**#include "application.h"**

**#include <iostream>**

**#include <limits>**

**#include <map>**

**#include <queue> // priority\_queue**

**#include <set>**

**#include <stack>**

**#include <string>**

**#include <unordered\_map>**

**#include <unordered\_set>**

**#include <vector>**

**#include "dist.h"**

**#include "graph.h"**

**#include "json.hpp"**

**using namespace std;**

**using json = nlohmann::json;**

**double INF = numeric\_limits<double>::max();**

**/// @brief Builds the graph from JSON input.**

**/// @param input The input stream containing the JSON data.**

**/// @param g The graph to populate.**

**/// @param buildings The vector of building information.**

**void buildGraph(istream& input, graph<long long, double>& g, vector<BuildingInfo>& buildings) {**

**json data;**

**input >> data;**

**// Add waypoints to the graph**

**for (const auto& waypoint : data["waypoints"]) {**

**long long id = waypoint["id"];**

**double lat = waypoint["lat"];**

**double lon = waypoint["lon"];**

**g.addVertex(id);**

**}**

**// Add edges for footways**

**for (const auto& footway : data["footways"]) {**

**const auto& nodes = footway["nodes"];**

**for (size\_t i = 0; i + 1 < nodes.size(); ++i) {**

**long long from = nodes[i];**

**long long to = nodes[i + 1];**

**Coordinates fromCoords(data["waypoints"][from]["lat"], data["waypoints"][from]["lon"]);**

**Coordinates toCoords(data["waypoints"][to]["lat"], data["waypoints"][to]["lon"]);**

**double distance = distBetween2Points(fromCoords, toCoords);**

**g.addEdge(from, to, distance);**

**g.addEdge(to, from, distance); // Bidirectional edge**

**}**

**}**

**// Add buildings**

**for (const auto& building : data["buildings"]) {**

**long long id = building["id"];**

**double lat = building["lat"];**

**double lon = building["lon"];**

**string name = building["name"];**

**string abbr = building["abbr"];**

**g.addVertex(id);**

**buildings.push\_back(BuildingInfo{id, Coordinates(lat, lon), name, abbr});**

**// Connect building to nearby waypoints**

**for (const auto& waypoint : data["waypoints"]) {**

**long long waypointId = waypoint["id"];**

**double waypointLat = waypoint["lat"];**

**double waypointLon = waypoint["lon"];**

**Coordinates buildingCoords(lat, lon);**

**Coordinates waypointCoords(waypointLat, waypointLon);**

**double distance = distBetween2Points(buildingCoords, waypointCoords);**

**if (distance <= 0.036) { // Threshold distance**

**g.addEdge(id, waypointId, distance);**

**g.addEdge(waypointId, id, distance);**

**}**

**}**

**}**

**}**

**/// Function to get building info by abbreviation or name**

**BuildingInfo getBuildingInfo(const vector<BuildingInfo>& buildings, const string& query) {**

**for (const BuildingInfo& building : buildings) {**

**if (building.abbr == query) {**

**return building;**

**} else if (building.name.find(query) != string::npos) {**

**return building;**

**}**

**}**

**BuildingInfo fail;**

**fail.id = -1;**

**return fail;**

**}**

**/// Function to get the closest building to a given coordinates**

**BuildingInfo getClosestBuilding(const vector<BuildingInfo>& buildings, Coordinates c) {**

**if (buildings.empty()) {**

**throw runtime\_error("No buildings available.");**

**}**

**double minDestDist = INF;**

**BuildingInfo ret = buildings.at(0);**

**for (const BuildingInfo& building : buildings) {**

**double dist = distBetween2Points(building.location, c);**

**if (dist < minDestDist) {**

**minDestDist = dist;**

**ret = building;**

**}**

**}**

**return ret;**

**}**

**/// Dijkstra's algorithm for finding the shortest path**

**vector<long long> dijkstra(const graph<long long, double>& G, long long start, long long target, const set<long long>& ignoreNodes) {**

