**1. Graph Data Structure:**

* **Node:**
  + id: Unique identifier for the node.
  + data: Optional data associated with the node.
  + edges: List of outgoing edges from this node.
* **Edge:**
  + source: Starting node.
  + target: Ending node.
  + weight: Weight of the edge.
* **Graph:**
  + nodes: List of nodes in the graph.
  + edges: List of edges in the graph.
  + num\_vertices: Number of vertices in the graph.
  + num\_edges: Number of edges in the graph.

**2. Tree Data Structure:**

* **Node:**
  + id: Unique identifier for the node.
  + parent: Parent node in the tree.
  + rank: Rank of the node in the disjoint-set forest (used in Kruskal's algorithm).
* **Tree:**
  + nodes: List of nodes in the tree.
  + num\_vertices: Number of vertices in the tree.

**3. Factory for MST Algorithms:**

* **MSTFactory:**
  + create\_mst\_solver(algorithm\_name): Creates an instance of the specified MST algorithm solver.

**4. MST Algorithm Solvers:**

* **Borůvka:**
  + Find the minimum outgoing edge for each vertex.
  + Merge vertices connected by these edges into a single component.
  + Repeat until all vertices are in the same component.
* **Prim:**
  + Start with a single vertex.
  + Find the minimum-weight edge connecting the current tree to a vertex outside the tree.
  + Add the edge and its corresponding vertex to the tree.
  + Repeat until all vertices are in the tree.
* **Kruskal:**
  + Sort the edges by weight in ascending order.
  + Iterate through the edges, adding each edge to the MST if it doesn't create a cycle.
  + Use a disjoint-set forest to efficiently check for cycles.
* **Tarjan:**
  + Find the strongly connected components (SCCs) of the graph.
  + For each SCC, find the minimum outgoing edge.
  + Add these edges to the MST.

**5. Calculating MST Properties:**

* **Total weight:** Sum the weights of all edges in the MST.
* **Longest distance:** Find the maximum weight of any edge in the MST.
* **Average distance:** Iterate through all pairs of vertices (i, j), calculate the distance using the MST, and compute the average.
* **Shortest distance:** Iterate through all pairs of vertices (i, j), find the shortest path using the MST, and record the minimum distance.

i have code in c++ that take graph and find the mst. ill send you my code and you will help me with some tasks. main.cpp: #include <iostream>

#include <vector>

#include <list>

#include <stack>

#include <algorithm>

#include <thread>

#include <sstream>

#include <cstring>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <mutex>

using namespace std;

// Define Edge structure

struct Edge {

    int u, v, weight;

};

// Global variables (consider thread safety if using multiple threads)

int n = 0, m = 0; // Number of vertices and edges

vector<Edge> edges; // List of edges

vector<vector<Edge>> adj; // Adjacency list representation of the graph

mutex graphMutex; // Mutex for thread safety

// Function prototypes (ensure they are defined or included)

vector<pair<int, int>> prim(int n, const vector<vector<Edge>>& adj);

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

// Function to handle client requests

void handleClient(int clientSocket) {

    char buffer[1024] = {0}; // Buffer to read client data

    string command; // Command from client

    while (true) {

        // Read data from the client

        int valread = read(clientSocket, buffer, 1024);

        if (valread <= 0) {

            // If no data is read or client disconnects, close the socket and return

            cout << "Client disconnected" << endl;

            close(clientSocket);

            return;

        }

        // Null-terminate the buffer to process as a string

        buffer[valread] = '\0';

        command = buffer;

        stringstream ss(command);

        ss >> command; // Extract the command from the stringstream

        if (command == "Newgraph") {

            // Handle initialization of a new graph

            lock\_guard<mutex> lock(graphMutex); // Lock the mutex for thread safety

            ss >> n >> m; // Read the number of vertices and edges

            edges.clear(); // Clear the existing edge list

            adj.assign(n + 1, vector<Edge>()); // Resize adjacency list to hold n vertices

            cout << "New graph initialized with n = " << n << " and m = " << m << endl;

        } else if (command == "MST") {

            // Handle Minimum Spanning Tree computation

            string algoType;

            ss >> algoType; // Read the type of MST algorithm

            auto algo = MSTFactory::createAlgorithm(algoType); // Create the algorithm object

            if (!algo) {

                // If the algorithm type is invalid, send an error message

                string error = "Invalid MST algorithm\n";

                send(clientSocket, error.c\_str(), error.length(), 0);

                continue; // Continue to handle next client request

            }

            vector<pair<int, int>> mstEdges;

