Graph Algorithms Simulation and Sorting Courses Topologically by Prerequisites

Dr. Shimaa El-Nada

Table of Contents

Team Members:

NO.	Name	Rules
1	Basmala Muhammed Abd El-Hakim	Kruskal’s algorithm simulation
2	Farah Mahmoud Abd El-Moneim	DFS and Prim’s simulation
3	Fayza Mahmoud Bakry	DFS algorithm and BFS simulation
4	Menna Tawfiq Nasr El-Dein	Kruskal’s algorithm and writing the report
5	Sandy Alaa Muhammed	Topological sort implementation
6	Sara Mohamed Abd El-Megid	Prim’s algorithm
7	Sondos Said El-Sayed	BFS algorithm and writing the report

Introduction:

Graph algorithms form the bedrock of modern computing, offering solutions to a myriad of complex problems across various domains. From network analysis to logistics optimization, their versatility and efficiency have propelled them into the forefront of computational science. In this report, we delve into the realm of graph algorithms, focusing specifically on their simulation and their application in sorting courses topologically based on prerequisites.

Purpose of the Project:

The primary aim of this project is to explore the practical implementation of graph algorithms in educational contexts, specifically in the organization and optimization of course structures. By sorting courses topologically according to their prerequisites, educational institutions can streamline curriculum planning, improve student progression, and enhance overall learning outcomes. Understanding the underlying algorithms and simulation techniques involved not only sheds light on the intricacies of educational systems but also provides valuable insights into the broader field of graph theory.

Objectives of the Report:

This report aims to achieve several key objectives:

Explore Graph Algorithms Simulation: Investigate various graph algorithms and their simulation techniques, with a focus on their applicability to educational systems.

Implement Topological Sorting: Develop and implement algorithms for sorting courses topologically based on their prerequisites, considering different scenarios and constraints.

Evaluate Performance and Effectiveness: Assess the performance and effectiveness of the implemented algorithms through experimentation and analysis, considering factors such as runtime efficiency and scalability.

Draw Insights and Conclusions: Draw meaningful insights from the results obtained and discuss their implications for educational institutions and graph algorithm research.

Overview of Topological Sort:

Central to our exploration is the concept of topological sorting, a fundamental algorithmic problem in directed acyclic graphs (DAGs). Topological sorting arranges the vertices of a graph in a linear order such that for every directed edge uv from vertex u to vertex v, u comes before v in the order. In essence, it provides a systematic way to resolve dependencies and establish a meaningful sequence of events or entities. Within the context of educational courses, topological sorting enables institutions to design coherent curriculum structures by ensuring that prerequisite courses are taken before their dependent counterparts.

Code and Analysis:

```
function topology(node) {
    isVisited.set(node, true);
    let neighbors = adj.get(node) || [];
    for (let s of neighbors) {
        if (!isVisited.get(s))
            topology(s);
    }
    arr.push(node);
}
function varifiySort(){
    isVisited = new Map();  ──────────→ O(1)
   ┌for (let [key, value] of adj) {
   │     if (!isVisited.get(key)) {
O(n)┤         topology(key);
   │    }
   │
   └}
}
```

1. Initializing Variables:
    - Initializing isVisited as a new Map takes O(1) time.
2. Iterating through the Adjacency List:
    - Looping through each node in the adjacency list takes O(n) time, where n is the number of nodes.
3. Topology Sorting (DFS):
    - Calling the topology() function for each node involves traversing its neighbors.
    - In the worst case, if each node has all other nodes as neighbors, the time complexity for each call is O(m), where m is the number of neighbors.
4. Overall Time Complexity:
    - Considering the worst-case scenario where each node has all other nodes as neighbors:
        - Looping through each node in the adjacency list takes O(n) time.
        - Calling topology() for each node takes O(m) time.
    - Hence, the overall time complexity is approximately O(n + m).

