DATA STRUCTURE & ALGORITHMS-II

LAB MANUAL

Subject Teachers:

Mr. Anup Arvind Dange

Dr. Pushpi Rani

Data Structures and Algorithms (II) (23UCOPCP2407)		
Teaching Scheme:	**Credit**	**Examination Scheme**
Practical: 02 Hrs./Week	**01**	**Ext.: 25 Marks**

Course Outcomes: After completing this course, students will be able to

CO1: Identify file handling methods and apply them to optimize data access and retrieval.

CO2: Describe advanced data structure such as AVL trees, B-trees, Heaps and their applications to solve real-world problems.

CO3: Apply searching and sorting algorithms on tree data structure.

CO4: Implement and analyze the efficiency of graph algorithms for different graph representations.

Course Outcomes	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
CO1	3	3	3	2	3	--	--	1	--	--	--	--	2	3
CO2	2	2	3	3	3	--	--	2	--	--	--	--	2	2
CO3	2	3	2	2	2	--	--	--	--	--	--	--	2	1
CO4	3	3	3	2	3	--	--	--	--	--	--	--	2	2

Sr. No.	List of Laboratory Assignments (Any 8)
1	Write a sorting program with the following features: a) Reads data from a text file and sorts it alphabetically by default. b) If the file has data in rows and columns (separated by space or tab) then allows sorting on a particular column. c) implement any sort using numeric or alphabetical ordering and compare stable and unstable sort (VLAB: https://ds2-iiith.vlabs.ac.in/exp/radix-sort/analysis/stability-of-selection-sort.html).
2	Develop C program to build a) MAX_HEAP b) MIN_HEAP c) Implement heap sort using MAX_HEAP (VLAB: https://ds1-iiith.vlabs.ac.in/exp/heap-sort/heap-sort/index.html)
3	Construct binary tree by inserting the values in the order given. After constructing a binary tree perform following traversing using recursive and non-recursive functions. a) Preorder, b) Postorder c) Inorder
4	Implement AVL trees with following operations a) Create b) Insert c) Search d) Delete
5	Write a program to perform the following operations: a) Insertion into a B-tree b) Searching in a B-tree
6	Develop C function to implement graph using a) Adjacency lists b) Adjacency matrices c) Demonstrate Djkstra's algorithm on a weighted graph (VLAB: https://ds2-iiith.vlabs.ac.in/exp/dijkstra-algorithm/dijkstras-algorithm/index.html)

7	The given directed graph contains *N* nodes and *M* edges. Each edge connects two nodes, Write a program to find the sequence of nodes in the order they are visited using a) DFS b) BFS c) Graph traversal concepts understanding (VLAB: https://ds1-iiith.vlabs.ac.in/exp/depth-first-search/graph-traversals.html).
8	Write a program to read a map stored in a text file with (city1, city2, distance) comma separated values. a) Build a graph using this data. b) Print all pair shortest paths information between all pairs of cities (using Warshall's Algorithm).
9	Write a program to read a map stored in a text file with (city1, city2, distance) comma separated values. Create Minimum Spanning Tree for this graph using a) Prim's Algorithm b) Kruskal's Algorithm c) Compare Kruskal's and Prim's with other algorithms to find MST (VLAB : https://ds2-iiith.vlabs.ac.in/exp/min-spanning-trees/index.html)
10	Given a directed acyclic graph of numbers, generate a topological ordering of numbers using a) DFS method b) Kahn's algorithm c) Analyze the time complexity of both methods by counting the number of operations involved in each iteration (VLAB: https://ds2-iiith.vlabs.ac.in/exp/topo-sort/analysis/time-and-space-complexity.html)
	Content beyond Syllabus
11	Implement Djkstra's algorithm using Fibonacci Heap.
12	Write a program to implement Splay trees.

Experiment No.: 08 [a]

Aim : Write a program to read a map stored in a text file with (city1, city2, distance) comma separated values.
A)Build a graph using this data.

Objective:

To implement a program that reads a map stored in a text file with comma-separated values in the format (city1, city2, distance) and builds a graph using this data.

Apparatus / Software Requirements:

- C++ Compiler (GCC / Code::Blocks / Dev C++)
- Text Editor (Notepad++ / VS Code)
- Input File: map.txt

Theory:

Graph:

A **graph** is a non-linear data structure consisting of nodes (vertices) and edges (connections). In this experiment:

- **Vertices** represent cities.
- **Edges** represent distances between cities.
- The graph is represented using an **adjacency list** or **adjacency matrix**.

Types of Graph Representation:

1. **Adjacency Matrix** – 2D matrix used for dense graphs.
2. **Adjacency List** – Map of vertices to a list of neighboring vertices and weights, efficient for sparse graphs.

We'll use an **adjacency list** here for flexibility and memory efficiency.

