AMOGH JALAN 2K19/IT/016
amoghjalan_2k19it016@dtu.ac.in

ANIKET GUPTA 2K19/IT/019

aniketgupta_2k19it019@dtu.ac.in

DELHI TECHNOLOGICAL UNIVERSITY

Pathfinding Visualizer

(DS Project Review)

SUBMITTED TO:
Mrs. Swati Sharda

ACKNOWLEGEMENT

We would like to thank Mrs. Swati Sharda Ma'am for providing us with this opportunity to learn and gain practical experience about Discrete Structures. We have learnt a lot by the means of our project, based on Pathfinding Visualizer which would not have been possible without your help and guidance.

-Sincerely

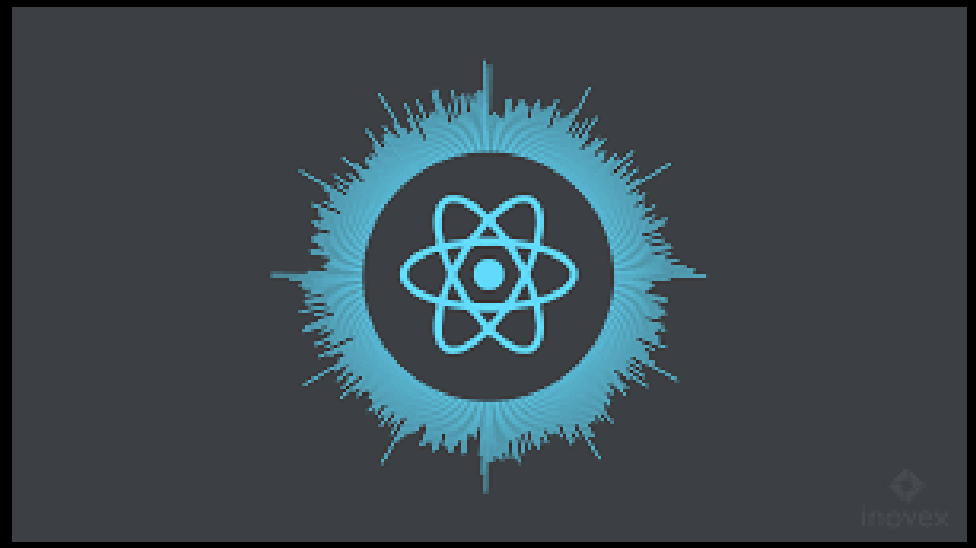
Amogh Jalan & Aniket Gupta

What is Pathfinding Visualizer?

Pathfinding algorithm uses various concepts of graph theory to find path from one node to another , the pathfinding visualizer gives a visual representation to that path.

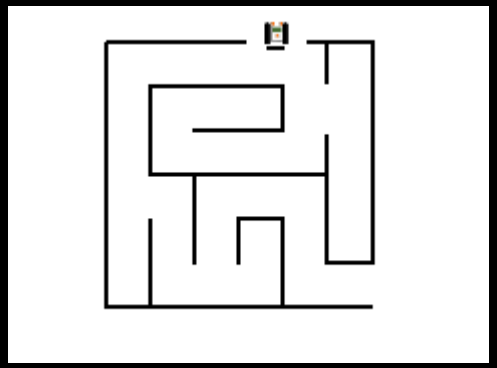
Tech stack used:

=



What are Pathfinding Algorithms?

