Phần C:

16.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class DisjointSet {

public:

vector<int> parent, rank;

DisjointSet(int n) {

parent.resize(n);

rank.resize(n, 0);

for (int i = 0; i < n; ++i) {

parent[i] = i;

}

}

int find(int u) {

if (parent[u] != u) {

parent[u] = find(parent[u]);

}

return parent[u];

}

void unite(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

} else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

};

vector<pair<int, pair<int, int>>> kruskal\_forest(vector<pair<int, pair<int, int>>> edges, int n) {

vector<pair<int, pair<int, int>>> minimum\_forest;

DisjointSet ds(n);

sort(edges.begin(), edges.end());

for (auto edge : edges) {

int weight = edge.first;

int u = edge.second.first;

int v = edge.second.second;

if (ds.find(u) != ds.find(v)) {

ds.unite(u, v);

minimum\_forest.push\_back(edge);

}

}

return minimum\_forest;

}

int main() {

// Example graph represented as an adjacency list

vector<pair<int, pair<int, int>>> edges = {

{2, {0, 1}},

{3, {0, 2}},

{5, {1, 2}},

{4, {1, 3}},

{6, {2, 3}}

};

int num\_vertices = 4;

vector<pair<int, pair<int, int>>> minimum\_forest = kruskal\_forest(edges, num\_vertices);

// Output the minimum spanning forest

cout << "Minimum Spanning Forest:\n";

for (auto edge : minimum\_forest) {

int weight = edge.first;

int u = edge.second.first;

int v = edge.second.second;

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

}

return 0;

}

17.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class DisjointSet {

public:

vector<int> parent, rank;

DisjointSet(int n) {

parent.resize(n);

rank.resize(n, 0);

for (int i = 0; i < n; ++i) {

parent[i] = i;

}

}

int find(int u) {

if (parent[u] != u) {

parent[u] = find(parent[u]);

}

return parent[u];

}

void unite(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

} else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

};

vector<pair<int, pair<int, int>>> vyssotsky\_algorithm(vector<pair<int, pair<int, int>>> edges, int n) {

vector<pair<int, pair<int, int>>> minimum\_spanning\_tree;

sort(edges.begin(), edges.end(), greater<>()); // Sort edges in descending order by weight

DisjointSet ds(n);

for (auto edge : edges) {

int weight = edge.first;

int u = edge.second.first;

int v = edge.second.second;

int rootU = ds.find(u);

int rootV = ds.find(v);

if (rootU != rootV) {

ds.unite(rootU, rootV);

minimum\_spanning\_tree.push\_back(edge);

}

}

return minimum\_spanning\_tree;

}

int main() {

// Example graph represented as an adjacency list

vector<pair<int, pair<int, int>>> edges = {

{2, {0, 1}},

{3, {0, 2}},

{5, {1, 2}},

{4, {1, 3}},

{6, {2, 3}}

};

int num\_vertices = 4;

vector<pair<int, pair<int, int>>> minimum\_spanning\_tree = vyssotsky\_algorithm(edges, num\_vertices);

// Output the minimum spanning tree

cout << "Minimum Spanning Tree:\n";

for (auto edge : minimum\_spanning\_tree) {

int weight = edge.first;

int u = edge.second.first;

int v = edge.second.second;

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

}

return 0;

}

18.

Chứng minh:

1. **Đảm bảo là cây bao trùm:** Ở mỗi bước, chúng ta xóa cạnh có trọng số lớn nhất. Nếu cạnh này không phải là cạnh của cây bao trùm, có thể chứng minh rằng nó không tạo thành chu trình với các cạnh còn lại trong cây bao trùm. Do đó, sau mỗi bước, chúng ta vẫn giữ được tính chất không tạo chu trình.
2. **Bảo toàn liên thông:** Nếu cạnh được xóa không phải là cạnh của cây bao trùm, thì việc xóa nó không làm mất liên thông của đồ thị. Điều này là vì đồ thị vẫn chứa một đường đi giữa bất kỳ cặp đỉnh nào khác nhau.
3. **Kết quả là cây bao trùm:** Sau khi xóa một số cạnh, chúng ta sẽ có một đồ thị liên thông, và tất cả các đỉnh đều được kết nối với nhau thông qua các cạnh còn lại. Do đó, kết quả là một cây bao trùm.

