<u>DSA ASSIGNMENT-1</u>

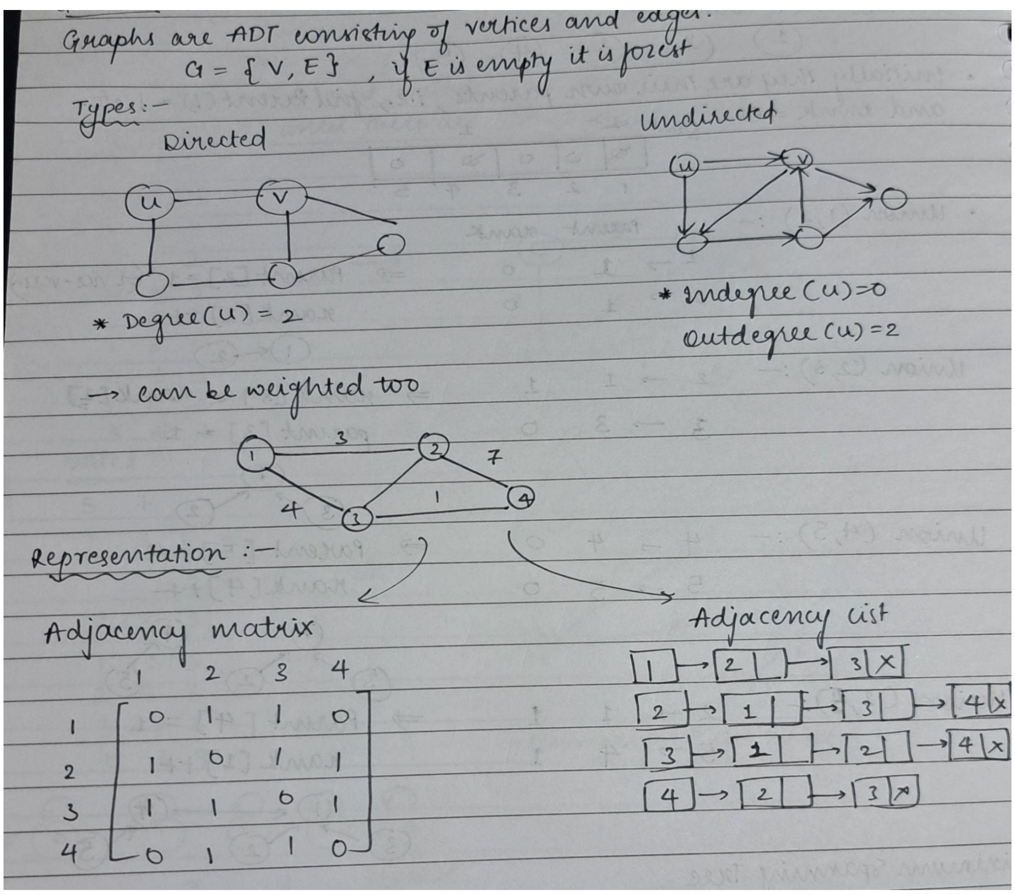
Name: URMI RITESHKUMAR MIRANI, TVISHA PATEL
Roll number: 22BCE363, 22BCE361