**map<long long, double> dist;**

**map<long long, long long> prev;**

**priority\_queue<pair<double, long long>, vector<pair<double, long long>>, greater<pair<double, long long>>> pq;**

**// Initialize distances and predecessors**

**for (const auto& node : G.getVertices()) {**

**dist[node] = INF;**

**prev[node] = -1;**

**}**

**dist[start] = 0;**

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

**while (!pq.empty()) {**

**double currentDist = pq.top().first;**

**long long node = pq.top().second;**

**pq.pop();**

**if (node == target) break; // Stop if we reach the target**

**// Skip outdated distances**

**if (currentDist > dist[node]) continue;**

**// Process neighbors**

**for (const auto& neighbor : G.getNeighbors(node)) {**

**if (ignoreNodes.find(neighbor.first) != ignoreNodes.end() && neighbor.first != target) continue;**

**double weight;**

**if (G.getWeight(node, neighbor.first, weight)) {**

**double alt = dist[node] + weight;**

**if (alt < dist[neighbor.first]) {**

**dist[neighbor.first] = alt;**

**prev[neighbor.first] = node;**

**pq.push({alt, neighbor.first});**

**}**

**}**

**}**

**}**

**// Reconstruct the path**

**vector<long long> path;**

**for (long long at = target; at != -1; at = prev[at]) {**

**path.push\_back(at);**

**}**

**reverse(path.begin(), path.end());**

**// Return an empty path if no valid path exists**

**if (path.size() == 1 && path[0] != start) {**

**return {};**

**}**

**return path;**

**}**

**/// Function to calculate the total path length from a series of nodes**

**double pathLength(const graph<long long, double>& G, const vector<long long>& path) {**

**double length = 0.0;**

**double weight;**

**for (size\_t i = 0; i + 1 < path.size(); i++) {**

**bool res = G.getWeight(path.at(i), path.at(i + 1), weight);**

**if (!res) {**

**return -1; // Return -1 if no edge exists between nodes**

**}**

**length += weight;**

**}**

**return length;**

**}**

**/// Function to output the path from start to target**

**void outputPath(const vector<long long>& path) {**

**for (size\_t i = 0; i < path.size(); i++) {**

**cout << path.at(i);**

**if (i != path.size() - 1) {**

**cout << "->";**

**}**

**}**

**cout << endl;**

**}**

**/// Main application function to handle user interaction and pathfinding**

**void application(const vector<BuildingInfo>& buildings, const graph<long long, double>& G) {**

**string person1Building, person2Building;**

**set<long long> buildingNodes;**

**// Build the set of building nodes**

**for (const auto& building : buildings) {**

**buildingNodes.insert(building.id);**

**}**

**cout << endl;**

**cout << "Enter person 1's building (partial name or abbreviation), or #> ";**

**getline(cin, person1Building);**

**while (person1Building != "#") {**

**cout << "Enter person 2's building (partial name or abbreviation)> ";**

**getline(cin, person2Building);**

**// Look up buildings by query**

**BuildingInfo p1 = getBuildingInfo(buildings, person1Building);**

**BuildingInfo p2 = getBuildingInfo(buildings, person2Building);**

**if (p1.id == -1) {**

**cout << "Person 1's building not found" << endl;**

**} else if (p2.id == -1) {**

**cout << "Person 2's building not found" << endl;**

**} else {**

**cout << endl;**

**cout << "Person 1's point:" << endl;**

**cout << " " << p1.name << endl;**

**cout << " " << p1.id << endl;**

**cout << " (" << p1.location.lat << ", " << p1.location.lon << ")" << endl;**

**cout << "Person 2's point:" << endl;**

**cout << " " << p2.name << endl;**

**cout << " " << p2.id << endl;**

**cout << " (" << p2.location.lat << ", " << p2.location.lon << ")" << endl;**

**Coordinates centerCoords = centerBetween2Points(p1.location, p2.location);**

**BuildingInfo dest = getClosestBuilding(buildings, centerCoords);**

**cout << "Destination Building:" << endl;**

**cout << " " << dest.name << endl;**

**cout << " " << dest.id << endl;**

**cout << " (" << dest.location.lat << ", " << dest.location.lon << ")"**

**<< endl;**

**// Get the shortest paths for both people**

**vector<long long> P1Path = dijkstra(G, p1.id, dest.id, buildingNodes);**

**vector<long long> P2Path = dijkstra(G, p2.id, dest.id, buildingNodes);**

**// Handle edge cases where paths are empty**

**if (P1Path.empty() || P2Path.empty()) {**

**cout << endl;**

**cout << "At least one person was unable to reach the destination building. Is an edge missing?" << endl;**

**cout << endl;**

**} else {**

**cout << endl;**

**cout << "Person 1's distance to dest: " << pathLength(G, P1Path) << " miles" << endl;**

**cout << "Path: ";**

**outputPath(P1Path);**

**cout << endl;**

**cout << "Person 2's distance to dest: " << pathLength(G, P2Path) << " miles" << endl;**

**cout << "Path: ";**

**outputPath(P2Path);**

**}**

**}**

**// Prompt for another navigation**

**cout << endl;**

**cout << "Enter person 1's building (partial name or abbreviation), or #> ";**

**getline(cin, person1Building);**

**}**

**}**

**#pragma once**

**#include <iostream>**

**#include <unordered\_map>**

**#include <vector>**

**#include <set>**

**#include <stdexcept>**

**#include <algorithm> // For std::remove\_if**

**/// @brief Simple directed graph using an adjacency list.