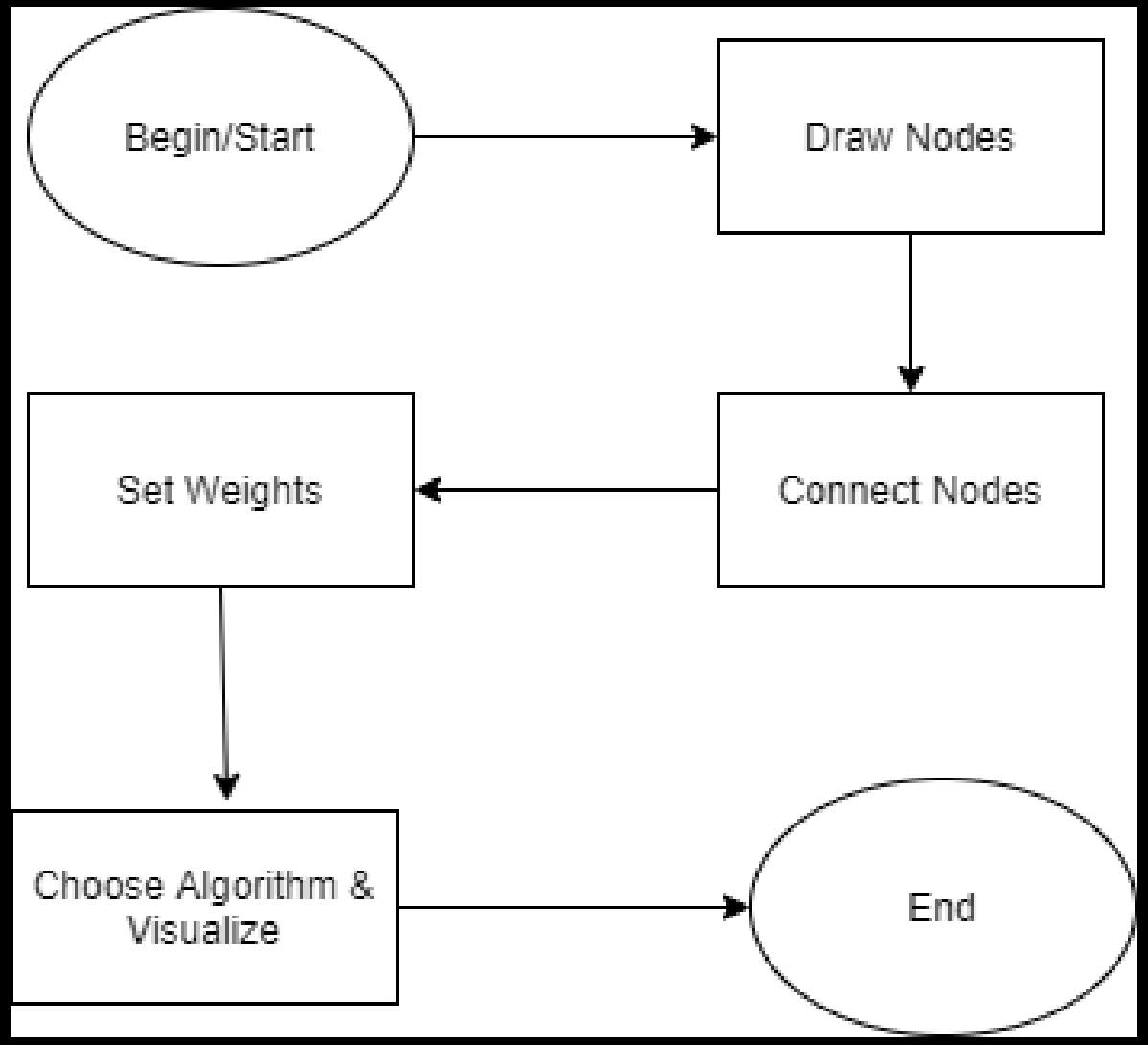
Pathfinding algorithms are usually an attempt to solve the shortest path problem in graph theory. They try to find the best path given a starting point and ending point based on some predefined criteria.



Why are they important?

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.

FLOWCHART


Begin/Start
Draw Nodes
Connect Nodes
Set Weights
Choose Algorithm & Visualize
End

QUEUE Implementation(Used in BFS)

```
export class Queue {
  constructor() {
    this.queue = [];
  }
  push(item) {
    this.queue.push(item);
  }
  front() {
    return !this.isEmpty() ? this.queue[0] : null;
  }
  back() {
    return !this.isEmpty() ? this.queue[this.queue.length - 1] : null;
  }
  pop() {
    if (!this.isEmpty()) {
      this.queue.shift();
    }
  }
  isEmpty() {
    return this.queue.length === 0;
  }
}
```