Problem Statement

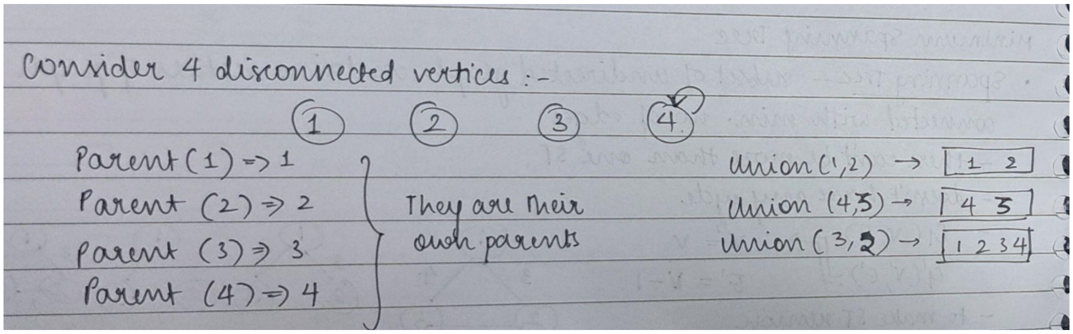
Design an abstract data type Graph to represent a data structure for an undirected graph. Write a program that implements Kruskal's minimal spanning tree algorithm. The data for the undirected weighted graph should be read from a file. You should determine a file format that will allow you to represent any undirected weighted graph. Discuss the correctness of Kruskal's Algorithm.

Background Information

1. <u>Graphs</u>



2. findParent() & union()



Consider 4 disconnected vertices :-

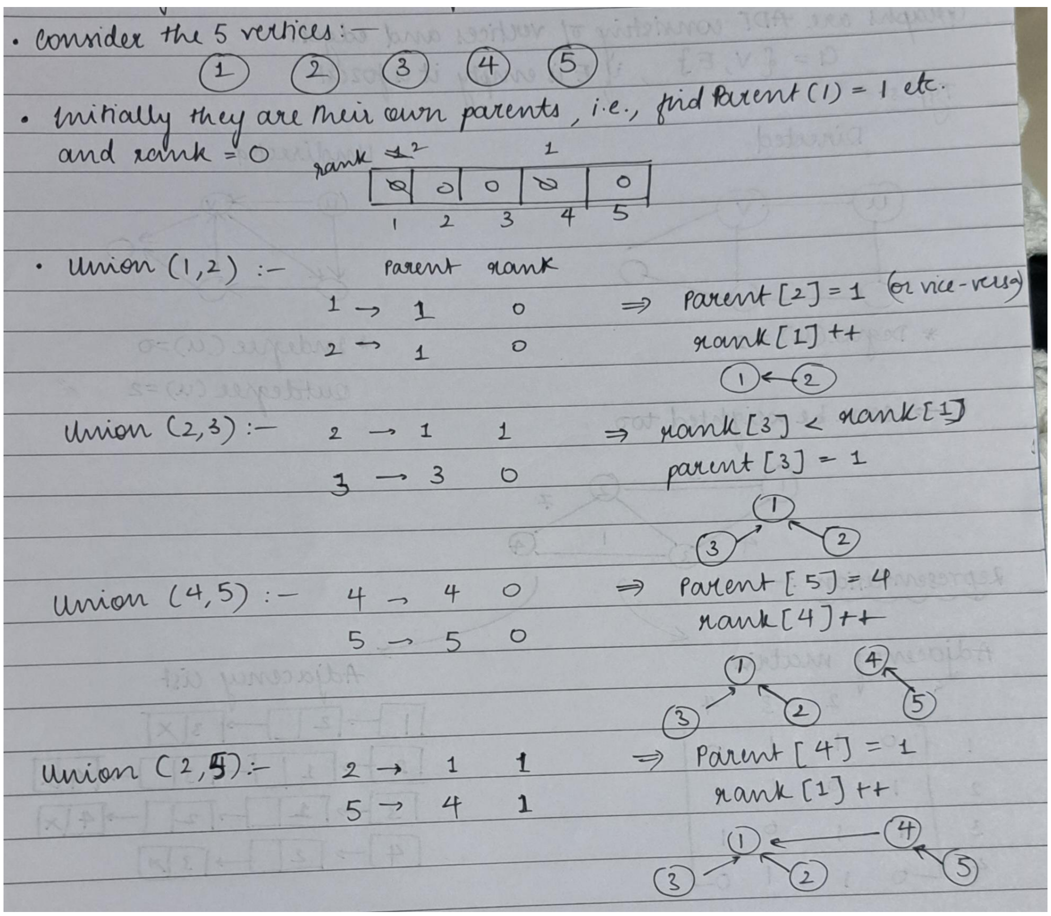
1 2 3 4

Parent (1) => 1
Parent (2) => 2
Parent (3) => 3
Parent (4) => 4

They are their own parents

Union (1,2) → $\boxed{1\ 2}$
Union (4,3) → $\boxed{4\ 3}$
Union (3,2) → $\boxed{1\ 2\ 3\ 4}$