Độ phức tạp:

* **Sắp xếp cạnh:** O(E log E), với E là số cạnh.
* **Kiểm tra liên thông và xóa cạnh:** O(E α(V)), với α là hàm nghịch đảo Ackermann và V là số đỉnh.

Vì số bước so sánh trọng số cạnh là độ phức tạp chính của thuật toán, và chúng ta có E cạnh, nên độ phức tạp cuối cùng của thuật toán Reverse-Delete là O(E log E + E α(V)).

19.

#include <iostream>

#include <vector>

#include <algorithm>

#include <tuple>

using namespace std;

class DisjointSet {

public:

vector<int> parent, rank;

DisjointSet(int n) {

parent.resize(n);

rank.resize(n, 0);

for (int i = 0; i < n; ++i) {

parent[i] = i;

}

}

int find(int u) {

if (parent[u] != u) {

parent[u] = find(parent[u]);

}

return parent[u];

}

void unite(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

} else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

};

vector<tuple<int, int, int>> find\_critical\_edges(vector<tuple<int, int, int>> edges, int num\_vertices) {

sort(edges.begin(), edges.end(), greater<>()); // Sort edges in descending order by weight

DisjointSet ds(num\_vertices);

vector<int> weight\_index(num\_vertices, -1);

vector<tuple<int, int, int>> critical\_edges;

for (int i = 0; i < edges.size(); ++i) {

int u = get<0>(edges[i]);

int v = get<1>(edges[i]);

int weight = get<2>(edges[i]);

int rootU = ds.find(u);

int rootV = ds.find(v);

if (rootU != rootV) {

ds.unite(rootU, rootV);

} else {

// u and v are in the same connected component, this edge is not in the MST

// However, it is critical if it has the maximum weight among edges between the two components

critical\_edges.push\_back(edges[i]);

continue;

}

// Update the weight index of the component root

if (weight\_index[rootU] == -1 || weight > get<2>(edges[weight\_index[rootU]])) {

weight\_index[rootU] = i;

}

if (weight\_index[rootV] == -1 || weight > get<2>(edges[weight\_index[rootV]])) {

weight\_index[rootV] = i;

}

}

return critical\_edges;

}

int main() {

// Example graph represented as an adjacency list

vector<tuple<int, int, int>> edges = {

{0, 1, 2},

{0, 2, 3},

{1, 2, 5},

{1, 3, 4},

{2, 3, 6}

};

int num\_vertices = 4;

vector<tuple<int, int, int>> critical\_edges = find\_critical\_edges(edges, num\_vertices);

// Output the critical edges

cout << "Critical Edges:\n";

for (auto edge : critical\_edges) {

int u = get<0>(edge);

int v = get<1>(edge);

int weight = get<2>(edge);

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

}

return 0;

}

20. #include <SFML/Graphics.hpp>

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <thread>

using namespace std;

struct Edge {

int u, v, weight;

bool isInMST;

Edge(int u, int v, int weight) : u(u), v(v), weight(weight), isInMST(false) {}

};

class DisjointSet {

public:

vector<int> parent, rank;

DisjointSet(int n) {

parent.resize(n);

rank.resize(n, 0);

for (int i = 0; i < n; ++i) {

parent[i] = i;

}

}

int find(int u) {

if (parent[u] != u) {

parent[u] = find(parent[u]);

}

return parent[u];

}

void unite(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

} else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

};

class GraphVisualization {

private:

int numVertices;

vector<Edge> edges;

vector<sf::CircleShape> vertices;

sf::RenderWindow window;

public:

GraphVisualization(int numVertices, const vector<Edge>& edges)

: numVertices(numVertices), edges(edges) {

initializeWindow();

initializeVertices();

}

void initializeWindow() {

window.create(sf::VideoMode(800, 600), "Minimum Spanning Tree Visualization");

}

void initializeVertices() {

for (int i = 0; i < numVertices; ++i) {

sf::CircleShape vertex(20);

vertex.setPosition(50 + i \* 100, 300);

vertex.setFillColor(sf::Color::Blue);

vertices.push\_back(vertex);

}

}

void drawGraph() {

window.clear(sf::Color::White);

// Draw edges

for (const Edge& edge : edges) {

sf::Vertex line[] = {

sf::Vertex(sf::Vector2f(vertices[edge.u].getPosition().x + 20, vertices[edge.u].getPosition().y + 20)),

sf::Vertex(sf::Vector2f(vertices[edge.v].getPosition().x + 20, vertices[edge.v].getPosition().y + 20))