            {

                lock\_guard<mutex> lock(graphMutex); // Lock the mutex for thread safety

                mstEdges = algo->compute(n, adj); // Compute the MST

            }

            // Prepare and send the MST result to the client

            stringstream response;

            response << "mst:\n";

            for (const auto& edge : mstEdges) {

                response << edge.first << " " << edge.second << "\n";

            }

            send(clientSocket, response.str().c\_str(), response.str().length(), 0);

        } else if (command == "Newedge") {

            // Handle adding a new edge

            int u, v, w;

            ss >> u >> v >> w; // Read edge details

            {

                lock\_guard<mutex> lock(graphMutex); // Lock the mutex for thread safety

                edges.push\_back({u, v, w}); // Add edge to edge list

                adj[u].push\_back({v, w}); // Add edge to adjacency list

                adj[v].push\_back({u, w}); // For undirected graph, add edge in both directions

            }

            cout << "Added new edge: (" << u << ", " << v << ") with weight " << w << endl;

        } else if (command == "Removeedge") {

            // Handle removing an edge

            int u, v;

            ss >> u >> v; // Read edge details

            {

                lock\_guard<mutex> lock(graphMutex); // Lock the mutex for thread safety

                // Remove edge from edge list

                edges.erase(remove\_if(edges.begin(), edges.end(), [u, v](const Edge& e) {

                    return (e.u == u && e.v == v) || (e.u == v && e.v == u);

                }), edges.end());

                // Remove edge from adjacency list

                adj[u].erase(remove\_if(adj[u].begin(), adj[u].end(), [v](const Edge& e) { return e.v == v; }), adj[u].end());

                adj[v].erase(remove\_if(adj[v].begin(), adj[v].end(), [u](const Edge& e) { return e.u == u; }), adj[v].end());

            }

            cout << "Removed edge: (" << u << ", " << v << ")" << endl;

        } else {

            // Handle invalid command

            string error = "Invalid command\n";

            send(clientSocket, error.c\_str(), error.length(), 0);

        }

    }

}

kruskal.cpp: #include <iostream>

#include <vector>

#include <algorithm>

#include "MSTAlgorithms.hpp"

using namespace std;

// Structure to represent an edge in the graph

struct Edge {

    int u, v, weight; // The vertices connected by the edge and its weight

    // Overload the less-than operator for sorting edges based on their weights

    bool operator<(const Edge& e) const {

        return weight < e.weight; // Compare edges by weight

    }

};

// Find the representative (or root) of the set containing vertex u

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

    if (parent[u] != u) {

        // Path compression: make the tree flatter for efficiency

        parent[u] = findParent(parent[u], parent);

    }

    return parent[u]; // Return the root of the set

}

// Union the sets containing vertices u and v

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

    int rootU = findParent(u, parent); // Find root of u

    int rootV = findParent(v, parent); // Find root of v

    if (rootU != rootV) {

        // Union by rank: attach the smaller tree under the larger tree

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

            parent[rootV] = rootU; // Attach rootV's tree under rootU's tree

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

            parent[rootU] = rootV; // Attach rootU's tree under rootV's tree

        } else {

            parent[rootV] = rootU; // Arbitrarily choose rootU as new root

            rank[rootU]++; // Increase the rank of the new root

        }

    }

}

// Kruskal's algorithm to find the Minimum Spanning Tree (MST)

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

    vector<Edge> mstEdges; // To store the edges of the MST

    vector<int> parent(n + 1), rank(n + 1, 0); // Initialize parent and rank for Union-Find

    iota(parent.begin(), parent.end(), 0); // Initialize parent array to self (each node is its own parent)

    sort(edges.begin(), edges.end()); // Sort all edges in non-decreasing order of their weight

    // Process each edge in sorted order

    for (const Edge& e : edges) {

        int u = e.u, v = e.v; // Get vertices of the edge

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

            // If u and v are not in the same set, include this edge in the MST

            unionSets(u, v, parent, rank); // Union the sets containing u and v

            mstEdges.push\_back(e); // Add edge to MST

        }

    }

    return mstEdges; // Return the edges of the MST

}

MSTAlgorithms.hpp: #ifndef MSTALGORITHMS\_H

#define MSTALGORITHMS\_H

#include <vector>

// Define the Edge structure

struct Edge {

    int u, v, weight;

};

// Function declarations

std::vector<Edge> prim(int n, const std::vector<std::vector<Edge>>& adj);

std::vector<Edge> kruskal(int n, std::vector<Edge>& edges);

#endif // MSTALGORITHMS\_H

MSTFactory.cpp: #include <memory>

#include <vector>

#include <string>

#include "MSTAlgorithms.hpp"

using namespace std;

// Define Edge structure (assumed to be defined elsewhere in your code)

struct Edge {

    int u, v, weight;

};

