Aim-topological sort

#include <iostream>

#include <unordered\_map>

#include <vector>

#include <queue>

#include <stack>

#include <set>

#include <algorithm>

using namespace std;

class Graph {

public:

unordered\_map<int, vector<int>> adjList;

bool isDirected;

Graph(bool directed = true) : isDirected(directed) {}

void addVertex(int vertex) {

adjList[vertex]; // Create an empty list for the vertex if it doesn't exist

}

void addEdge(int vertex1, int vertex2) {

adjList[vertex1].push\_back(vertex2);

if (!isDirected) {

adjList[vertex2].push\_back(vertex1); // If the graph is undirected, add reverse edge

}

}

void display() {

for (const auto& pair : adjList) {

cout << pair.first << ": ";

for (int neighbor : pair.second) {

cout << neighbor << " ";

}

cout << endl;

}

}

void bfs(int start) {

set<int> visited;

queue<int> q;

q.push(start);

visited.insert(start);

while (!q.empty()) {

int vertex = q.front();

q.pop();

cout << vertex << " ";

for (int neighbor : adjList[vertex]) {

if (visited.find(neighbor) == visited.end()) {

visited.insert(neighbor);

q.push(neighbor);

}

}

}

}

void dfs(int start) {

set<int> visited;

stack<int> s;

s.push(start);

while (!s.empty()) {

int vertex = s.top();

s.pop();

if (visited.find(vertex) == visited.end()) {

visited.insert(vertex);

cout << vertex << " ";

// Add neighbors in reverse order to process them in correct order

for (auto it = adjList[vertex].rbegin(); it != adjList[vertex].rend(); ++it) {

if (visited.find(\*it) == visited.end()) {

s.push(\*it);

}

}

}

}

}

// Topological sort using DFS (only for Directed Acyclic Graphs)

void topologicalSortUtil(int v, set<int>& visited, stack<int>& Stack) {

visited.insert(v);

// Recur for all the vertices adjacent to this vertex

for (int neighbor : adjList[v]) {

if (visited.find(neighbor) == visited.end()) {

topologicalSortUtil(neighbor, visited, Stack);

}

}

// Push current vertex to stack which stores result

Stack.push(v);

}

void topologicalSort() {

stack<int> Stack;

set<int> visited;

// Call the recursive helper function for all vertices

for (const auto& pair : adjList) {

if (visited.find(pair.first) == visited.end()) {

topologicalSortUtil(pair.first, visited, Stack);

}

}

// Print the contents of the stack

while (!Stack.empty()) {

cout << Stack.top() << " ";

Stack.pop();

}

cout << endl;

}

};

int main() {

Graph g(true); // Directed graph (set to false for undirected graph)

g.addVertex(1);

g.addVertex(2);

g.addVertex(3);

g.addVertex(4);

// Add edges

g.addEdge(1, 2);

g.addEdge(1, 3);

g.addEdge(3, 4);

// Display the graph

cout << "Graph: " << endl;

g.display();

// Perform BFS

cout << "BFS Traversal starting from vertex 1: ";

g.bfs(1);

cout << endl;

// Perform DFS

cout << "DFS Traversal starting from vertex 1: ";

g.dfs(1);

cout << endl;

// Perform Topological Sort (only valid for Directed Acyclic Graphs)

cout << "Topological Sort: ";

g.topologicalSort();

return 0;

}