In the worst-case scenario of a densely connected graph, where every node is linked to every other node, the time complexity can reach O(n^2). Conversely, in a sparse graph where connections between nodes are limited, the time complexity tends to be closer to O(n), indicating a more efficient performance.

Graph Algorithms Simulation:

Our exploration covers several classic algorithms: Depth-First Search (DFS), Breadth-First Search (BFS), Kruskal's algorithm for minimum spanning trees, and Prim's algorithm, another approach for finding minimum spanning trees. These algorithms represent a diverse range of techniques, from traversal to spanning tree construction, each with its unique set of applications and optimizations.

DFS Algorithm:

DFS is a fundamental graph traversal algorithm that systematically explores vertices and edges, effectively traversing through the depths of the graph. Its recursive nature allows for straightforward implementation, making it a popular choice for various tasks such as cycle detection, topological sorting, and finding connected components.

Code and Analysis:

```
// Function to perform DFS traversal
function DFS(node) {
    isVisited[node] = true;
    visitedNodes.push(node); // Store visited node
    for (let eachNode of adj[node]) {
        if (!isVisited[eachNode]) {
            DFS(eachNode);
        }
    }
}
```

- Visiting a Node: Visiting a node takes constant time $O(1)$.
- Traversing an Edge: Traversing an edge from a vertex v to its adjacent vertex takes constant time $O(1)$.
- In the worst-case scenario, each vertex and edge are visited once.
- Visiting all vertices: $O(V)$
- Traversing all edges$O(E)$
- Hence, the overall time complexity of the DFS algorithm is $O(V+E)O(V+E)$.

- Best Case: O(V) - This happens when the graph is disconnected or when DFS has already traversed all vertices. In this scenario, each vertex is visited once, resulting in linear time complexity.
- Worst Case: O(V + E) - Even in densely connected graphs, where each edge is visited once, DFS still maintains a linear time complexity by traversing each vertex and edge once.

In essence, DFS has a time complexity of O(V + E) and an auxiliary space complexity of O(V).

BFS Algorithm:

In contrast to DFS, BFS traverses a graph level by level, exploring neighbors of a vertex before moving to its neighbors' neighbors. This characteristic lends itself well to finding shortest paths in unweighted graphs, determining connectivity, and solving puzzles like finding the shortest path in a maze.

Code and Analysis:

```
Bfs : class Graph {
    constructor() {
     // Initialize an empty adjacency list to store vertices and their neighbors
        this.adjacencyList = {};
    }
    addVertex(vertex) {
     // If the vertex does not already exist in the adjacency list, add it with
an empty array as its value
        if (!this.adjacencyList[vertex]) this.adjacencyList[vertex] = [];
    }
    addEdge(v1, v2) {
        // Add v2 to the adjacency list of v1 and vice versa
        this.adjacencyList[v1].push(v2);
        this.adjacencyList[v2].push(v1);
    }
    bfs(start) {

       const queue = [start];  ──────────────→ O(1)
        //  array to store the BFS traversal result
        const result = [];
        // to keep track of visited vertices
        const visited = {};
        visited[start] = true;
   ┌while (queue.length) {
   │
   │        let currentVertex = queue.shift(); ────────→ O(1) ┐
C+E0│                                                          ├ C
   │        result.push(currentVertex); ────────→ O(1)        ┘
   │        this.adjacencyList[currentVertex].forEach(neighbor => {
C+E1│ ←         if (!visited[neighbor]) {
   │                visited[neighbor] = true;
C+E2│               queue.push(neighbor);
   │            }
CV+ ∑E│     });
   └    }
    return result;
```

Time : O(V+E)   Space : O(V+E)

Kruskal’s algorithm:

Kruskal's algorithm addresses the problem of finding a minimum spanning tree in a weighted graph. By iteratively selecting edges in ascending order of weight while avoiding cycles, Kruskal's algorithm constructs a minimum spanning tree efficiently, making it a cornerstone in network design, clustering, and optimization problems.