};

if (edge.isInMST) {

window.draw(line, 2, sf::Lines, sf::Color::Green);

} else {

window.draw(line, 2, sf::Lines, sf::Color::Black);

}

}

// Draw vertices

for (const sf::CircleShape& vertex : vertices) {

window.draw(vertex);

}

window.display();

}

void visualize() {

window.setFramerateLimit(5);

DisjointSet ds(numVertices);

for (const Edge& edge : edges) {

if (ds.find(edge.u) != ds.find(edge.v)) {

ds.unite(edge.u, edge.v);

edge.isInMST = true;

}

drawGraph();

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

}

window.setFramerateLimit(60);

while (window.isOpen()) {

sf::Event event;

while (window.pollEvent(event)) {

if (event.type == sf::Event::Closed) {

window.close();

}

}

}

}

};

int main() {

int numVertices = 4;

vector<Edge> edges = {

{0, 1, 2},

{0, 2, 3},

{1, 2, 5},

{1, 3, 4},

{2, 3, 6}

};

GraphVisualization graphVisualization(numVertices, edges);

graphVisualization.visualize();

return 0;

}

21.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Edge {

int u, v, weight;

Edge(int u, int v, int weight) : u(u), v(v), weight(weight) {}

};

class DisjointSet {

public:

vector<int> parent, rank;

DisjointSet(int n) {

parent.resize(n);

rank.resize(n, 0);

for (int i = 0; i < n; ++i) {

parent[i] = i;

}

}

int find(int u) {

if (parent[u] != u) {

parent[u] = find(parent[u]);

}

return parent[u];

}

void unite(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

} else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

};

vector<Edge> find\_min\_spanning\_tree(const vector<Edge>& graph\_edges, const vector<Edge>& set\_edges, int num\_vertices) {

vector<Edge> min\_spanning\_tree;

DisjointSet ds(num\_vertices);

// Sort all edges in non-decreasing order of weight

vector<Edge> all\_edges = graph\_edges;

all\_edges.insert(all\_edges.end(), set\_edges.begin(), set\_edges.end());

sort(all\_edges.begin(), all\_edges.end(), [](const Edge& a, const Edge& b) {

return a.weight < b.weight;

});

for (const Edge& edge : all\_edges) {

int rootU = ds.find(edge.u);

int rootV = ds.find(edge.v);

// Check if adding this edge creates a cycle

if (rootU != rootV) {

// Add the edge to the minimum spanning tree

min\_spanning\_tree.push\_back(edge);

ds.unite(rootU, rootV);

}

// Stop when the MST has (num\_vertices - 1) edges

if (min\_spanning\_tree.size() == num\_vertices - 1) {

break;

}

}

return min\_spanning\_tree;

}

int main() {

int num\_vertices = 4;

vector<Edge> graph\_edges = {

{0, 1, 2},

{0, 2, 3},

{1, 2, 5},

{1, 3, 4},

{2, 3, 6}

};

vector<Edge> set\_edges = {

{0, 1, 2},

{1, 2, 5}

};

vector<Edge> min\_spanning\_tree = find\_min\_spanning\_tree(graph\_edges, set\_edges, num\_vertices);

// Output the minimum spanning tree

cout << "Minimum Spanning Tree:\n";

for (const Edge& edge : min\_spanning\_tree) {

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

}

return 0;

}

22.

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <queue>

using namespace std;

using namespace chrono;

struct Edge {

int u, v, weight;

Edge(int u, int v, int weight) : u(u), v(v), weight(weight) {}

};

class DisjointSet {

public:

vector<int> parent, rank;

DisjointSet(int n) {

parent.resize(n);

rank.resize(n, 0);

for (int i = 0; i < n; ++i) {

parent[i] = i;

}

}

int find(int u) {

if (parent[u] != u) {

parent[u] = find(parent[u]);

}

return parent[u];

}

void unite(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

} else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

};

vector<Edge> prim\_lazy(const vector<vector<pair<int, int>>>& graph) {

int num\_vertices = graph.size();

vector<bool> visited(num\_vertices, false);

vector<Edge> min\_spanning\_tree;

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

// Start from vertex 0

pq.push({0, 0});

while (!pq.empty()) {

int u = pq.top().second;

int weight = pq.top().first;

pq.pop();

if (visited[u]) {

continue;