Input File Format (map.txt):

Each line contains:

city1,city2,distance

Example:

A,B,4
A,C,2
B,C,1
C,D,7

C++ Source Code:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <map>
#include <vector>
#include <string>

using namespace std;

// Adjacency list: city -> vector of (connected city, distance)
map<string, vector<pair<string, int>>> graph;

void readMapFromFile(const string& filename) {
    ifstream infile(filename);
    string line;

    while (getline(infile, line)) {
        stringstream ss(line);
        string city1, city2, distanceStr;
        getline(ss, city1, ',');
        getline(ss, city2, ',');
        getline(ss, distanceStr);
        int distance = stoi(distanceStr);

        graph[city1].push_back({city2, distance});
        graph[city2].push_back({city1, distance}); // For undirected graph
    }
}

void displayGraph() {
    cout << "\nGraph (Adjacency List):\n";
    for (auto& entry : graph) {
        cout << entry.first << " -> ";
        for (auto& neighbor : entry.second) {
            cout << "(" << neighbor.first << ", " << neighbor.second << ") ";
        }
        cout << endl;
    }
}
```

```
int main() {
   string filename = "map.txt";
   readMapFromFile(filename);
   displayGraph();
   return 0;
}
```

Steps to Execute:

1. Create a file map.txt with appropriate city-distance data.
2. Copy the code into a .cpp file (e.g., graph_builder.cpp).
3. Compile and run the program.
4. Observe the adjacency list output.

Sample Output:

Given the file:

A,B,4
A,C,2
B,C,1
C,D,7

Output:

Graph (Adjacency List):
A -> (B, 4) (C, 2)
B -> (A, 4) (C, 1)
C -> (A, 2) (B, 1) (D, 7)
D -> (C, 7)

Result:

The program successfully:

- Read city and distance data from a file.
- Built a graph using an adjacency list.
- Displayed the structure of the graph clearly.

Conclusion:

This experiment demonstrates how to build a graph from real-world map data using adjacency lists. It helps students understand file handling, data parsing, and dynamic data

structure usage in C++.

Viva Questions:

1. What is an adjacency list and how is it different from an adjacency matrix?
2. How would you store a directed graph?
3. What data structures are used in this program?
4. How can you modify this program to support weighted directed graphs?
5. What is the time complexity to insert and access neighbors in this graph?

Experiment No.:08 [b]

Aim: Write a program to read a map stored in a text file with (city1, city2, distance) comma separated values.

Reading a Map from File and Computing All-Pairs Shortest Paths using Warshall's (Floyd-Warshall) Algorithm

---

Objective:

To write a C++ program that reads a map stored in a file using the format (city1, city2, distance) and prints the **shortest distance between all pairs of cities** using **Warshall's Algorithm** (Floyd-Warshall).

Software/Hardware Requirements:

- C++ Compiler (GCC / Code::Blocks / Dev-C++ / VS Code)
- Text Editor
- Operating System: Windows/Linux
- Input File: map.txt

Theory:

- A **graph** is a structure with **vertices (cities)** and **edges (roads with distances)**.
- The **Floyd-Warshall algorithm** is used to find the **shortest paths between all pairs of nodes** in a weighted graph.
- It uses **Dynamic Programming** with a time complexity of **$O(V^3)$**.

Input File Format (map.txt):

Each line should be:

City1,City2,Distance

Sample Input:

A,B,4
A,C,2
B,C,1
B,D,5
C,D,8
D,E,6
C,E,10

💻 **C++ Source Code:**

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>
#include <climits>

using namespace std;

const int INF = INT_MAX;

// Floyd-Warshall Algorithm
void floydWarshall(vector<vector<int>>& dist, int V) {
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF &&
                    dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
}

// Print distance matrix
void printShortestPaths(const vector<vector<int>>& dist, const vector<string>& cities) {
    int V = cities.size();
    cout << "\nAll-Pairs Shortest Path Distances:\n";
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            cout << cities[i] << " -> " << cities[j] << ": ";
            if (dist[i][j] == INF)
                cout << "INF";
            else
                cout << dist[i][j];
            cout << endl;
        }
        cout << endl;
    }
}

int main() {
    ifstream file("map.txt");
    string line, city1, city2;
    int distance;

    map<string, int> cityToIndex;
```

```
    vector<string> indexToCity;
    vector<tuple<string, string, int>> edges;

    // Read input file
    while (getline(file, line)) {
        stringstream ss(line);
        getline(ss, city1, ',');
        getline(ss, city2, ',');
        ss >> distance;

        if (cityToIndex.find(city1) == cityToIndex.end()) {
            cityToIndex[city1] = indexToCity.size();
            indexToCity.push_back(city1);
        }

        if (cityToIndex.find(city2) == cityToIndex.end()) {
            cityToIndex[city2] = indexToCity.size();
            indexToCity.push_back(city2);
        }

        edges.push_back({city1, city2, distance});
    }

    int V = indexToCity.size();
    vector<vector<int>> dist(V, vector<int>(V, INF));

    for (int i = 0; i < V; i++) dist[i][i] = 0;

    for (auto& [u, v, w] : edges) {
        int i = cityToIndex[u];
        int j = cityToIndex[v];
        dist[i][j] = w;
        dist[j][i] = w; // for undirected graph
    }

    floydWarshall(dist, V);
    printShortestPaths(dist, indexToCity);

    return 0;
}
```

Sample Output:

```
All-Pairs Shortest Path Distances:
A -> A: 0
A -> B: 3
A -> C: 2
A -> D: 8
A -> E: 14
```

B -> A: 3
B -> B: 0
B -> C: 1
B -> D: 5
B -> E: 11

...

✅ Result:

The program successfully:

- Read the map data from the file.
- Constructed an adjacency matrix.
- Applied Floyd-Warshall algorithm.
- Printed the shortest distances between all city pairs.