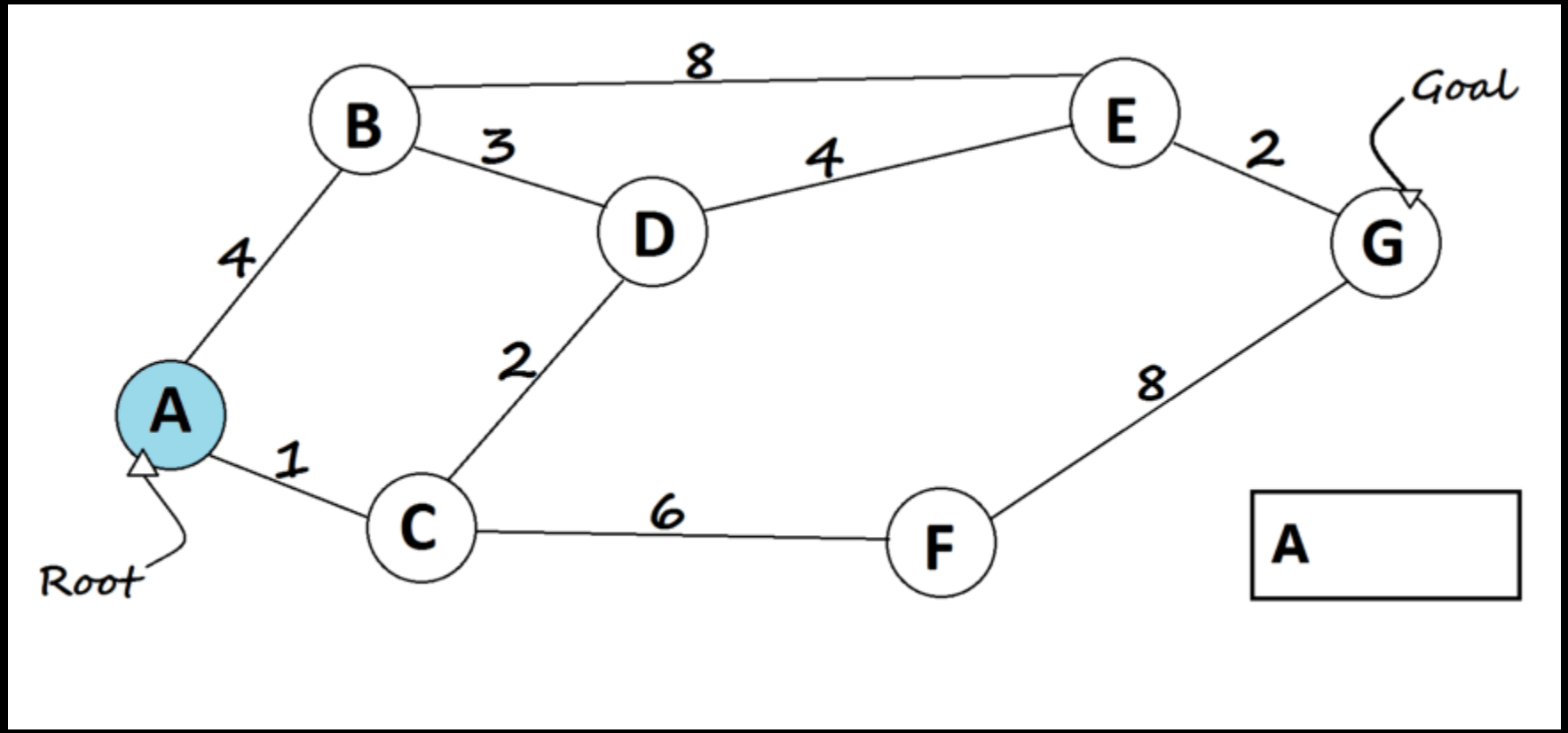
STACK Implementation(Used in DFS)

```
export class Stack {
  constructor() {
    this.stack = [];
  }
  push(item) {
    this.stack.push(item);
  }
  top() {
    return !this.isEmpty() ? this.stack[this.stack.length - 1] : null;
  }
  pop() {
    if (!this.isEmpty()) {
      this.stack.pop();
    }
  }
  isEmpty() {
    return this.stack.length === 0;
  }
}
```

Implemented Algorithms:

Dijkstra

Dijkstra's algorithm tries to find the shortest path from the starting(root) node to every node, hence we can get the shortest path from the starting node to the goal.



```
export const dijkstra = (edges, startNodeId, endNodeId) => {
  const mockEdge = {
    x1: NaN,
    x2: NaN,
    y1: NaN,
    y2: NaN,
    nodeX2: NaN,
    nodeY2: NaN,
    from: "Infinity",
    to: startNodeId.toString(),
    type: "directed",
    weight: NaN,
    isUsedInTraversal: false
  };
  if (startNodeId === endNodeId)
    return { shortestPath: [mockEdge], visitedEdges: [mockEdge] };
  let newEdges = new Map(edges);
  let distance = new Map();
  let prev = new Map();
  let unvisitedSet = new Set();
  let visitedEdges = [];
  distance.set(mockEdge, 0);
  newEdges.forEach((edges, nodeId) => {
    edges?.forEach(edge => {
      distance.set(edge, Infinity);
    });
    unvisitedSet.add(nodeId);
  });
  let currentEdge = mockEdge;
```