3. Union by rank & Path compression



• Consider the 5 vertices :-

1 2 3 4 5

• Initially they are their own parents, i.e., find Parent (1) = 1 etc. and rank = 0

rank	~~0~~ 1 2	0	0	~~0~~ 1	0
	1	2	3	4	5

• Union (1,2) :-

	Parent	rank
1 →	1	0
2 →	1	0

⇒ Parent [2] = 1 (or vice-versa)
rank [1] ++

1 ← 2

Union (2,3) :-

	Parent	rank
2 →	1	1
3 →	3	0

⇒ rank [3] < rank [1]
parent [3] = 1

Union (4,5) :-

	Parent	rank
4 →	4	0
5 →	5	0

⇒ Parent [5] = 4
rank [4] ++

Union (2,5) :-

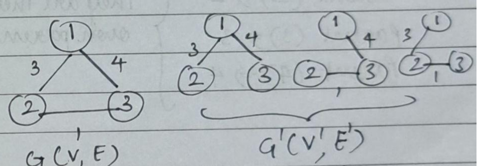
	Parent	rank
2 →	1	1
5 →	4	1

⇒ Parent [4] = 1
rank [1] ++

4. Minimum Spanning Tree

Minimum Spanning Tree

- Spanning Tree – subset of undirected graph contains vertices of graph connected with min. no of edges
  - there can't be more than one ST
  - doesn't have any cycle
  - $G(V, E) \rightarrow V' = V$
    $G'(V', E') \rightarrow E' = V-1$
  - to make ST remove $\max(E-V+1)$ edges



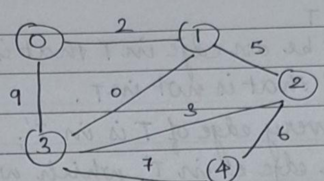
- Min. spanning Tree – spanning tree with min. possible total edge weight

5. Kruskal's algorithm

Algorithm:-
1. Sort all edges in inc. order of wt.
2. Pick smallest edge. Check if forms a cycle with spanning tree formed so far. If cycle not formed, include edge. otherwise, discard.
3. Repeat step (2) until there are $(V-1)$ edges in spanning tree.

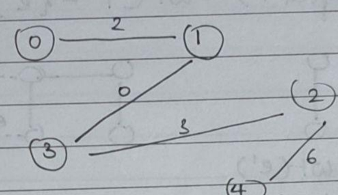
input.txt

5 7
0 1 2
0 3 9
1 2 5
1 3 0
2 3 3
2 4 6
3 4 7

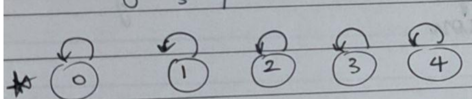


sorted

5 7
1 3 0 ✓
0 1 2 ✓
2 3 3 ✓
1 2 5
2 4 6 ✓
3 4 7
0 3 9



Min. weight = 0 + 2 + 3 + 6 = 11

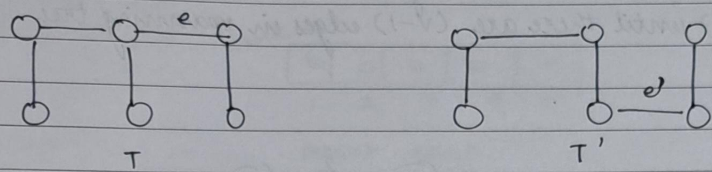


x	y	parents	
1	3	1 → 1 3 → 3	$\boxed{1\ 3}$
0	1	0 → 0 1 → 1	$\boxed{0\ 1\ 3}$
2	3	2 → 2 3 → 3	$\boxed{0\ 1\ 3\ 2}$
1	2	1 → 0 2 → 0	
2	4	2 → 0 4 → 4	$\boxed{0\ 1\ 3\ 2\ 4}$

