#include <iostream>

#include <vector>

#include <queue>

#include <map>

#include <climits>

#include <algorithm>

using namespace std;

/\*\*

\* @file PA1\_Naveen\_Karasu.cpp

\* @brief Implementation of a graph search algorithm using lowest-cost-first search.

\*

\* This program demonstrates a graph representation and performs lowest-cost-first search which is similar to Dijkstra's algorithm.

\* to find the shortest path from a start node to a goal node.

\*

\* Author: Naveen Karasu

\* Date: 09/22/2024

\*

\* Libraries used:

\* - iostream: Used for input and output operations (e.g., displaying graph and results).

\* - vector: For storing lists of neighbors of nodes.

\* - queue: Specifically, the priority\_queue is used to maintain nodes in the order of lowest cost.

\* - map: Used to store the graph as a mapping of nodes to their neighbors.

\* - climits: Provides INT\_MAX, which is used to initialize the cost to reach each node as "infinity."

\* - algorithm: Includes utilities like reverse(), which is used to reverse the final path after reconstruction.

\*

\*/

/\*\*

\* @brief Lowest-cost-first Algorithm Definition:

\*

\* The lowest-cost-first algorithm is a graph traversal technique, where the next node to be processed is always the one with the least cost

\* from the starting node. It operates by using a priority queue (min-heap) to maintain nodes based on the cumulative cost to reach them.

\*

\* 1. Start from the source node and set its cost to 0, while all other nodes are initialized with an infinite cost.

\* 2. Push the starting node into a priority queue.

\* 3. The priority queue extracts the node with the least cost and updates the cost of its neighbors.

\* 4. If a better (lower) cost to a neighboring node is found, update it and push the neighbor into the queue.

\* 5. This process continues until the goal node is reached or all nodes have been processed.

\*

\* The algorithm ensures that the first time a node is processed from the queue, it has the minimum possible cost.

\*/

// Graph representation: Each node maps to a vector of (neighbor, cost) pairs

map<char, vector<pair<char, int>>> graph;

/\*\*

\* @brief Displays the graph in an adjacency list format.

\*

\* This function iterates through each node in the graph and prints

\* all of its neighbors along with the respective edge costs.

\*

\* @param graph The graph represented as a map where the key is a node and the value is a vector of pairs (neighbor, cost).

\*/

void show\_graph(const map<char, vector< pair<char, int>>>& graph) {

for (const auto& node : graph) {

cout << node.first << " -> ";

for (const auto& edge : node.second) {

cout << "(" << edge.first << ", " << edge.second << ") ";

}

cout << endl;

}

}

/\*\*

\* @brief Adds a directed edge to the graph from one node to another with a specified cost.

\*

\* Ensures that both 'from' and 'to' nodes are initialized in the graph if they don't already exist.

\* The edge is then added from the 'from' node to the 'to' node with the associated cost.

\*

\* @param from The starting node of the edge.

\* @param to The destination node of the edge.

\* @param cost The cost of traveling from 'from' to 'to'.

\*/

void add\_edge(char from, char to, int cost) {

// Ensure both 'from' and 'to' nodes exist in the graph

if (graph.find(from) == graph.end()) {

graph[from] = {}; // Initialize the node if it doesn't exist

}

if (graph.find(to) == graph.end()) {

graph[to] = {}; // Initialize the 'to' node even if it has no outgoing edges yet

}

// Add the edge from 'from' to 'to'

graph[from].push\_back({to, cost});

}

/\*\*

\* @struct Compare

\* @brief Comparator to prioritize the nodes with the lowest cost in the priority queue.

\*

\* This struct defines a custom comparison operator that enables a priority queue

\* to sort nodes based on their path cost in ascending order, enabling the priority queue

\* to act as a min-heap.

\*/

struct Compare {

bool operator()(const pair<int, char>& a, const pair<int, char>& b) {

return a.first > b.first; // Min-heap based on the cost

}

};

/\*\*

\* @brief Prints the current contents of the priority queue.

\*

\* This function is useful for debugging, as it shows the node and its associated cost

\* in the priority queue at each step of the algorithm.

\*

\* Example:

\* Suppose the priority queue contains the following nodes:

\* (B, Cost: 5), (C, Cost: 3), (D, Cost: 7)

\* The function will print:

\* (C, Cost: 3) (B, Cost: 5) (D, Cost: 7)

\* The node with the lowest cost (C) is at the top of the queue.

\*

\* @param pq The priority queue containing pairs of (cost, node).

\*/

void print\_priority\_queue(priority\_queue<pair<int, char>, vector<pair<int, char>>, Compare> pq) {

cout << endl<<"Priority Queue Contents: "<<endl;

while (!pq.empty()) {

cout << "(" << pq.top().second << ", Cost: " << pq.top().first << ") ";

pq.pop(); // Pop the top element to move through the queue

}

cout << endl<<endl;

}

/\*\*

\* @brief Performs lowest-cost-first search to find the shortest path from a start node to a goal node.

