1. MST

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

// Function to implement Dijkstra's Algorithm

void dijkstra(int n, int src, const vector<vector<pair<int, int>>>& adj) {

    vector<int> dist(n, INT\_MAX); // Distance array to store the shortest distance from source

    dist[src] = 0;

    // Min-Heap: {distance, node}

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    pq.push({0, src});

    while (!pq.empty()) {

        int currDist = pq.top().first;

        int u = pq.top().second;

        pq.pop();

        if (currDist > dist[u]) continue; // Skip if the distance is already updated

        for (const auto& edge : adj[u]) {

            int v = edge.first;

            int weight = edge.second;

            if (dist[u] + weight < dist[v]) {

                dist[v] = dist[u] + weight;

                pq.push({dist[v], v});

            }

        }

    }

    // Print the shortest distances

    cout << "Shortest distances from node " << src << ":\n";

    for (int i = 0; i < n; ++i) {

        cout << "Node " << i << " : " << (dist[i] == INT\_MAX ? -1 : dist[i]) << endl;

    }

}

int main() {

    int n = 6; // Number of nodes

    vector<vector<pair<int, int>>> adj(n);

    // Add edges to the adjacency list (directed graph)

    adj[0].push\_back({1, 4});

    adj[0].push\_back({2, 4});

    adj[1].push\_back({2, 2});

    adj[1].push\_back({3, 5});

    adj[2].push\_back({3, 8});

    adj[2].push\_back({4, 4});

    adj[3].push\_back({5, 6});

    adj[4].push\_back({3, 1});

    adj[4].push\_back({5, 2});

    // Run Dijkstra's algorithm from source node 0

    dijkstra(n, 0, adj);

    return 0;

}

1. BFS and DFS

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

void bfs(int start, const vector<vector<int>>& adj, vector<bool>& visited) {

    queue<int> q;

    q.push(start);

    visited[start] = true;

    cout << "BFS Traversal: ";

    while (!q.empty()) {

        int node = q.front();

        q.pop();

        cout << node << " ";

        for (int neighbor : adj[node]) {

            if (!visited[neighbor]) {

                visited[neighbor] = true;

                q.push(neighbor);

            }

        }

    }

    cout << endl;

}

void dfs(int node, const vector<vector<int>>& adj, vector<bool>& visited) {

    visited[node] = true;

    cout << node << " ";

    for (int neighbor : adj[node]) {

        if (!visited[neighbor]) {

            dfs(neighbor, adj, visited);

        }

    }

}

int main() {

    int n = 5; // Number of nodes

    vector<vector<int>> adj = {

        {},           // Node 0 (no edges)

        {2, 3},       // Node 1

        {1, 4},       // Node 2

        {1, 5},       // Node 3

        {2},          // Node 4

        {3}           // Node 5

    };

    vector<bool> visited(n + 1, false);

    // Perform BFS

    bfs(1, adj, visited);

    // Reset visited for DFS

    fill(visited.begin(), visited.end(), false);

    // Perform DFS

    cout << "DFS Traversal: ";

    dfs(1, adj, visited);

    cout << endl;

    return 0;

}

1. Dijkstra

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

void bfs(int start, const vector<vector<int>>& adj, vector<bool>& visited) {

    queue<int> q;

    q.push(start);

    visited[start] = true;

    cout << "BFS Traversal: ";

    while (!q.empty()) {

        int node = q.front();

        q.pop();

        cout << node << " ";

        for (int neighbor : adj[node]) {

            if (!visited[neighbor]) {

                visited[neighbor] = true;

                q.push(neighbor);

            }

        }

    }

    cout << endl;

}

void dfs(int node, const vector<vector<int>>& adj, vector<bool>& visited) {

    visited[node] = true;

    cout << node << " ";

    for (int neighbor : adj[node]) {

        if (!visited[neighbor]) {

            dfs(neighbor, adj, visited);

        }

    }

}

int main() {

    int n = 5; // Number of nodes

    vector<vector<int>> adj = {

        {},           // Node 0 (no edges)

