Module 9 - Advanced Graph

1. Introduction

Module 9: Advanced Graph

Author: YP

Learning Objectives

After completing this module, students are expected to be able to:

1. Explain the basic concepts of **Graph** (including adjacency list and adjacency matrix representation).
2. Implement graph traversal using **DFS (Depth First Search)** and **BFS (Breadth First Search)**.
3. Understand and implement the **Shortest Path algorithm (Dijkstra)**.
4. Implement **Minimum Spanning Tree (MST)** algorithms: **Prim** and **Kruskal**.
5. Compare the efficiency of manual graph algorithms with STL libraries such as **priority_queue** and **disjoint set union (DSU)**.

---

2. Graph Concept

2.1 Theory

2.1.1 Definition of Graph

A **Graph** is a data structure consisting of:

- **Vertex (node)** → represents an object.
- **Edge** → represents the relationship between objects.

Graphs can be:

- **Directed** or **Undirected**.
- **Weighted** or **Unweighted**.

2.1.2 Graph Representation

There are two main representations of graphs in programming:

1. **Adjacency Matrix**
   A 2D matrix [V][V], where V is the number of vertices. Each element indicates whether an edge exists between two nodes (commonly 1 for an edge, 0 for no edge).
   - Suitable for **dense graphs**.
   - Edge lookup is efficient with **O(1)** time complexity.

```
int graph[5][5] = {
    {0, 1, 0, 0, 1},
    {1, 0, 1, 1, 0},
    {0, 1, 0, 1, 0},
    {0, 1, 1, 0, 1},
    {1, 0, 0, 1, 0}
};
```

2. **Adjacency List**
   Each vertex stores a list of its adjacent vertices (neighbors).
   - Memory-efficient for **sparse graphs**, as only existing edges are stored.
   - Edge lookup may take up to **O(V)** in the worst case.

```
vector<int> adj[5];

adj[0].push_back(1);
adj[0].push_back(4);
adj[1].push_back(0);
adj[1].push_back(2);
adj[1].push_back(3);
```

2.1.3 Graph Traversal

Graph traversal can be performed in two main ways:

1. **Breadth First Search (BFS)**
   - Uses a queue.
   - Explores nodes level by level (broad exploration).
   - Effective in unweighted graphs to find the shortest path from one node to another.

```
void BFS(int start, vector<int> adj[], int V) {
    vector<bool> visited(V, false);
    queue<int> q;

    visited[start] = true;
    q.push(start);

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        cout << u << " ";

        for (int v : adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}
```

2. **Depth First Search (DFS)**

- Uses recursion or a stack.
- Explores as deep as possible before backtracking.
- Useful for detecting cycles, performing topological sort, or finding connected components.

```
void DFSUtil(int u, vector<int> adj[], vector<bool>& visited) {
    visited[u] = true;
    cout << u << " ";
    for (int v : adj[u]) {
        if (!visited[v]) {
            DFSUtil(v, adj, visited);
        }
    }
}

void DFS(int start, vector<int> adj[], int V) {
    vector<bool> visited(V, false);
    DFSUtil(start, adj, visited);
}
```

2.1.4 Shortest Path Algorithm: Dijkstra

- Finds the shortest path from a source node to all other nodes in a **positively weighted graph**.
- Uses a priority queue (min-heap) to efficiently select edges with the smallest weights.
- Time complexity: **O((V + E) log V)**.

```
void dijkstra(int V, vector<pair<int,int>> adj[], int src) {
    vector<int> dist(V, INT_MAX);
    dist[src] = 0;

    priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;
    pq.push({0, src});

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        for (auto [v, w] : adj[u]) {
            if (dist[u] + w < dist[v]) {
```

```
                dist[v] = dist[u] + w;
                pq.push({dist[v], v});
            }
        }
    }

    cout << "Jarak terpendek dari " << src << ":\n";
    for (int i = 0; i < V; i++) {
        cout << "Ke " << i << " = " << dist[i] << endl;
    }
}
```