\*

\* This function uses a priority queue (min-heap) to explore nodes in the order of lowest cumulative cost.

\* It updates the cost of reaching each node as it explores the graph and ultimately finds the shortest path

\* from the start node to the goal node.

\*

\* Example (Graph with 5 nodes):

\* Suppose we have a graph with nodes A, B, C, D, E where:

\* - A is connected to B (cost 2) and C (cost 3).

\* - B is connected to D (cost 5).

\* - C is connected to D (cost 1) and E (cost 6).

\* - D is connected to E (cost 2).

\*

\* 1. Starting at A, the queue contains (A, 0).

\* 2. A is expanded, adding B and C to the queue: (B, 2), (C, 3).

\* 3. B is expanded, adding D: (C, 3), (D, 7).

\* 4. C is expanded, updating D's cost to 4 and adding E: (D, 4), (E, 9).

\* 5. D is expanded, updating E's cost to 6.

\* The final shortest path from A to E would be A -> C -> D -> E with a cost of 6.

\*

\* @param start The starting node for the search.

\* @param goal The goal node to reach.

\*/

void lowest\_cost\_first\_search(char start, char goal) {

priority\_queue<pair<int, char>, vector<pair<int, char>>, Compare> pq;

map<char, int> costs; // Stores the minimum cost to reach each node

map<char, char> parent; // Stores the parent of each node for path reconstruction

vector<bool> visited(256, false); // Keep track of visited nodes (assuming ASCII characters as node names).

// Initialize costs to infinity for all nodes

for (const auto& node : graph) {

costs[node.first] = INT\_MAX;

}

// Starting point initialization

pq.push({0, start});

costs[start] = 0;

parent[start] = start;

while (!pq.empty()) {

// Debugging: Print the current state of the priority queue

print\_priority\_queue(pq);

int current\_cost = pq.top().first;

char current = pq.top().second;

pq.pop();

// If the node has already been visited, skip it

if (visited[current]) {

continue;

}

visited[current] = true;

cout << "Processing node: " << current << " with current cost: " << current\_cost << endl;

// Check all neighbors of the current node

for (const auto& neighbor\_info : graph[current]) {

char neighbor = neighbor\_info.first;

int edge\_cost = neighbor\_info.second;

// Calculate new cost to reach this neighbor

int new\_cost = current\_cost + edge\_cost;

cout << "New cost to reach " << neighbor << " is " << new\_cost << " (current known cost: " << costs[neighbor] << ")" << endl;

// If a shorter path is found, update the cost and re-add the neighbor to the priority queue

if (new\_cost < costs[neighbor]) {

costs[neighbor] = new\_cost;

parent[neighbor] = current;

pq.push({new\_cost, neighbor});

}

}

}

// Check if the goal is reachable

if (costs[goal] == INT\_MAX) {

cout << "There is no path from " << start << " to " << goal << endl;

} else {

// Display the minimum cost and reconstruct the path

cout << "Minimum cost from " << start << " to " << goal << " is " << costs[goal] << endl;

vector<char> path;

for (char at = goal; at != start; at = parent[at]) {

path.push\_back(at);

}

path.push\_back(start);

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

// Print the reconstructed path

cout << "Path: ";

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

cout << path[i];

if (i < path.size() - 1) cout << " -> ";

}

cout << endl;

}

}

int main() {

// Constructing the graph with edges

add\_edge('A', 'B', 2);

add\_edge('A', 'C', 3);

add\_edge('A', 'D', 4);

add\_edge('B', 'E', 2);

add\_edge('B', 'F', 3);

add\_edge('C', 'J', 7);

add\_edge('D', 'H', 4);

add\_edge('F', 'D', 2);

add\_edge('H', 'G', 3);

add\_edge('J', 'G', 4);

// Display the graph

show\_graph(graph);

// Perform lowest-cost-first search from 'A' to 'G'

cout << endl<<"Performing lowest-cost-first search from A to G:" << endl<<endl;

lowest\_cost\_first\_search('A', 'G');

return 0;

}

/\*\*

\* @brief Priority Queue Definition:

\*

\* A priority queue is a data structure where each element is associated with a priority, and the element with the highest priority

\* is served first. In this case, we use a min-heap where the element with the lowest cost has the highest priority.

\*

\* Basic Operations:

\* - push(element): Adds an element to the queue.

\* - pop(): Removes the element with the highest priority (in this case, the lowest cost).

\* - top(): Returns the element with the highest priority without removing it.

\*

\* Example:

\* If we push (B, 5), (C, 3), and (D, 7) into a min-heap priority queue, the queue will store them as:

\* (C, 3), (B, 5), (D, 7). Calling top() would return (C, 3) because it has the lowest cost.

\*/