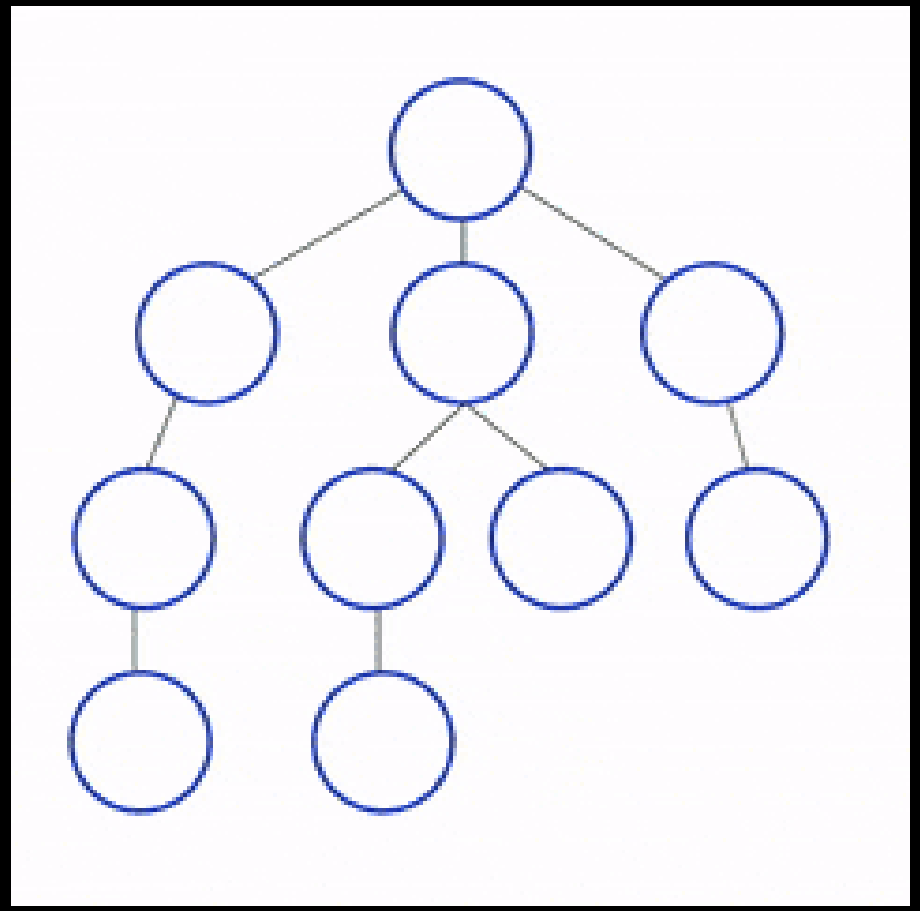
CODE Snippet for Dijkstra

Continued CODE Snippet(Dijkstra)

```
  let currentNodeId = parseInt(currentEdge.to);
  visitedEdges.push(currentEdge);
  unvisitedSet.delete(currentNodeId);
  while (unvisitedSet.size !== 0) {
    getUnvisitedNeighbours(currentEdge, newEdges, distance, unvisitedSet, prev);
    currentEdge = getSmallestUnvisited(distance, unvisitedSet);
    if (currentEdge === undefined || distance.get(currentEdge) === Infinity) {
      return {
        shortestPath: [],
        visitedEdges: visitedEdges
      };
    }
    currentNodeId = parseInt(currentEdge.to);
    visitedEdges.push(currentEdge);
    unvisitedSet.delete(currentNodeId);
    if (currentNodeId === endNodeId) {
      return {
        shortestPath: backtrack(prev, startNodeId, endNodeId),
        visitedEdges: visitedEdges
      };
    }
  }
};
```

Breadth-First Search (BFS)

It starts at the root and explores all of it’s children in the next level(neighbours) before moving to each of the root children, and then, it explores the children of the root children, and so on. The algorithm uses a queue to perform the BFS.



