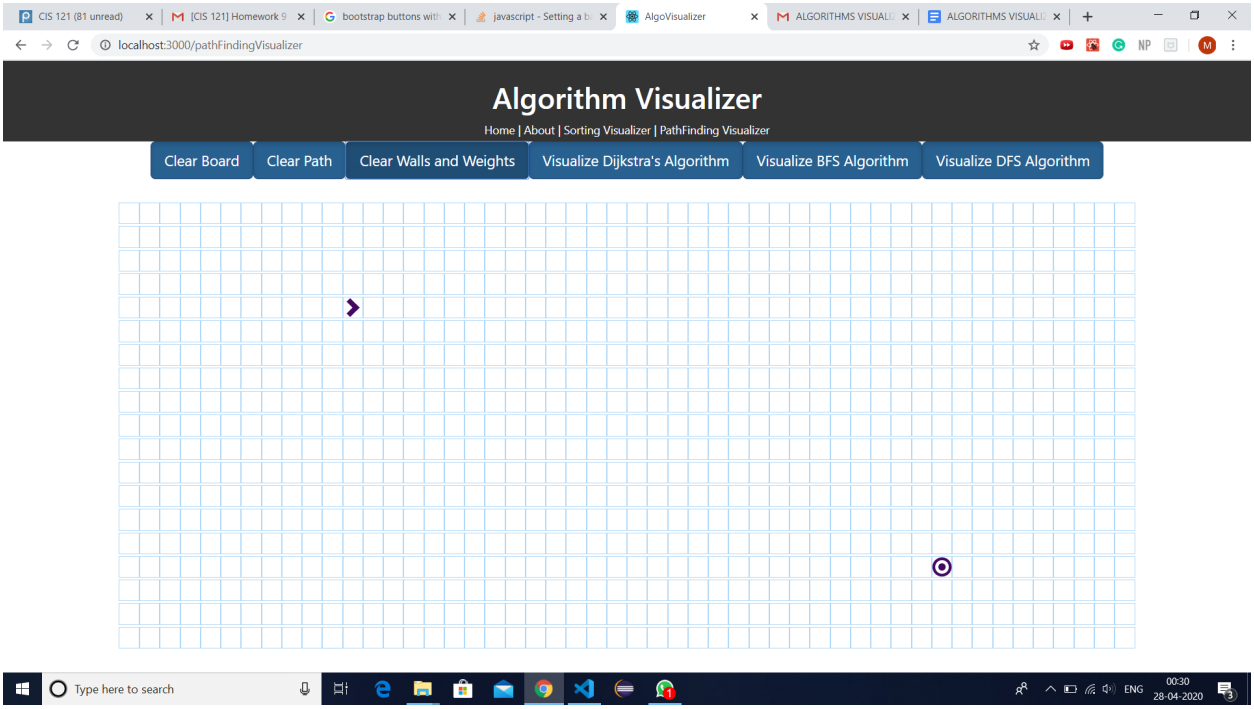




e) On Click ClearPath



f) After Clearing Weights and Walls



FUTURE SCOPE:

1) Improved UI and UX to make it more user friendly.
2) We can further design and construct an app or an application program for the same purpose.
3) We can add more Sorting Algorithms such as HeapSort, SelectionSort,LinearSort and RadixSort.
4) We can add more Path Finding Algorithms like A* -Search,Convergent Swarm Search,Bidirectional Swarm Search,Swarm Algorithm,Greedy best-first Search algorithm.

CONCLUSION:

The Algorithm Visualizer Website has been successfully implemented. It can help learn algorithms by simple visualization.