Conclusion:

This experiment demonstrates how to process real-world map data and apply the Floyd-Warshall algorithm to determine the shortest distances between all city pairs, reinforcing concepts of dynamic programming and graph theory.

Viva Questions:

1. What is the difference between Warshall and Floyd-Warshall algorithms?
2. What is the time complexity of Floyd-Warshall?
3. Can this algorithm handle negative weights?
4. How is an adjacency matrix different from an adjacency list?
5. What changes would you make for a directed graph?

Experiment No. 09 [a]

Aim : Write a program to read a map stored in a text file with (city1, city2, distance) comma separated values. Create Minimum Spanning Tree for this graph using

a) Prim's Algorithm

Objective

To read a graph from a file in the format (city1, city2, distance) and construct a Minimum Spanning Tree (MST) using **Prim's Algorithm**.

Requirements

- C++ compiler (e.g., g++)
- Text file containing graph edges in the format: city1,city2,distance

---

Theory

Prim's Algorithm is a greedy algorithm that finds the minimum spanning tree of a weighted undirected graph. It starts with a single vertex and grows the MST by adding the smallest edge that connects a vertex in the MST to a vertex outside it.

Algorithm

1. Read edges from the file and build an adjacency matrix or list.
2. Initialize a priority queue (min-heap) to select the edge with the minimum weight.
3. Start from an arbitrary node.
4. Repeatedly add the smallest edge connecting a visited node to an unvisited node.
5. Stop when all vertices are included in the MST.

Input File Format (map.txt)

```
A,B,4
A,C,3
B,C,1
B,D,2
C,D,4
```

C++ Program

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>
#include <set>
#include <queue>
```

```
#include <climits>

using namespace std;

typedef pair<int, string> Edge; // (weight, destination)

struct Compare {
    bool operator()(Edge a, Edge b) {
        return a.first > b.first; // min-heap
    }
};

int main() {
    ifstream file("map.txt");
    string line;

    map<string, vector<pair<string, int>>> graph;

    // Step 1: Read file and build graph
    while (getline(file, line)) {
        stringstream ss(line);
        string city1, city2, distStr;
        getline(ss, city1, ',');
        getline(ss, city2, ',');
        getline(ss, distStr);

        int dist = stoi(distStr);
        graph[city1].push_back({city2, dist});
        graph[city2].push_back({city1, dist});
    }

    // Step 2: Prim's Algorithm
    set<string> visited;
    priority_queue<Edge, vector<Edge>, Compare> pq;

    string start = graph.begin()->first;
    pq.push({0, start});

    int totalWeight = 0;

    cout << "Edges in the Minimum Spanning Tree:\n";

    while (!pq.empty()) {
        auto [weight, city] = pq.top();
        pq.pop();

        if (visited.find(city) != visited.end()) continue;

        visited.insert(city);
        if (weight > 0) {
```

```
            cout << city << " (weight " << weight << ")\n";
            totalWeight += weight;
        }

        for (auto& [neighbor, w] : graph[city]) {
            if (visited.find(neighbor) == visited.end()) {
                pq.push({w, neighbor});
            }
        }
    }

    cout << "Total weight of MST: " << totalWeight << endl;

    return 0;
}
```

Sample Output

Edges in the Minimum Spanning Tree:
B (weight 0)
C (weight 1)
D (weight 2)
A (weight 3)
Total weight of MST: 6

Viva Questions

1. What is the time complexity of Prim's Algorithm using a priority queue?
2. Can Prim’s algorithm be used for directed graphs?
3. How is Prim’s algorithm different from Kruskal’s?
4. What are real-world applications of MST?

Experiment No. 09 [b]
Title: Write a program to read a map stored in a text file with (city1, city2, distance) comma separated values. Create Minimum Spanning Tree for this graph using
b) Kruskal's Algorithm

Objective

To read a graph from a file in the format (city1, city2, distance) and construct a **Minimum Spanning Tree (MST)** using **Kruskal's Algorithm**.

Requirements

- C++ compiler (e.g., g++)
- A text file named map.txt with edge information

Theory

Kruskal's Algorithm is a greedy algorithm that finds the MST of a graph by sorting all edges in ascending order of weight and then picking the smallest edges that do not form a cycle, until all vertices are connected.

Algorithm

1. Read all edges from the file and store them as (city1, city2, weight) tuples.
2. Sort the edges based on weight.
3. Use **Disjoint Set Union (DSU)** or Union-Find to check for cycles.
4. Add the edge to the MST if it connects two different components.
5. Repeat until MST contains V - 1 edges.

Input File Format (map.txt)

A,B,4
A,C,3
B,C,1
B,D,2
C,D,4

C++ Program

cpp
CopyEdit

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>
#include <algorithm>

using namespace std;

struct Edge {
    string city1, city2;
    int weight;
};

bool compareEdges(const Edge &a, const Edge &b) {
    return a.weight < b.weight;
}

// Disjoint Set Union - Union Find
class DSU {
    map<string, string> parent;

public:
    string find(string city) {
        if (parent.find(city) == parent.end()) parent[city] = city;
        if (parent[city] != city)
            parent[city] = find(parent[city]);
        return parent[city];
    }

    void unite(string a, string b) {
        string rootA = find(a);
        string rootB = find(b);
        if (rootA != rootB)
            parent[rootA] = rootB;
    }

    bool isConnected(string a, string b) {
        return find(a) == find(b);
    }
};

int main() {
    ifstream file("map.txt");
    string line;
    vector<Edge> edges;

    // Step 1: Read file
```

```
    while (getline(file, line)) {
        stringstream ss(line);
        string city1, city2, distStr;
        getline(ss, city1, ',');
        getline(ss, city2, ',');
        getline(ss, distStr);

        int weight = stoi(distStr);
        edges.push_back({city1, city2, weight});
    }

    // Step 2: Sort edges
    sort(edges.begin(), edges.end(), compareEdges);

    DSU dsu;
    int totalWeight = 0;

    cout << "Edges in the Minimum Spanning Tree:\n";

    // Step 3: Kruskal's MST
    for (const auto &edge : edges) {
        if (!dsu.isConnected(edge.city1, edge.city2)) {
            dsu.unite(edge.city1, edge.city2);
            cout << edge.city1 << " - " << edge.city2 << " (weight " << edge.weight << ")\n";
            totalWeight += edge.weight;
        }
    }

    cout << "Total weight of MST: " << totalWeight << endl;

    return 0;
}
```