Code and Analysis:

```
function makeSet(parent, rank, n) {
    for (let i = 0; i < n; i++) {
        parent[i] = i; rank[i] = 0; }
}
function findParent(parent, component) {
    if (parent[component] === component)
        return component;
    return parent[component] = findParent(parent, parent[component]);
}
function unionSet(u, v, parent, rank) {
    u = findParent(parent, u);
    v = findParent(parent, v);
    if (rank[u] < rank[v]) {
        parent[u] = v;
    } else if (rank[u] > rank[v]) {
        parent[v] = u;
    } else {
        parent[v] = u; rank[u]++;
    }
}
function kruskalAlgo(n, edge) {
    edge.sort((a, b) => {
        return a[2] - b[2];
    });
    let parent = new Array(n);
    let rank = new Array(n);
    makeSet(parent, rank, n);
    let minCost = 0; let mstEdges = [];
    for (let i = 0; i < edge.length; i++) {
        let [u, v, wt] = edge[i];
        let parentU = findParent(parent, u);
        let parentV = findParent(parent, v);
        if (parentU !== parentV) {
            unionSet(u, v, parent, rank);
            minCost += wt;
            mstEdges.push([u, v, wt]);
        }
    }
return mstEdges;
}
```

1. makeSet(parent, rank, n):
    - O(n) time complexity because it iterates through the arrays once.
2. findParent(parent, component):
    - In the worst-case scenario, it traverses through the parent array until it reaches the root of the tree, taking up to O(n) time.
3. unionSet(u, v, parent, rank):
    - In the worst-case scenario, it may take up to O(n) time due to the findParent operations.
4. kruskalAlgo(n, edge):
- Sorting the edges takes O(E log E) time, where E is the number of edges.
- Initialization and edge processing are both O(E) operations.
- Thus, the overall time complexity of Kruskal's algorithm is dominated by the sorting step, giving O(E log E).

Auxiliary Space: O (V + E), where V is the total number of edges in the graph and E is the total number of vertices

Prim’s Algorithm:

Similar to Kruskal's algorithm, Prim's algorithm also tackles the minimum spanning tree problem. However, it takes a different approach by growing the tree from an arbitrary starting vertex, iteratively adding the shortest edge that connects the growing tree to an unvisited vertex. This greedy strategy ensures the construction of a minimum spanning tree and finds applications in network design, routing, and clustering.

Code and Analysis:

```
class MinHeap {
    constructor() { this.heap = []; }
    insert(vertex,key){
        this.heap.push({ vertex, key });
        this.bubbleUp(this.heap.length - 1);        O(VlogV)
    }
    extractMin(){
        const min = this.heap[0];
        const last = this.heap.pop();
        if (this.heap.length > 0) {                 O(logV)
            this.heap[0] = last;
            this.bubbleDown(0); }
        return min.vertex;
    }
    bubbleUp(index) {}  bubbleDown(index) {}
    isEmpty() {return this.heap.length === 0;}
  }
  function primMST(graph) {
    const V = graph.length;
    const inMST = new Array(V).fill(false);         O(1)
    const key = new Array(V).fill(Infinity);
    const parent = new Array(V).fill(-1);
    key[0] = 0;
    const minHeap = new MinHeap();
    minHeap.insert(0, 0);
    while (!minHeap.isEmpty()) {
        const u = minHeap.extractMin();
        inMST[u]=true;
      for (const [v, weight] of graph[u]) {         O(ElogV)
            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                parent[v] = u;
                minHeap.insert(v, key[v]); } }
    }
const result = [];
for (let i = 1; i < V;i++)
    return result;
```

- Best Case: O(E log V) - When the graph is already an MST or consists of disconnected components.
- Average Case: O((V + E) log V) - Randomly distributed edges.
- Worst Case: O((V + E) log V) - Densely connected graph.
- Here, V represents the number of vertices, and E represents the number of edges.
- The auxiliary space complexity of Prim’s algorithm is O(V + E) for the priority queue used to store vertices and their key values.