// Abstract base class for MST algorithms

class MSTAlgorithm {

public:

    // Pure virtual function to compute the MST

    // Subclasses must implement this function

    virtual vector<pair<int, int>> compute(int n, const vector<vector<Edge>>& adj) = 0;

};

// Implementation of Prim's algorithm for MST

class PrimAlgorithm : public MSTAlgorithm {

public:

    // Override the compute method to use Prim's algorithm

    vector<pair<int, int>> compute(int n, const vector<vector<Edge>>& adj) override {

        return prim(n, adj); // Call the Prim's algorithm function (assumed to be defined elsewhere)

    }

};

// Implementation of Kruskal's algorithm for MST

class KruskalAlgorithm : public MSTAlgorithm {

public:

    // Override the compute method to use Kruskal's algorithm

    vector<pair<int, int>> compute(int n, const vector<vector<Edge>>& adj) override {

        vector<Edge> edgeList;

        // Convert adjacency list to a list of edges

        for (int i = 1; i <= n; ++i) {

            for (const Edge& e : adj[i]) {

                // Ensure each edge is added only once by checking i < e.to

                if (i < e.v) {

                    edgeList.push\_back({i, e.v, e.weight});

                }

            }

        }

        // Compute MST using Kruskal's algorithm

        vector<Edge> mstEdges = kruskal(n, edgeList); // Call Kruskal's algorithm function (assumed to be defined elsewhere)

        // Convert edge list to a list of pairs for the result

        vector<pair<int, int>> mstPairs;

        for (const Edge& e : mstEdges) {

            mstPairs.emplace\_back(e.u, e.v);

        }

        return mstPairs;

    }

};

// Factory class to create MST algorithm objects

class MSTFactory {

public:

    // Static method to create an MST algorithm instance based on the given type

    static unique\_ptr<MSTAlgorithm> createAlgorithm(const string& type) {

        if (type == "Prim") {

            return make\_unique<PrimAlgorithm>(); // Return a unique\_ptr to a PrimAlgorithm instance

        } else if (type == "Kruskal") {

            return make\_unique<KruskalAlgorithm>(); // Return a unique\_ptr to a KruskalAlgorithm instance

        }

        return nullptr; // Return nullptr if the type is invalid

    }

};

Prim.cpp: #ifndef PRIM\_H

#define PRIM\_H

#include "MSTAlgorithms.hpp"

#include <queue>

#include <vector>

#include <limits>

// Prim class inherits from MST and implements Prim's algorithm to find the Minimum Spanning Tree (MST)

class Prim : public MST {

public:

    // Override the calculateMST method to implement Prim's algorithm

    std::vector<std::pair<int, int>> calculateMST(const Graph& graph) const override {

        std::vector<std::pair<int, int>> mstEdges; // To store the edges of the MST

        int numVertices = graph.getNumVertices();   // Get the number of vertices in the graph

        std::vector<bool> inMST(numVertices, false); // Track vertices included in the MST

        std::vector<int> parent(numVertices, -1);     // To store the parent of each vertex in the MST

        std::vector<int> key(numVertices, std::numeric\_limits<int>::max()); // Key values to pick minimum weight edge

        using pii = std::pair<int, int>; // Pair to represent (weight, vertex)

        // Comparator for the priority queue (min-heap)

        auto cmp = [](const pii& a, const pii& b) { return a.first > b.first; };

        std::priority\_queue<pii, std::vector<pii>, decltype(cmp)> minHeap(cmp); // Priority queue for edges

        key[0] = 0; // Start with vertex 0

        minHeap.push({0, 0}); // Push the starting vertex into the min-heap

        // Main loop to process all vertices

        while (!minHeap.empty()) {

            int u = minHeap.top().second; // Get the vertex with the minimum key value

            minHeap.pop(); // Remove the vertex from the priority queue

            inMST[u] = true; // Mark the vertex as included in the MST

            // Iterate through all adjacent vertices of u

            for (const auto& edge : graph.getAdjLists()[u]) {

                int v = edge.first; // Adjacent vertex

                int weight = edge.second; // Weight of the edge

                // If the adjacent vertex v is not in the MST and the weight of the edge is smaller than the current key

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

                    key[v] = weight; // Update the key value for vertex v

                    minHeap.push({key[v], v}); // Add the vertex to the priority queue with the updated key

                    parent[v] = u; // Update the parent of vertex v

                }

            }

        }

        // Build the MST edge list from parent information

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

            if (parent[i] != -1) {

                mstEdges.emplace\_back(parent[i], i); // Add edge to the MST

            }

        }

        return mstEdges; // Return the list of edges in the MST

    }

};

#endif // PRIM\_H

הצגת הטיוטות