BFS

- A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
- The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

- The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

```
import { Queue } from "../data-structures/Queue";
import { Stack } from "../data-structures/Stack";

export const bfs = (edges, startNodeId) => {
  const bfsQueue = new Queue();
  const visitedEdges = [];
  const mockEdge = {
    x1: NaN,
    x2: NaN,
    y1: NaN,
    y2: NaN,
    nodeX2: NaN,
    nodeY2: NaN,
    from: "Infinity",
    to: startNodeId.toString(),
    type: "directed",
    weight: NaN,
    isUsedInTraversal: false
  };
  const visitedSet = new Set();
  bfsQueue.push(mockEdge);
  let newEdges = new Map(edges);
  while (!bfsQueue.isEmpty()) {
    let lastVisitedEdge = bfsQueue.front();
    let nodeId = parseInt(lastVisitedEdge.to);
    bfsQueue.pop();
    if (!visitedSet.has(nodeId)) {
      visitedEdges.push({
        ...mockEdge,
```

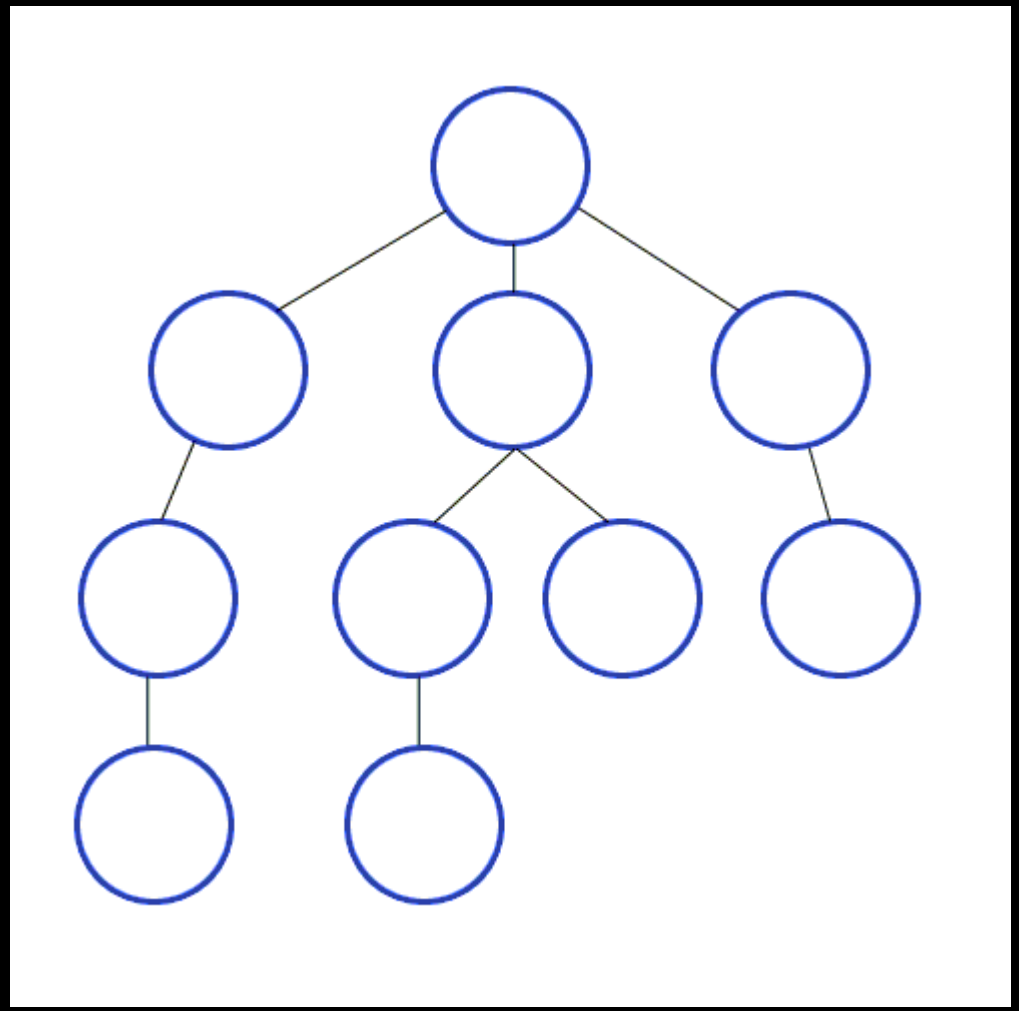
CODE Snippets for BFS

Continued CODE Snippet(BFS)

```
            from: lastVisitedEdge.from,
            to: lastVisitedEdge.to
          });
          const neighbours = findNeighbours(nodeId, newEdges, visitedSet);
          neighbours?.forEach(id => {
            bfsQueue.push({
              ...mockEdge,
              from: nodeId.toString(),
              to: id.toString()
            });
          });
        }
      }
      return visitedEdges;
};
```

Depth-First Search (DFS)

It starts at the root and explores one of it’s children’s sub tree, and then move to the next child’s sub tree, and so on. It uses stack, or recursion to perform the DFS.