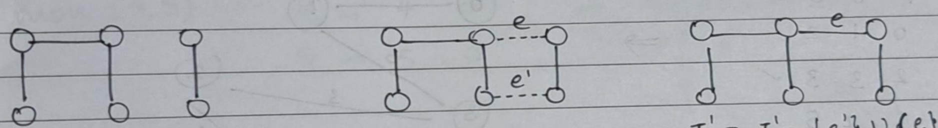
(disjoint ⇒ belong to diff. components)
different parents → union
same parents → ignore

6. Correctness of Kruskal's algorithm

- Let T be spanning tree generated by Kruskal's algorithm for G.
  Let T' be minimum spanning tree for G
  Show that :- T and T' have same wt.
- Assume $wt(T') < wt(T)$



- There must be an edge in T that is not in T' and there should be an edge e' in T' that is not in T.
  Because if every edge of T is in T'; $T = T'$ & $wt(T) = wt(T')$
- Remove the edge e' in T', which will disconnect T' into two components
- According to Kruskal's algorithm, $wt(e) \le wt(e')$. The two components of T' could be merged using edge e which will lower the weight of T'.
- $wt(\text{modified } T') = wt(T' - \{e'\} \cup \{e\}) \le wt(T')$.



- Hence by reducing edge difference, and making T' approach T, we are able to decrease the weight of T' only further. Thus making T' not a MST to start with $\Rightarrow$ contradiction
- T is a minimum spanning tree.

Code

1. Abstract datatype of graph

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

const int MAX_V = 100;
class Graph {
private:
    int vertices;
    int matrix[MAX_V][MAX_V];

public:

    Graph(int V) {
        vertices = V;
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                matrix[i][j] = 0;
            }
        }
    }

    void addEdge(int u, int v) {
        if (u >= vertices || v >= vertices) {
            cout << "Vertex out of range." << endl;
            return;
        }
        matrix[u][v] = 1;
        matrix[v][u] = 1;
    }

    void removeEdge(int u, int v) {
```

```
    if (u >= vertices || v >= vertices) {
        cout << "Vertex out of range." << endl;
        return;
    }
    matrix[u][v] = 0;
    matrix[v][u] = 0;
}

bool hasEdge(int u, int v) {
    if (u >= vertices || v >= vertices) {
        cout << "Vertex out of range." << endl;
        return false;
    }
    return (matrix[u][v] == 1);
}

void addVertex() {
    vertices++;
    for (int i = 0; i < vertices; i++) {
        matrix[i][vertices - 1] = 0;
        matrix[vertices - 1][i] = 0;
    }
}

void removeVertex(int v) {
    if (v >= vertices) {
        cout << "Vertex out of range." << endl;
        return;
    }
    for (int i = 0; i < vertices; i++) {
        matrix[i][v] = 0;
        matrix[v][i] = 0;
    }
    vertices--;
}
```

```
    int getDegree(int v) {
        if (v >= vertices) {
            cout << "Vertex out of range." << endl;
            return -1;
        }
        int degree = 0;
        for (int i = 0; i < vertices; i++) {
            if (matrix[v][i] == 1) {
                degree++;
            }
        }
        return degree;
    }

    void printGraph() {
        cout << "Adjacency Matrix:" << endl;
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                cout << matrix[i][j] << " ";
            }
            cout << endl;
        }
    }
};

int main() {
    int choice;
    int nVertices;
    bool exitProgram = false;

    cout << "Enter the number of vertices for the graph: ";
    cin >> nVertices;