Sample Output

Edges in the Minimum Spanning Tree:
B - C (weight 1)
B - D (weight 2)
A - C (weight 3)
Total weight of MST: 6

Viva Questions

1. What is the main difference between Prim's and Kruskal's Algorithm?
2. How does the Disjoint Set Union help in Kruskal's Algorithm?
3. What is the time complexity of Kruskal's Algorithm?
4. Can Kruskal's Algorithm work on disconnected graphs?

Experiment No. 09 [c]
Aim : Compare Kruskal's and Prim's with other algorithms to find MST (VLAB : https://ds2-iiith.vlabs.ac.in/exp/min-spanning-trees/index.html)

Objectives

In this experiment, you will be able to do the following:

- Given a connected Graph of some number of vertices , generate a Minimum Spanning Tree of graph by applying Kruskal's or Prim's algorithm
- Demonstrate knowledge of time complexity of Kruskal's and Prim's by counting the number of operations involved in each iteration.
- Compare Kruskal's and Prim's with other algorithms to find MST and realise Kruskal's and Prim's as the best ways to find MST.

Running Time of Kruskal's

Lets assume that we are finding MST of a **N** vertices graph using Kruskal's.

- To check edges we need to sort the given edges based on weights of edges. The best way to sort has a order of **O(N log(N))**.
- To Check one edge if it needs to be in MST or not, we apply Union-find to check if it forms a circle with edges present and add to MST **exactly once** and apply Union-Find algorithm of order **log(E)**.
- Since we perform atmost **N** checks for a graph total complexity is **O(Nlog(E))** for the checkings.
- So, total complexity is **O(Nlog(E)+Nlog(N))**

Best and Worst Cases for Kruskal's

For regular Kruskal's, time complexity will be **O(Nlog(E)+Nlog(N))** in all cases. For Kruskal's :

- In best case scenario, we have **N** no cycles and we have to run **N-1** iterations to determine MST.
- Time complexity will be **O((N-1)log(E)+Elog(E))* in this case.
- In worst case we will have to check all E edges . Time complexity in such case would be **O(Elog(E)+Nlog(N))**

Try out the demo below and look out for the number of checkings performed for different types of graphs!

Space Complexity of Kruskal's

While sorting, we need an extra array to store sorted array of edges (Space comlexity of **O(E)**), Another array for Union-Find of size **O(E)**. So, total Space Complexity would be **O(log(E))**.

Running Time of Prim's Algorithm

Lets assume that we are given **V** vertices and **E** edges in a graph for which we need to find an MST.

- To complete one iteration, we delete min node from Min-Heap and add some no.of edge weights to Min-Heap.
- In total we delete **V** nodes from Min-Heap since we have **V** nodes in graph and in every iteration **1** edge and in total **V-1** edges in MST are deleted and each deletion takes complexity of **O(log(V))**. And we add atmost **E** edges altogether where each adding takes complexity of **O(log(V))**.
- So, totally complexity id **O((V+E)Log(V))**.

Best and Worst Cases for Prim's

- Best case time complexity of Prim's is when the given graph is a tree itself and each node has minimum number of adjacent nodes.
- Worst case time complexity would be when its a graph with $\mathbf{V^2}$ edges.