}

visited[u] = true;

for (const auto& neighbor : graph[u]) {

int v = neighbor.first;

int edge\_weight = neighbor.second;

if (!visited[v]) {

pq.push({edge\_weight, v});

min\_spanning\_tree.push\_back(Edge(u, v, edge\_weight));

}

}

}

return min\_spanning\_tree;

}

vector<Edge> prim\_queue(const vector<vector<pair<int, int>>>& graph) {

int num\_vertices = graph.size();

vector<bool> visited(num\_vertices, false);

vector<Edge> min\_spanning\_tree;

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

// Start from vertex 0

pq.push({0, {0, 0}});

while (!pq.empty()) {

int u = pq.top().second.second;

int weight = pq.top().first;

pq.pop();

if (visited[u]) {

continue;

}

visited[u] = true;

for (const auto& neighbor : graph[u]) {

int v = neighbor.first;

int edge\_weight = neighbor.second;

if (!visited[v]) {

pq.push({edge\_weight, {u, v}});

min\_spanning\_tree.push\_back(Edge(u, v, edge\_weight));

}

}

}

return min\_spanning\_tree;

}

vector<Edge> kruskal\_lazy(const vector<Edge>& graph\_edges, int num\_vertices) {

vector<Edge> min\_spanning\_tree;

DisjointSet ds(num\_vertices);

sort(graph\_edges.begin(), graph\_edges.end(), [](const Edge& a, const Edge& b) {

return a.weight < b.weight;

});

for (const Edge& edge : graph\_edges) {

int rootU = ds.find(edge.u);

int rootV = ds.find(edge.v);

if (rootU != rootV) {

min\_spanning\_tree.push\_back(edge);

ds.unite(rootU, rootV);

}

if (min\_spanning\_tree.size() == num\_vertices - 1) {

break;

}

}

return min\_spanning\_tree;

}

vector<Edge> kruskal\_queue(const vector<Edge>& graph\_edges, int num\_vertices) {

vector<Edge> min\_spanning\_tree;

DisjointSet ds(num\_vertices);

priority\_queue<Edge, vector<Edge>, greater<Edge>> pq(graph\_edges.begin(), graph\_edges.end());

while (!pq.empty()) {

Edge edge = pq.top();

pq.pop();

int rootU = ds.find(edge.u);

int rootV = ds.find(edge.v);

if (rootU != rootV) {

min\_spanning\_tree.push\_back(edge);

ds.unite(rootU, rootV);

}

if (min\_spanning\_tree.size() == num\_vertices - 1) {

break;

}

}

return min\_spanning\_tree;

}

void measure\_time(const vector<vector<pair<int, int>>>& graph) {

int num\_vertices = graph.size();

auto start\_prim\_lazy = high\_resolution\_clock::now();

prim\_lazy(graph);

auto stop\_prim\_lazy = high\_resolution\_clock::now();

auto duration\_prim\_lazy = duration\_cast<microseconds>(stop\_prim\_lazy - start\_prim\_lazy);

cout << "Prim (Lazy): " << duration\_prim\_lazy.count() << " microseconds\n";

auto start\_prim\_queue = high\_resolution\_clock::now();

prim\_queue(graph);

auto stop\_prim\_queue = high\_resolution\_clock::now();

auto duration\_prim\_queue = duration\_cast<microseconds>(stop\_prim\_queue - start\_prim\_queue);

cout << "Prim (Queue): " << duration\_prim\_queue.count() << " microseconds\n";

vector<Edge> graph\_edges;

for (int u = 0; u < num\_vertices; ++u) {

for (const auto& neighbor : graph[u]) {

int v = neighbor.first;

int weight = neighbor.second;

if (u < v) {

graph\_edges.push\_back(Edge(u, v, weight));

}

}

}

auto start\_kruskal\_lazy = high\_resolution\_clock::now();

kruskal\_lazy(graph\_edges, num\_vertices);

auto stop\_kruskal\_lazy = high\_resolution\_clock::now();

auto duration\_kruskal\_lazy = duration\_cast<microseconds>(stop\_kruskal\_lazy - start\_kruskal\_lazy);

cout << "Kruskal (Lazy): " << duration\_kruskal\_lazy.count() << " microseconds\n";

auto start\_kruskal\_queue = high\_resolution\_clock::now();

kruskal\_queue(graph\_edges, num\_vertices);

auto stop\_kruskal\_queue = high\_resolution\_clock::now();

auto duration\_kruskal\_queue = duration\_cast<microseconds>(stop\_kruskal\_queue - start\_kruskal\_queue);

cout << "Kruskal (Queue): " << duration\_kruskal\_queue.count() << " microseconds\n";