DFS

- A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
- The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

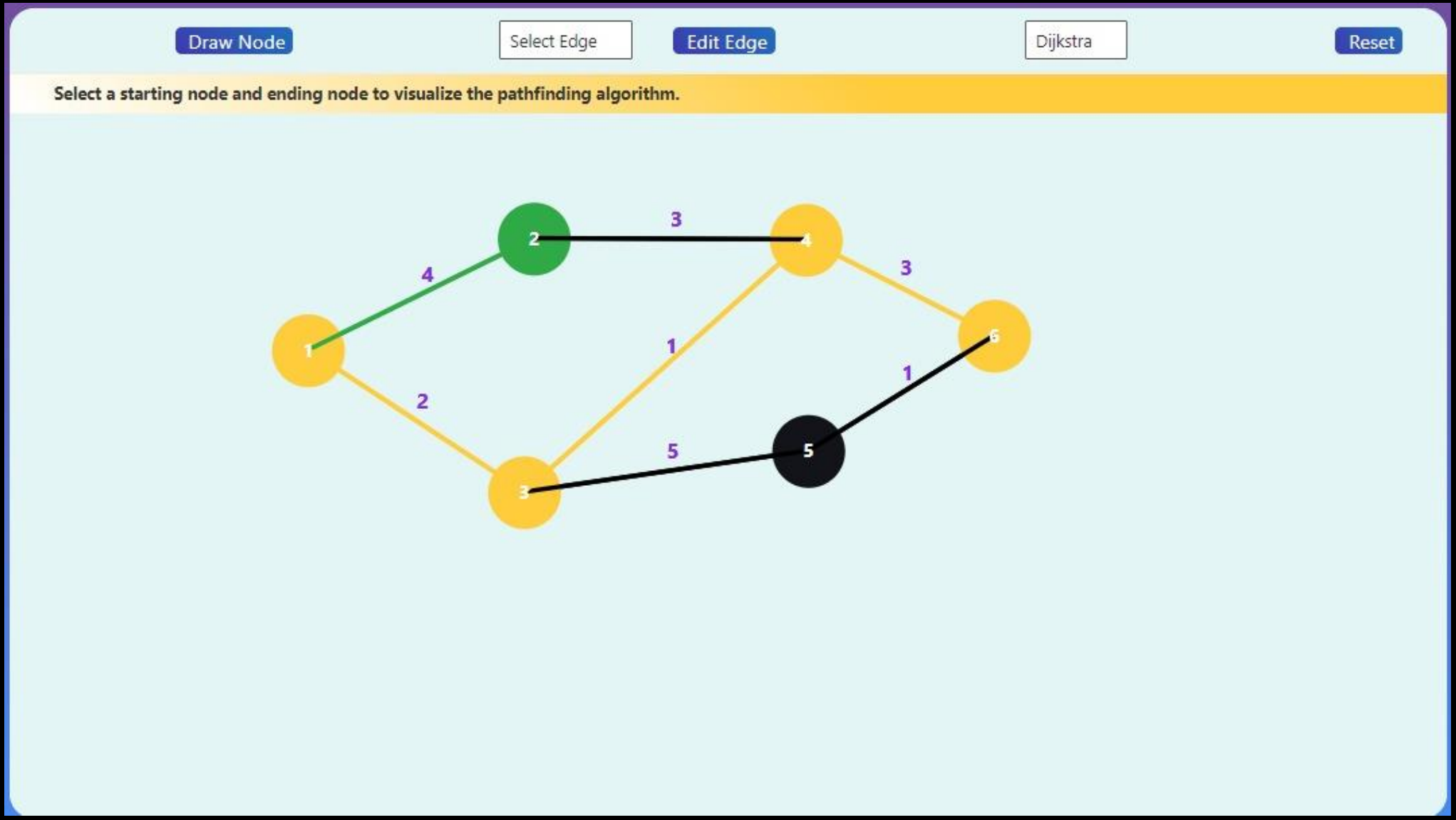
```
export const dfs = (edges, startNodeId) => {
  let dfsStack = new Stack();
  const mockEdge = {
    x1: NaN,
    x2: NaN,
    y1: NaN,
    y2: NaN,
    nodeX2: NaN,
    nodeY2: NaN,
    from: "Infinity",
    to: startNodeId.toString(),
    type: "directed",
    weight: NaN,
    isUsedInTraversal: false
  };
  dfsStack.push(mockEdge);
  const visitedSet = new Set();
  const visitedEdges = [];
  let newEdges = new Map(edges);
  while (!dfsStack.isEmpty()) {
    let lastVisitedEdge = dfsStack.top();
    let nodeId = parseInt(lastVisitedEdge.to);
    dfsStack.pop();
    if (!visitedSet.has(parseInt(lastVisitedEdge.to))) {
      visitedEdges.push({
        ...mockEdge,
        from: lastVisitedEdge.from,
        to: lastVisitedEdge.to
      });
```