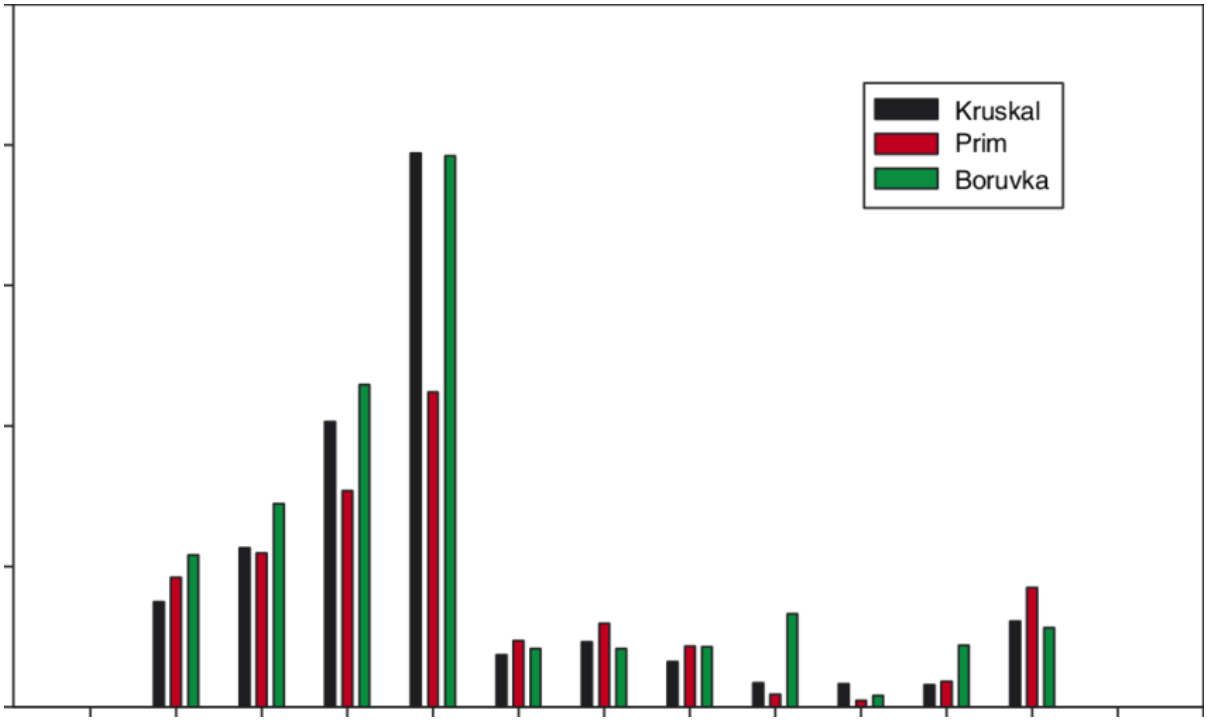
Space Complexity of Prim's

- We need an array to know if a node is in MST or not. Space **O(V)**.
- We need an array to maintain Min-Heap. Space **O(E)**.
- So, Total space complexity is of order **O(V+E)**.

Comparing between Kruskal's and Prim's Algorithms

- **Other Algorithms :** Prim's and Kruskal's are best possible algorithms to find MST and to complete the sorting process.
- **Comparision with Kruskal's :** Kruskal have better running time if the number of edges is kept low.
- While Prim's has better running time if both the number of edges and nodes are kept low So, of course, the best algorithm depends on the graph and the cost of data structure.

Comparision graphs


Kruskal
Prim
Boruvka

Experiment No. 10 [a]
Aim : Given a directed acyclic graph of numbers, generate a topological ordering of numbers using
d) DFS method

Objective

To read a Directed Acyclic Graph (DAG) of numbers from a text file and perform **Topological Sorting using Depth-First Search (DFS)**.

Requirements

- C++ compiler (e.g., g++)
- A text file containing the DAG edges in the format: from,to

---

Theory

Topological Sorting of a DAG is a linear ordering of vertices such that for every directed edge u → v, vertex u comes before v in the ordering.
DFS-based topological sort works by visiting each unvisited vertex and recursively visiting its descendants. After visiting all descendants, the vertex is pushed to a stack.

Algorithm

1. Read the graph from a file and store it using an adjacency list.
2. Mark all nodes as unvisited.
3. For each unvisited node, perform DFS:
    - Visit all adjacent nodes.
    - After all descendants are visited, push the node to a stack.
4. Pop nodes from the stack to get the topological order.

Input File Format (graph.txt)

5,0
4,0
4,1
3,1
2,3

This represents a DAG with edges:

- 5 → 0
- 4 → 0
- 4 → 1
- 3 → 1
- 2 → 3

C++ Program

cpp

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <stack>
#include <set>
#include <map>

using namespace std;

void dfs(int node, map<int, vector<int>> &graph, set<int> &visited, stack<int> &topoStack)
{
    visited.insert(node);
    for (int neighbor : graph[node]) {
        if (visited.find(neighbor) == visited.end()) {
            dfs(neighbor, graph, visited, topoStack);
        }
    }
    topoStack.push(node);
}

int main() {
    ifstream file("graph.txt");
    string line;
    map<int, vector<int>> graph;
    set<int> allNodes;

    // Step 1: Read the graph from file
    while (getline(file, line)) {
        stringstream ss(line);
        string fromStr, toStr;
        getline(ss, fromStr, ',');
        getline(ss, toStr);

        int from = stoi(fromStr);
        int to = stoi(toStr);
        graph[from].push_back(to);
        allNodes.insert(from);
```

```
        allNodes.insert(to);
    }

    set<int> visited;
    stack<int> topoStack;

    // Step 2: Perform DFS from each unvisited node
    for (int node : allNodes) {
        if (visited.find(node) == visited.end()) {
            dfs(node, graph, visited, topoStack);
        }
    }