2.1.5 Minimum Spanning Tree (MST)

Prim’s Algorithm

- Starts from a single node, repeatedly adding the minimum weight edge that connects a new node to the tree.
- Efficient with **O(E log V)** complexity using a priority queue.
- Suitable for **dense graphs**.

```
#include <bits/stdc++.h>
using namespace std;

const int INF = 1e9;

void primMST(int n, vector<vector<pair<int, int>>> &adj) {
    vector<int> key(n, INF);
    vector<bool> inMST(n, false);
    vector<int> parent(n, -1);

    // Mulai dari node 0
    key[0] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
    pq.push({0, 0}); // {weight, node}

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
```

```
        if (inMST[u]) continue;
        inMST[u] = true;

        for (auto &[v, w] : adj[u]) {
            if (!inMST[v] && w < key[v]) {
                key[v] = w;
                pq.push({key[v], v});
                parent[v] = u;
            }
        }
    }

    cout << "Edges in MST (Prim):\n";
    for (int i = 1; i < n; i++) {
        cout << parent[i] << " - " << i << "\n";
    }
}

int main() {
    int n = 5;
    vector<vector<pair<int, int>>> adj(n);

    // contoh graf berbobot (undirected)
    adj[0].push_back({1, 2});
    adj[1].push_back({0, 2});

    adj[0].push_back({3, 6});
    adj[3].push_back({0, 6});

    adj[1].push_back({2, 3});
    adj[2].push_back({1, 3});

    adj[1].push_back({3, 8});
    adj[3].push_back({1, 8});

    adj[1].push_back({4, 5});
    adj[4].push_back({1, 5});

    adj[2].push_back({4, 7});
```

```
    adj[4].push_back({2, 7});

    primMST(n, adj);
}
```

Kruskal’s Algorithm

- Sorts all edges by weight, then adds them one by one as long as they do not form a cycle.
- Uses **Disjoint Set Union (DSU)** to detect and prevent cycles.
- Time complexity: **O(E log E)**.
- More efficient for **sparse graphs**.

```
#include <bits/stdc++.h>
using namespace std;

struct Edge {
    int u, v, w;
    bool operator<(const Edge &other) const {
        return w < other.w;
    }
};

struct DSU {
    vector<int> parent, rank;
    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        iota(parent.begin(), parent.end(), 0);
    }

    int find(int x) {
        if (x != parent[x])
            parent[x] = find(parent[x]);
        return parent[x];
    }

    bool unite(int a, int b) {
        a = find(a);
        b = find(b);
        if (a == b) return false;
```

```
        if (rank[a] < rank[b]) swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b]) rank[a]++;
        return true;
    }
};

void kruskalMST(int n, vector<Edge> &edges) {
    sort(edges.begin(), edges.end());
    DSU dsu(n);

    cout << "Edges in MST (Kruskal):\n";
    for (auto &e : edges) {
        if (dsu.unite(e.u, e.v)) {
            cout << e.u << " - " << e.v << "\n";
        }
    }
}

int main() {
    int n = 5;
    vector<Edge> edges = {
        {0, 1, 2}, {0, 3, 6}, {1, 2, 3},
        {1, 3, 8}, {1, 4, 5}, {2, 4, 7}
    };

    kruskalMST(n, edges);
}
```

2.2 Example of Graph Traversal

Example of an undirected weighted graph:

image

- **BFS (starting from node 0)**
  Traversal per level (broad):
  0 → 1 → 2 → 3

- **DFS (starting from node 0)**
  Traversal as deep as possible before backtracking:
  0 → 1 → 3 → 2
  (order may vary depending on implementation, e.g., 0 → 1 → 2 → 3)
- **Dijkstra (shortest path from node 0)**
  - Distance to **0** = 0
  - Distance to **1** = 4
  - Distance to **2** = 1
  - Distance to **3** = 6