CODE Snippet for DFS

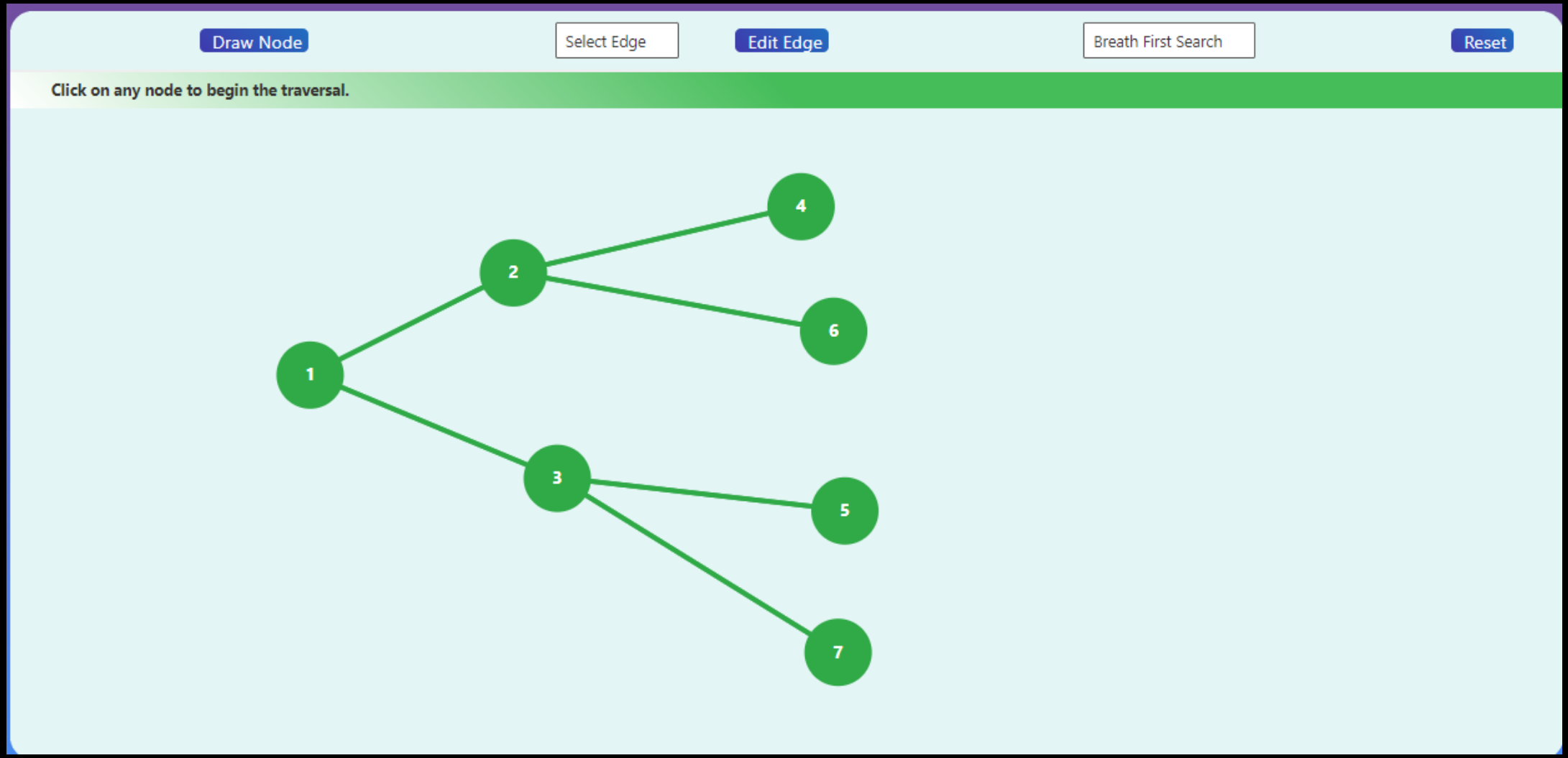
Continued CODE Snippet(DFS)

```
      const neighbours = findNeighbours(nodeId, newEdges, visitedSet);
      neighbours?.forEach(id => {
        dfsStack.push({
          ...mockEdge,
          from: nodeId.toString(),
          to: id.toString()
        });
      });
    }
  }
  return visitedEdges;
};
```