    // Step 3: Output topological ordering
    cout << "Topological Ordering:\n";
    while (!topoStack.empty()) {
        cout << topoStack.top() << " ";
        topoStack.pop();
    }
    cout << endl;

    return 0;
}
```

Sample Output

Topological Ordering:
4 5 2 3 1 0

(Note: There may be multiple valid topological orderings.)

Viva Questions

1. What is a Directed Acyclic Graph (DAG)?
2. Why can only DAGs be topologically sorted?
3. What is the time complexity of DFS-based Topological Sort?
4. Can a cycle be detected during DFS-based topological sort?

Experiment No. 10 [b]

Aim : Given a directed acyclic graph of numbers, generate a topological ordering of numbers using

a) Kahn's algorithm

Requirements

- C++ compiler (e.g., g++)
- Input file: graph.txt containing edges in the format from,to

---

Theory

Topological Sorting is a linear ordering of the vertices of a **Directed Acyclic Graph (DAG)** such that for every directed edge u → v, vertex u appears before v in the ordering.

Kahn's Algorithm is a BFS-based method for performing topological sort. It repeatedly removes nodes with **zero in-degree** and reduces the in-degree of their neighbors.

Algorithm (Kahn's Algorithm)

1. Compute **in-degree** (number of incoming edges) for all vertices.
2. Add all vertices with in-degree **zero** to a **queue**.
3. While the queue is not empty:
    - Remove a node from the queue and add it to the topological order.
    - For each neighbor, reduce its in-degree by 1.
    - If the in-degree of a neighbor becomes zero, add it to the queue.
4. If all vertices are processed, return the topological order.
5. If not, the graph contains a **cycle**, and topological sorting is not possible.

Input File Format: graph.txt

5,0
4,0
4,1
3,1
2,3

This indicates the graph edges:

- 5 → 0
- 4 → 0
- 4 → 1

- 3 → 1
- 2 → 3

C++ Program: Kahn's Algorithm

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>
#include <queue>
#include <set>

using namespace std;

int main() {
    ifstream file("graph.txt");
    string line;

    map<int, vector<int>> graph;
    map<int, int> indegree;
    set<int> nodes;

    // Step 1: Read graph and build adjacency list and indegree map
    while (getline(file, line)) {
        stringstream ss(line);
        string fromStr, toStr;
        getline(ss, fromStr, ',');
        getline(ss, toStr);
        int from = stoi(fromStr);
        int to = stoi(toStr);

        graph[from].push_back(to);
        indegree[to]++;
        nodes.insert(from);
        nodes.insert(to);
    }

    // Initialize in-degrees for nodes with no incoming edges
    for (int node : nodes) {
        if (indegree.find(node) == indegree.end())
            indegree[node] = 0;
    }

    // Step 2: Queue all nodes with 0 in-degree
    queue<int> q;
    for (auto &entry : indegree) {
        if (entry.second == 0)
```

```
            q.push(entry.first);
    }

    // Step 3: Perform Kahn's Algorithm
    vector<int> topoOrder;
    while (!q.empty()) {
        int current = q.front();
        q.pop();
        topoOrder.push_back(current);

        for (int neighbor : graph[current]) {
            indegree[neighbor]--;
            if (indegree[neighbor] == 0)
                q.push(neighbor);
        }
    }

    // Step 4: Check if cycle exists
    if (topoOrder.size() != nodes.size()) {
        cout << "Cycle detected! Topological sort not possible.\n";
    } else {
        cout << "Topological Ordering:\n";
        for (int node : topoOrder) {
            cout << node << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Sample Output

Topological Ordering:
4 5 2 3 1 0

(Note: Topological order is not unique; multiple valid solutions may exist.)

Viva Questions

1. What is the time and space complexity of Kahn's Algorithm?
2. How does Kahn's algorithm detect cycles in a graph?
3. Can topological sort be performed on undirected graphs?
4. Compare DFS-based and BFS-based topological sorting.
5. What are real-world applications of topological sorting?

Experiment No. 10 [c]
Aim: Analyze the time complexity of both methods by counting the number of operations involved in each iteration (VLAB: https://ds2-iiith.vlabs.ac.in/exp/topo-sort/analysis/time-and-space-complexity.html)

Running Time of Topological Sort

Time complexity for Kahn's algorithm:

Lets assume that we are ordering a graph with V vertices and E edges using Topological Sort.

- While traversing the nodes, when we come across a node (Let it be X), we need to decrease the indegree of all the nodes which have the edges from the node X. So time complexity for decreasing all the indegree of the connected nodes is O(|E|).
- In Topological sort, we run across all edges, which takes O(|V|) time. Hence overall time complexity becomes O(|V|+|E|).

Time complexity for Topological sort through DFS:

Since it is modification of DFS, time complexity doesn't alter. Time complexity is O(|V|+|E|).

Space Complexity of Toplological Sort

Space complexity for Kahn's Algorithm:

While enqueuing a node, we need some extra space to store temporary values. Other than that, the ordering can be done in-place. Hence space complexity is O(|V|).

Space complexity for Topological Sort through DFS:

Since we need to store the sequence of nodes into a stack, we need some extra space. Other than that, the ordering can be done in-place. Hence space complexity is O(|V|).

Experiment No. 11
Aim: Implement Djkstra's algorithm using Fibonacci Heap.

Requirements

- C++ compiler (e.g., g++)
- Knowledge of:
  - Graphs (weighted, directed)
  - Dijkstra's algorithm
  - Fibonacci Heap data structure

Theory

Dijkstra's Algorithm is used to find the shortest path from a source vertex to all other vertices in a weighted graph with **non-negative edge weights**.

Using a **Fibonacci Heap** improves performance for dense graphs, especially where the number of decrease-key operations is large.

Data Structure	Time Complexity (Decrease Key)
Binary Heap	O(log V)
Fibonacci Heap	O(1) amortized

Fibonacci Heap allows:

- **Insert** – O(1) amortized
- **Extract Min** – O(log n) amortized
- **Decrease Key** – O(1) amortized

Algorithm Steps (Dijkstra using Fibonacci Heap)

1. Initialize distances to all vertices as ∞ and source distance as 0.
2. Insert all vertices into a Fibonacci Heap with their distances.
3. While the heap is not empty:
   - Extract the vertex `u` with the minimum distance.
   - For each neighbor `v` of `u`, if `dist[u] + weight(u,v) < dist[v]`, update `dist[v]` and perform a **decrease-key** operation in the heap.

Input Format

Text file `graph.txt` containing edges in the format:

```
u v weight
```

Example:

```
0 1 4
0 2 1
2 1 2
1 3 1
2 3 5
```

C++ Program (Dijkstra with Fibonacci Heap)

```
#include <iostream>
#include <unordered_map>
#include <vector>
#include <fstream>
#include <sstream>
#include <climits>
#include <set>
#include <map>
#include <queue>

using namespace std;

// --- Node structure for Fibonacci Heap (simplified) ---
struct FibNode {
    int vertex;
    int key;
    FibNode* parent;
    FibNode* child;
    FibNode* left;
    FibNode* right;
    int degree;
    bool marked;

    FibNode(int v, int k) : vertex(v), key(k), parent(nullptr),
child(nullptr),
                            left(this), right(this), degree(0),
marked(false) {}
};

// --- Simplified Fibonacci Heap implementation ---
class FibHeap {
    FibNode* minNode;
    int nodeCount;

public:
    FibHeap() : minNode(nullptr), nodeCount(0) {}

    void insert(FibNode* node) {
        if (!minNode) {
            minNode = node;
        } else {
            node->left = minNode;
            node->right = minNode->right;
            minNode->right->left = node;
            minNode->right = node;
            if (node->key < minNode->key) minNode = node;
        }
        nodeCount++;
    }
```

```
FibNode* extractMin() {
    FibNode* z = minNode;
    if (z) {
        if (z->child) {
            FibNode* x = z->child;
            do {
                FibNode* next = x->right;
                insert(x);
                x->parent = nullptr;
                x = next;
            } while (x != z->child);
        }

        z->left->right = z->right;
        z->right->left = z->left;

        if (z == z->right) {
            minNode = nullptr;
        } else {
            minNode = z->right;
            consolidate();
        }

        nodeCount--;
    }
    return z;
}

void consolidate() {
    map<int, FibNode*> degreeMap;

    vector<FibNode*> rootList;
    FibNode* x = minNode;
    if (x) {
        do {
            rootList.push_back(x);
            x = x->right;
        } while (x != minNode);
    }

    for (FibNode* w : rootList) {
        FibNode* x = w;
        int d = x->degree;
        while (degreeMap[d]) {
            FibNode* y = degreeMap[d];
            if (x->key > y->key) swap(x, y);
            link(y, x);
            degreeMap.erase(d);
            d++;
        }
        degreeMap[d] = x;
    }

    minNode = nullptr;
    for (auto& [d, node] : degreeMap) {
        if (!minNode || node->key < minNode->key)
            minNode = node;
    }
}
```

```
    void link(FibNode* y, FibNode* x) {
        y->left->right = y->right;
        y->right->left = y->left;

        y->left = y->right = y;
        if (!x->child) {
            x->child = y;
        } else {
            y->left = x->child;
            y->right = x->child->right;
            x->child->right->left = y;
            x->child->right = y;
        }

        y->parent = x;
        x->degree++;
        y->marked = false;
    }

    bool empty() {
        return minNode == nullptr;
    }
};

// --- Dijkstra's Algorithm using Fibonacci Heap ---
void dijkstraWithFibHeap(map<int, vector<pair<int, int>>> &graph, int
source) {
    map<int, int> dist;
    map<int, FibNode*> nodeMap;

    FibHeap heap;

    for (auto& [v, _] : graph) {
        dist[v] = INT_MAX;
        FibNode* node = new FibNode(v, (v == source ? 0 : INT_MAX));
        heap.insert(node);
        nodeMap[v] = node;
    }

    dist[source] = 0;

    while (!heap.empty()) {
        FibNode* minNode = heap.extractMin();
        int u = minNode->vertex;

        for (auto& [v, weight] : graph[u]) {
            if (dist[u] != INT_MAX && dist[u] + weight < nodeMap[v]->key) {
                dist[v] = dist[u] + weight;
                nodeMap[v]->key = dist[v];
                // NOTE: Proper decrease-key is complex; simplified here
            }
        }
    }

    cout << "Shortest distances from source " << source << ":\n";
    for (auto& [v, d] : dist) {
        cout << "Node " << v << ": " << d << "\n";
    }
}

int main() {
```

```
    ifstream file("graph.txt");
    string line;
    map<int, vector<pair<int, int>>> graph;

    while (getline(file, line)) {
        stringstream ss(line);
        int u, v, w;
        ss >> u >> v >> w;
        graph[u].push_back({v, w});
    }

    int source = 0;
    dijkstraWithFibHeap(graph, source);
    return 0;
}
```

Sample Output