    Graph graph(nVertices);
```

```
while (!exitProgram) {
    cout << "Graph Operations:" << endl;
    cout << "1. Add Edge" << endl;
    cout << "2. Remove Edge" << endl;
    cout << "3. Check Edge Existence" << endl;
    cout << "4. Add Vertex" << endl;
    cout << "5. Remove Vertex" << endl;
    cout << "6. Get Vertex Degree" << endl;
    cout << "7. Print Graph" << endl;
    cout << "8. Quit" << endl;

    cout << "Enter your choice: ";
    cin >> choice;

    if (choice == 1) {
        int u, v;
        cout << "Enter vertices to add an edge (u v): ";
        cin >> u >> v;
        graph.addEdge(u, v);
    } else if (choice == 2) {
        int u, v;
        cout << "Enter vertices to remove an edge (u v): ";
        cin >> u >> v;
        graph.removeEdge(u, v);
    } else if (choice == 3) {
        int u, v;
        cout << "Enter vertices to check edge existence (u v): ";
        cin >> u >> v;
        if (graph.hasEdge(u, v)) {
            cout << "Edge exists." << endl;
        } else {
            cout << "Edge does not exist." << endl;
        }
    } else if (choice == 4) {
```

```
            graph.addVertex();
        } else if (choice == 5) {
            int vertexToRemove;
            cout << "Enter the vertex to remove: ";
            cin >> vertexToRemove;
            graph.removeVertex(vertexToRemove);
        } else if (choice == 6) {
            int vertexToCheck;
            cout << "Enter the vertex to get degree: ";
            cin >> vertexToCheck;
            int degree = graph.getDegree(vertexToCheck);
            if (degree != -1) {
                cout << "Degree of vertex " << vertexToCheck << ": " << degree <<
endl;
            }
        } else if (choice == 7) {
            graph.printGraph();
        } else if (choice == 8) {
            exitProgram = true;
        } else {
            cout << "Invalid choice" << endl;
        }
    }
    return 0;
}
```

2. Kruskal’s algorithm

```
#include <iostream>
#include <algorithm> //library for sorting, searching and manipulating elements
#include <fstream> //for file handling- input and output
#include <vector>

bool compare(std::vector<int> &a, std::vector<int> &b) {
    return a[2] < b[2];
```

```
}

void makeSet(std::vector<int> &parent, std::vector<int> &ranks, int n) {
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        ranks[i] = 0;
    }
}

int findParent(std::vector<int> &parent, int node) {
    if (parent[node] == node) {
        return node;
    }
    return parent[node] = findParent(parent, parent[node]);
}

void unionSet(int u, int v, std::vector<int> &parent, std::vector<int> &ranks) {
    u = findParent(parent, u);
    v = findParent(parent, v);
    if (ranks[u] < ranks[v])
        parent[u] = v;
    else if (ranks[v] < ranks[u])
        parent[v] = u;
    else {
        parent[v] = u;
        ranks[u]++;
    }
}

int main() {
    std::ifstream inputFile("input.txt");
    if (!inputFile.is_open()) {
        std::cerr << "Failed to open the input file." << std::endl;
        return 1;
    }
```

```
    int n, m;
    inputFile >> n >> m;
    std::vector<std::vector<int>> edges(m);
    for (int i = 0; i < m; i++) {
        int x, y, w;
        inputFile >> x >> y >> w;
        edges[i] = {x, y, w};
    }
    inputFile.close();

    std::sort(edges.begin(), edges.end(), compare);
    std::vector<int> parent(n);
    std::vector<int> ranks(n);
    makeSet(parent, ranks, n);
    int minWeight = 0;
    for (int i = 0; i < edges.size(); i++) {
        int u = findParent(parent, edges[i][0]);
        int v = findParent(parent, edges[i][1]);
        int weight = edges[i][2];
        if (u != v) {
            minWeight += weight;
            unionSet(u, v, parent, ranks);
        }
    }
    std::cout << "Minimum Weight: " << minWeight << std::endl;
    return 0;
}
```

Output

1.