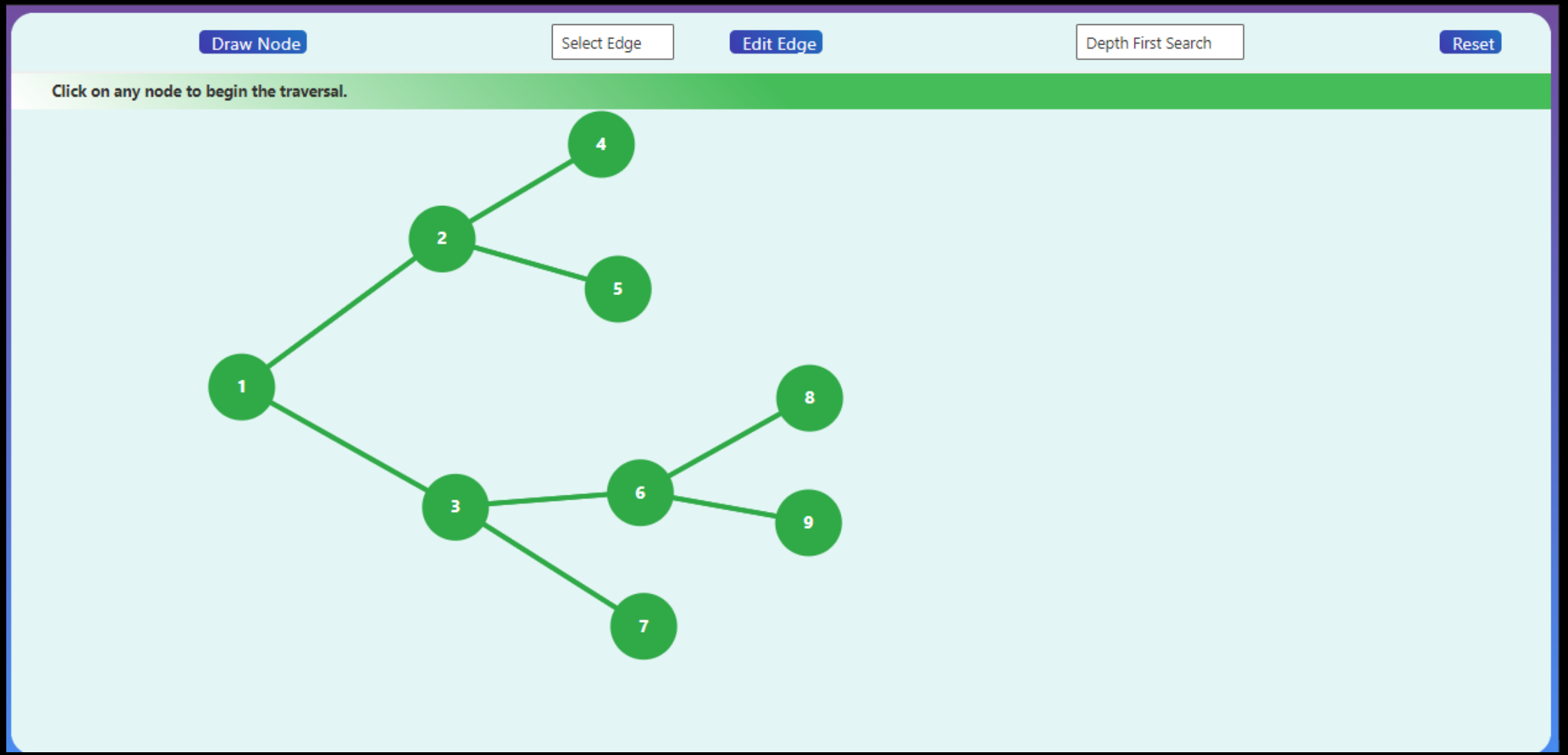
OUTPUT-1(Dijkstra's)



OUTPUT-2(BFS)


Draw Node
Select Edge
Edit Edge
Breath First Search
Reset
Click on any node to begin the traversal.
1
2
3
4
5
6
7

OUTPUT-3(DFS)


Draw Node
Select Edge
Edit Edge
Depth First Search
Reset
Click on any node to begin the traversal.
4
2
5
1
8
3
6
9
7

Thank You!