}

int main() {

// Example graph represented as an adjacency list

vector<vector<pair<int, int>>> graph = {

{{1, 2}, {2, 3}},

{{0, 2}, {2, 5}, {3, 4}},

{{0, 3}, {1, 5}, {4, 6}},

{{1, 4}, {2, 6}},

{{3, 6}},

{{1, 4}, {2, 6}},

{{3, 6}}

};

measure\_time(graph);

return 0;

}

23.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Edge {

int u, v, weight;

Edge(int u, int v, int weight) : u(u), v(v), weight(weight) {}

};

class DisjointSet {

public:

vector<int> parent, rank;

DisjointSet(int n) {

parent.resize(n);

rank.resize(n, 0);

for (int i = 0; i < n; ++i) {

parent[i] = i;

}

}

int find(int u) {

if (parent[u] != u) {

parent[u] = find(parent[u]);

}

return parent[u];

}

void unite(int u, int v) {

int rootU = find(u);

int rootV = find(v);

if (rootU != rootV) {

if (rank[rootU] > rank[rootV]) {

parent[rootV] = rootU;

} else if (rank[rootU] < rank[rootV]) {

parent[rootU] = rootV;

} else {

parent[rootV] = rootU;

rank[rootU]++;

}

}

}

};

vector<Edge> boruvka(vector<Edge>& edges, int num\_vertices) {

vector<Edge> min\_spanning\_tree;

DisjointSet ds(num\_vertices);

while (min\_spanning\_tree.size() < num\_vertices - 1) {

// For each component, find the minimum edge that connects it to another component

vector<Edge> cheapest\_edge(num\_vertices, Edge(-1, -1, INT\_MAX));

for (const Edge& edge : edges) {

int rootU = ds.find(edge.u);

int rootV = ds.find(edge.v);

if (rootU != rootV) {

if (edge.weight < cheapest\_edge[rootU].weight) {

cheapest\_edge[rootU] = edge;

}

if (edge.weight < cheapest\_edge[rootV].weight) {

cheapest\_edge[rootV] = edge;

}

}

}

// Add the cheapest edges to the minimum spanning tree

for (int i = 0; i < num\_vertices; ++i) {

if (cheapest\_edge[i].weight != INT\_MAX) {

int rootU = ds.find(cheapest\_edge[i].u);

int rootV = ds.find(cheapest\_edge[i].v);

if (rootU != rootV) {

min\_spanning\_tree.push\_back(cheapest\_edge[i]);

ds.unite(rootU, rootV);

}

}

}

}

return min\_spanning\_tree;

}

int main() {

int num\_vertices = 4;

vector<Edge> edges = {

{0, 1, 2},

{0, 2, 3},

{1, 2, 5},

{1, 3, 4},

{2, 3, 6}

};

vector<Edge> min\_spanning\_tree = boruvka(edges, num\_vertices);

// Output the minimum spanning tree

cout << "Minimum Spanning Tree:\n";

for (const Edge& edge : min\_spanning\_tree) {

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

}

return 0;

}

24.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Edge {

int u, v, weight;

Edge(int u, int v, int weight) : u(u), v(v), weight(weight) {}

};

struct Node {

int parent;

Edge edge;

Node() : parent(-1) {}

};

class BoruvkaImproved {

private:

vector<Node> nodes;

vector<Edge> cheapestEdge;

public:

BoruvkaImproved(int num\_vertices) {

nodes.resize(num\_vertices);

cheapestEdge.resize(num\_vertices);

for (int i = 0; i < num\_vertices; ++i) {

cheapestEdge[i].weight = INT\_MAX;

}

}

void addEdge(int u, int v, int weight) {

if (weight < cheapestEdge[u].weight) {

cheapestEdge[u] = Edge(u, v, weight);

}

if (weight < cheapestEdge[v].weight) {

cheapestEdge[v] = Edge(u, v, weight);

}

}

void mergeTrees(int u, int v) {

nodes[u].parent = v;