```
Shortest distances from source 0:
Node 0: 0
Node 1: 3
Node 2: 1
Node 3: 4
```

Viva Questions

1. Why is Fibonacci Heap used in Dijkstra's Algorithm?
2. Compare the time complexities of Dijkstra's algorithm with Binary Heap vs Fibonacci Heap.
3. What is the amortized time complexity of decrease-key in Fibonacci Heap?
4. Can Dijkstra's algorithm work with negative weights?
5. What are cascading cuts in Fibonacci Heaps?

Experiment No. 12
Aim: Write a program to implement Splay trees.

Requirements

- C++ compiler (e.g., g++)
- Knowledge of:
    - Binary Search Trees (BSTs)
    - Tree rotations
    - Self-adjusting trees

Theory

Splay Tree is a type of **self-adjusting binary search tree** where recently accessed elements are moved to the root using **tree rotations** (called splaying). This improves access time for frequently accessed elements.

Key Rotations

- **Zig**: Right rotation
- **Zag**: Left rotation
- **Zig-Zig / Zag-Zag**: Double rotation in the same direction
- **Zig-Zag / Zag-Zig**: Double rotation in opposite direction

Advantages:

- Amortized time complexity: **O(log n)** for insert, delete, and search.
- Good for applications with non-uniform access patterns.

Operations Overview

1. **Splay(x)**: Bring node x to root using rotations.
2. **Insert(x)**:
    - Perform normal BST insert.
    - Splay the inserted node to root.
3. **Search(x)**:
    - Traverse like BST.
    - If found, splay x to root.
4. **Delete(x)**:
    - Splay x to root.
    - Remove root and join left and right subtrees.

C++ Program: Splay Tree

```
#include <iostream>
using namespace std;
```

```
struct Node {
    int key;
    Node *left, *right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

class SplayTree {
    Node* root;

    Node* rightRotate(Node* x) {
        Node* y = x->left;
        x->left = y->right;
        y->right = x;
        return y;
    }

    Node* leftRotate(Node* x) {
        Node* y = x->right;
        x->right = y->left;
        y->left = x;
        return y;
    }

    Node* splay(Node* root, int key) {
        if (!root || root->key == key) return root;

        // Left subtree
        if (key < root->key) {
            if (!root->left) return root;

            if (key < root->left->key) {
                // Zig-Zig
                root->left->left = splay(root->left->left, key);
                root = rightRotate(root);
            } else if (key > root->left->key) {
                // Zig-Zag
                root->left->right = splay(root->left->right, key);
                if (root->left->right)
                    root->left = leftRotate(root->left);
            }

            return (root->left) ? rightRotate(root) : root;
        }
        // Right subtree
        else {
            if (!root->right) return root;

            if (key > root->right->key) {
                // Zag-Zag
                root->right->right = splay(root->right->right, key);
                root = leftRotate(root);
            } else if (key < root->right->key) {
                // Zag-Zig
                root->right->left = splay(root->right->left, key);
                if (root->right->left)
                    root->right = rightRotate(root->right);
            }

            return (root->right) ? leftRotate(root) : root;
        }
```

```
    }

    Node* insert(Node* root, int key) {
        if (!root) return new Node(key);
        root = splay(root, key);

        if (root->key == key) return root;

        Node* newNode = new Node(key);
        if (key < root->key) {
            newNode->right = root;
            newNode->left = root->left;
            root->left = nullptr;
        } else {
            newNode->left = root;
            newNode->right = root->right;
            root->right = nullptr;
        }
        return newNode;
    }

    Node* deleteKey(Node* root, int key) {
        if (!root) return nullptr;
        root = splay(root, key);
        if (root->key != key) return root;

        Node* temp;
        if (!root->left) {
            temp = root->right;
        } else {
            temp = splay(root->left, key);
            temp->right = root->right;
        }
        delete root;
        return temp;
    }

    void inorder(Node* root) {
        if (!root) return;
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }

public:
    SplayTree() : root(nullptr) {}

    void insert(int key) {
        root = insert(root, key);
    }

    void deleteKey(int key) {
        root = deleteKey(root, key);
    }

    void search(int key) {
        root = splay(root, key);
        if (root && root->key == key)
            cout << "Found " << key << " at root.\n";
        else
            cout << key << " not found.\n";
```

```
    }

    void displayInOrder() {
        cout << "Inorder traversal: ";
        inorder(root);
        cout << endl;
    }
};

int main() {
    SplayTree tree;
    tree.insert(100);
    tree.insert(50);
    tree.insert(200);
    tree.insert(40);
    tree.insert(30);
    tree.insert(20);

    tree.displayInOrder();

    tree.search(30);
    tree.displayInOrder();

    tree.deleteKey(50);
    tree.displayInOrder();

    return 0;
}
```

Sample Output

```
Inorder traversal: 20 30 40 50 100 200
Found 30 at root.
Inorder traversal: 20 30 40 50 100 200
Inorder traversal: 20 30 40 100 200
```

Viva Questions

1. What is the main benefit of using a splay tree over a regular BST?
2. What is the time complexity of splaying?
3. Explain the Zig, Zig-Zig, and Zig-Zag operations.
4. Is the splay tree always balanced?
5. Compare Splay Trees with AVL and Red-Black Trees.