        {2, 3},       // Node 1

        {1, 4},       // Node 2

        {1, 5},       // Node 3

        {2},          // Node 4

        {3}           // Node 5

    };

    vector<bool> visited(n + 1, false);

    // Perform BFS

    bfs(1, adj, visited);

    // Reset visited for DFS

    fill(visited.begin(), visited.end(), false);

    // Perform DFS

    cout << "DFS Traversal: ";

    dfs(1, adj, visited);

    cout << endl;

    return 0;

}

1. Prims

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

// Function to implement Prim's Algorithm

void primsAlgorithm(int n, const vector<vector<pair<int, int>>>& adj) {

    vector<int> key(n, INT\_MAX); // Store the minimum weight edge for each node

    vector<int> parent(n, -1);  // Store the parent of each node in the MST

    vector<bool> inMST(n, false); // Mark nodes already included in the MST

    priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

    key[0] = 0;

    pq.push({0, 0}); // {key, node}

    while (!pq.empty()) {

        int u = pq.top().second;

        pq.pop();

        if (inMST[u]) continue; // Skip if already in MST

        inMST[u] = true;

        for (const auto& edge : adj[u]) { // Replace structured bindings

            int v = edge.first;

            int weight = edge.second;

            if (!inMST[v] && weight < key[v]) {

                key[v] = weight;

                parent[v] = u;

                pq.push({key[v], v});

            }

        }

    }

    // Print the edges in the MST

    cout << "Edges in the MST:\n";

    for (int i = 1; i < n; ++i) {

        cout << parent[i] << " - " << i << "\n";

    }

}

int main() {

    int n = 5; // Number of nodes

    vector<vector<pair<int, int>>> adj(n);

    // Add edges to the adjacency list (undirected graph)

    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});

    // Run Prim's algorithm

    primsAlgorithm(n, adj);

    return 0;

}

1. Kruskals

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Structure to represent an edge

struct Edge {

    int u, v, weight;

    bool operator<(const Edge& other) const {

        return weight < other.weight;

    }

};

// Find function for the Disjoint Set Union (DSU)

int findParent(int node, vector<int>& parent) {

    if (node != parent[node]) {

        parent[node] = findParent(parent[node], parent); // Path compression

    }

    return parent[node];

}

// Union function for the DSU

void unionNodes(int u, int v, vector<int>& parent, vector<int>& rank) {

    int pu = findParent(u, parent);

    int pv = findParent(v, parent);

    if (pu != pv) {

        if (rank[pu] < rank[pv]) {

            parent[pu] = pv;

        } else if (rank[pu] > rank[pv]) {

            parent[pv] = pu;

        } else {

            parent[pv] = pu;

            rank[pu]++;

        }

    }

}

// Kruskal's algorithm

void kruskal(int n, vector<Edge>& edges) {

    // Sort edges by weight

    sort(edges.begin(), edges.end());

    vector<int> parent(n);

    vector<int> rank(n, 0);

    // Initialize DSU

    for (int i = 0; i < n; ++i) {

        parent[i] = i;

    }

    vector<Edge> mst; // Store edges in the MST

    int mstWeight = 0;

    for (const Edge& edge : edges) {

        if (findParent(edge.u, parent) != findParent(edge.v, parent)) {

            mst.push\_back(edge);

            mstWeight += edge.weight;

            unionNodes(edge.u, edge.v, parent, rank);

        }

    }

    // Print the MST and its weight

    cout << "Edges in the MST:\n";

    for (const Edge& edge : mst) {

        cout << edge.u << " - " << edge.v << " (Weight: " << edge.weight << ")\n";

    }

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

}

int main() {

    int n = 5; // Number of nodes

    vector<Edge> edges = {

        {0, 1, 2},

        {0, 3, 6},

        {1, 2, 3},

        {1, 3, 8},

        {1, 4, 5},

        {2, 4, 7}

    };

    // Run Kruskal's algorithm

    kruskal(n, edges);

    return 0;

}