```
Enter the number of vertices for the graph: 4
Graph Operations:
1. Add Edge
2. Remove Edge
3. Check Edge Existence
4. Add Vertex
5. Remove Vertex
6. Get Vertex Degree
7. Print Graph
8. Quit
Enter your choice: 1
Enter vertices to add an edge (u v): 1 2
Graph Operations:
1. Add Edge
2. Remove Edge
3. Check Edge Existence
4. Add Vertex
5. Remove Vertex
6. Get Vertex Degree
7. Print Graph
8. Quit
Enter your choice: 1
Enter vertices to add an edge (u v): 3 1
Graph Operations:
1. Add Edge
2. Remove Edge
3. Check Edge Existence
4. Add Vertex
5. Remove Vertex
6. Get Vertex Degree
7. Print Graph
8. Quit
Enter your choice: 1
Enter vertices to add an edge (u v): 0 1
Graph Operations:
1. Add Edge
2. Remove Edge
3. Check Edge Existence
4. Add Vertex
5. Remove Vertex
6. Get Vertex Degree
7. Print Graph
8. Quit
Enter your choice: 3
Enter vertices to check edge existence (u v): 1 2
Edge exists.
Graph Operations:
1. Add Edge
2. Remove Edge
3. Check Edge Existence
4. Add Vertex
5. Remove Vertex
6. Get Vertex Degree
7. Print Graph
8. Quit
Enter your choice: 7
Adjacency Matrix:
0 1 0 0
1 0 1 1
0 1 0 0
```

2.

```
Graph Operations:
1. Add Edge
2. Remove Edge
3. Check Edge Existence
4. Add Vertex
5. Remove Vertex
6. Get Vertex Degree
7. Print Graph
8. Quit
Enter your choice: 4
Graph Operations:
1. Add Edge
2. Remove Edge
3. Check Edge Existence
4. Add Vertex
5. Remove Vertex
6. Get Vertex Degree
7. Print Graph
8. Quit
Enter your choice: 7
Adjacency Matrix:
0 1 0 0 0
1 0 1 1 0
0 1 0 0 0
0 1 0 0 0
0 0 0 0 0
Graph Operations:
1. Add Edge
2. Remove Edge
3. Check Edge Existence
4. Add Vertex
5. Remove Vertex
6. Get Vertex Degree
7. Print Graph
8. Quit
Enter your choice: 5
Enter the vertex to remove: 0
Graph Operations:
1. Add Edge
2. Remove Edge
3. Check Edge Existence
4. Add Vertex
5. Remove Vertex
6. Get Vertex Degree
7. Print Graph
8. Quit
Enter your choice: 6
Enter the vertex to get degree: 1
Degree of vertex 1: 2
Graph Operations:
1. Add Edge
2. Remove Edge
3. Check Edge Existence
4. Add Vertex
5. Remove Vertex
6. Get Vertex Degree
7. Print Graph
8. Quit
Enter your choice: 8
```

2.

```
C:\Users\urmim_fax022t\OneI

Minimum Weight: 11

Process returned 0 (0x0)   execution time : 1.103 s
Press any key to continue.
```

Input file:

```
input

File   Edit   View

5 7
0 1 2
0 3 9
1 2 5
1 3 0
2 3 3
2 4 6
3 4 7
```

Format:

n m
x1 y1 w1
x2 y2 w2
...
xm ym wm

Where n is the number of nodes, m is the number of edges, and each subsequent line represents an edge with nodes xi and yi and weight wi.

Conclusion

In conclusion, we have successfully addressed the task of designing an ADT for an undirected graph, reading graph data from a file, and implementing Kruskal's algorithm to find the minimum spanning tree. Kruskal's algorithm is a reliable and widely used method for solving minimum spanning tree problems in various domains.