**

**/// @tparam VertexT vertex type**

**/// @tparam WeightT edge weight type**

**template <typename VertexT, typename WeightT>**

**class graph {**

**private:**

**std::unordered\_map<VertexT, std::vector<std::pair<VertexT, WeightT>>> adjacency\_list;**

**size\_t edge\_count = 0; // Track the number of edges**

**public:**

**/// Default constructor**

**graph() = default;**

**/// @brief Add a vertex to the graph. O(1).**

**bool addVertex(const VertexT& v) {**

**if (adjacency\_list.find(v) == adjacency\_list.end()) {**

**adjacency\_list[v] = {};**

**return true; // Vertex added**

**}**

**return false; // Vertex already exists**

**}**

**/// @brief Add a directed edge from `from` to `to` with weight `weight`.**

**/// If the edge already exists, overwrite the weight.**

**bool addEdge(const VertexT& from, const VertexT& to, const WeightT& weight) {**

**// Ensure both vertices exist**

**if (!containsVertex(from) || !containsVertex(to)) {**

**return false; // Edge cannot be added if either vertex is missing**

**}**

**// Check if the edge already exists and update its weight**

**for (auto& edge : adjacency\_list[from]) {**

**if (edge.first == to) {**

**edge.second = weight; // Update weight if edge exists**

**return true; // Edge updated**

**}**

**}**

**// Add a new edge**

**adjacency\_list[from].emplace\_back(to, weight);**

**edge\_count++;**

**return true;**

**}**

**/// @brief Remove a directed edge from `from` to `to`.**

**void removeEdge(const VertexT& from, const VertexT& to) {**

**if (adjacency\_list.find(from) != adjacency\_list.end()) {**

**auto& edges = adjacency\_list[from];**

**auto initial\_size = edges.size();**

**edges.erase(std::remove\_if(edges.begin(), edges.end(),**

**[&to](const std::pair<VertexT, WeightT>& edge) {**

**return edge.first == to;**

**}),**

**edges.end());**

**if (edges.size() < initial\_size) {**

**edge\_count--; // Only decrement if an edge was removed**

**}**

**}**

**}**

**/// @brief Get all neighbors of a vertex.**

**std::vector<std::pair<VertexT, WeightT>> getNeighbors(const VertexT& v) const {**

**auto it = adjacency\_list.find(v);**

**if (it == adjacency\_list.end()) {**

**return {}; // Return empty if vertex does not exist**

**}**

**return it->second;**

**}**

**/// @brief Get all neighbors of a vertex as a set of vertices.**

**std::set<VertexT> neighbors(const VertexT& v) const {**

**std::set<VertexT> neighborSet;**

**auto it = adjacency\_list.find(v);**

**if (it == adjacency\_list.end()) {**

**return neighborSet; // Return empty set if vertex does not exist**

**}**

**for (const auto& neighbor : it->second) {**

**neighborSet.insert(neighbor.first); // Only insert the vertex IDs**

**}**

**return neighborSet;**

**}**

**/// @brief Check if the graph contains a vertex.**

**bool containsVertex(const VertexT& v) const {**

**return adjacency\_list.find(v) != adjacency\_list.end();**

**}**

**/// @brief Get the number of vertices in the graph.**

**size\_t numVertices() const {**

**return adjacency\_list.size();**

**}**

**/// @brief Get the number of edges in the graph.**

**size\_t numEdges() const {**

**return edge\_count;**

**}**

**/// @brief Get all vertices in the graph.**

**std::vector<VertexT> getVertices() const {**

**std::vector<VertexT> vertices;**

**for (const auto& [vertex, edges] : adjacency\_list) {**

**vertices.push\_back(vertex);**

**}**

**return vertices;**

**}**

**/// @brief Get the weight of an edge between `from` and `to`.**

**/// @param from Starting vertex**

**/// @param to Ending vertex**

**/// @param weight The weight of the edge, if it exists**

**/// @return `true` if the edge exists, otherwise `false`**

**bool getWeight(const VertexT& from, const VertexT& to, WeightT& weight) const {**

**auto it = adjacency\_list.find(from);**

**if (it != adjacency\_list.end()) {**

**for (const auto& edge : it->second) {**

**if (edge.first == to) {**

**weight = edge.second;**

**return true; // Edge found**

**}**

**}**

**}**

**return false; // No edge found**

**}**

**/// @brief Print the graph (for debugging).**

**void print(std::ostream& out = std::cout) const {**

**for (const auto& [vertex, edges] : adjacency\_list) {**

**out << vertex << ": ";**

**for (const auto& edge : edges) {**

**out << "{" << edge.first << ", " << edge.second << "} ";**

**}**

**out << "\n";**

**}**

**}**

**};**