}

vector<Edge> boruvka(const vector<Edge>& edges) {

vector<Edge> min\_spanning\_tree;

int num\_vertices = nodes.size();

while (min\_spanning\_tree.size() < num\_vertices - 1) {

for (int i = 0; i < num\_vertices; ++i) {

if (cheapestEdge[i].weight != INT\_MAX) {

int rootU = findRoot(i);

int rootV = findRoot(cheapestEdge[i].v);

if (rootU != rootV) {

min\_spanning\_tree.push\_back(cheapestEdge[i]);

mergeTrees(rootU, rootV);

}

}

}

for (int i = 0; i < num\_vertices; ++i) {

cheapestEdge[i].weight = INT\_MAX;

}

for (const Edge& edge : edges) {

int rootU = findRoot(edge.u);

int rootV = findRoot(edge.v);

if (rootU != rootV) {

if (edge.weight < cheapestEdge[rootU].weight) {

cheapestEdge[rootU] = edge;

}

if (edge.weight < cheapestEdge[rootV].weight) {

cheapestEdge[rootV] = edge;

}

}

}

}

return min\_spanning\_tree;

}

private:

int findRoot(int u) {

while (nodes[u].parent != -1) {

u = nodes[u].parent;

}

return u;

}

};

int main() {

int num\_vertices = 4;

vector<Edge> edges = {

{0, 1, 2},

{0, 2, 3},

{1, 2, 5},

{1, 3, 4},

{2, 3, 6}

};

BoruvkaImproved boruvka(num\_vertices);

for (const Edge& edge : edges) {

boruvka.addEdge(edge.u, edge.v, edge.weight);

}

vector<Edge> min\_spanning\_tree = boruvka.boruvka(edges);

// Output the minimum spanning tree

cout << "Minimum Spanning Tree:\n";

for (const Edge& edge : min\_spanning\_tree) {

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

}

return 0;

}

25.

#include <iostream>

#include <vector>

#include <algorithm>

#include <set>

using namespace std;

struct Edge {

int u, v, weight;

Edge(int u, int v, int weight) : u(u), v(v), weight(weight) {}

};

vector<Edge> findOptimalEdges(const vector<Edge>& edges, int maxSize) {

vector<Edge> optimalEdges;

// Sort edges by weight in ascending order

vector<Edge> sortedEdges = edges;

sort(sortedEdges.begin(), sortedEdges.end(), [](const Edge& a, const Edge& b) {

return a.weight < b.weight;

});

// Select edges until reaching maxSize

set<int> vertices;

for (const Edge& edge : sortedEdges) {

vertices.insert(edge.u);

vertices.insert(edge.v);

if (vertices.size() <= maxSize) {

optimalEdges.push\_back(edge);

} else {

break;

}

}

return optimalEdges;

}

vector<Edge> primMinSpanningTree(const vector<Edge>& edges, int maxSize) {

vector<Edge> minSpanningTree;

int numVertices = 0;

while (numVertices < edges.size()) {

// Select a subset of edges to process

vector<Edge> subsetEdges = findOptimalEdges(edges, maxSize);

// Run Prim's algorithm on the subset

vector<bool> visited(numVertices + maxSize, false);

for (const Edge& edge : subsetEdges) {

visited[edge.u] = visited[edge.v] = true;

}

// Continue Prim's algorithm until all vertices are visited

while (count(visited.begin(), visited.end(), true) < numVertices + maxSize) {

Edge minEdge = { -1, -1, INT\_MAX };

// Find the minimum edge that connects visited and unvisited vertices

for (const Edge& edge : edges) {

if ((visited[edge.u] && !visited[edge.v]) || (!visited[edge.u] && visited[edge.v])) {

if (edge.weight < minEdge.weight) {

minEdge = edge;

}

}

}

// Add the minimum edge to the minimum spanning tree

minSpanningTree.push\_back(minEdge);

visited[minEdge.u] = visited[minEdge.v] = true;

}

// Update the number of vertices processed

numVertices += maxSize;

}

return minSpanningTree;

}

int main() {

// Example: Constructing a large graph

int numVertices = 10000;

vector<Edge> edges;

// Populate edges with some data (replace this with your data source)

for (int i = 0; i < numVertices - 1; ++i) {

edges.push\_back(Edge(i, i + 1, rand() % 100));

}

// Run the Prim's algorithm with edge selection

int maxSize = 100; // Adjust the maximum size based on available memory

vector<Edge> minSpanningTree = primMinSpanningTree(edges, maxSize);

// Output the minimum spanning tree

cout << "Minimum Spanning Tree:\n";

for (const Edge& edge : minSpanningTree) {

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

}

return 0;

}