Bài tập do samsung tài trợ:

Phần B:

8. Sắp xếp tô pô và BFS

Lý do chính là do BFS thực hiện tìm kiếm theo tầng, tức là đỉnh cùng tầng sẽ được xét trước các đỉnh ở tầng sau. Trong quá trình tìm kiếm theo tầng, các đỉnh có thể được duyệt qua theo bất kỳ thứ tự nào trong một tầng cụ thể. Do đó, không có đảm bảo rằng các đỉnh cùng tầng sẽ được xét theo thứ tự tăng dần.

Điều này có nghĩa là, trong quá trình thực hiện BFS, các đỉnh cùng tầng có thể được xét qua theo một thứ tự ngẫu nhiên. Điều này dẫn đến việc không có đảm bảo rằng đỉnh nào sẽ được tô pô trước, và kết quả không phải là một thứ tự tô pô đúng.

9. Chu trình Euler có hướng

#include <iostream>

#include <vector>

#include <stack>

using namespace std;

class Graph {

private:

int V;

vector<vector<int>> adj;

vector<int> inDegree;

vector<int> outDegree;

public:

Graph(int vertices) {

V = vertices;

adj.resize(V);

inDegree.assign(V, 0);

outDegree.assign(V, 0);

}

void addEdge(int u, int v) {

adj[u].push\_back(v);

outDegree[u]++;

inDegree[v]++;

}

bool isEulerianCycle() {

// Check if the graph is connected

if (!isConnected()) {

cout << "No Eulerian cycle exists." << endl;

return false;

}

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

if (inDegree[i] != outDegree[i]) {

cout << "No Eulerian cycle exists." << endl;

return false;

}

}

return true;

}

bool isConnected() {

vector<bool> visited(V, false);

DFS(0, visited);

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

if (!visited[i]) {

return false;

}

}

return true;

}

void DFS(int v, vector<bool>& visited) {

visited[v] = true;

for (int u : adj[v]) {

if (!visited[u]) {

DFS(u, visited);

}

}

}

void findEulerianCycle(int startVertex) {

if (!isEulerianCycle()) {

return;

}

stack<int> currentPath;

vector<int> circuit;

currentPath.push(startVertex);

int currentVertex = startVertex;

while (!currentPath.empty()) {

if (outDegree[currentVertex] > 0) {

currentPath.push(currentVertex);

int nextVertex = adj[currentVertex].back();

outDegree[currentVertex]--;

adj[currentVertex].pop\_back();

currentVertex = nextVertex;

} else {

circuit.push\_back(currentVertex);

currentVertex = currentPath.top();

currentPath.pop();

}

}

for (auto it = circuit.rbegin(); it != circuit.rend(); ++it) {

cout << \*it << " ";

}

cout << endl;

}

};

int main() {

// Example

Graph g(4);

g.addEdge(0, 1);

g.addEdge(1, 2);

g.addEdge(2, 3);

g.addEdge(3, 0);

cout << "Eulerian cycle: ";

g.findEulerianCycle(0);

return 0;

}

10. Liên thông mạnh

Khởi tạo một vector **visited** để theo dõi các đỉnh đã được ghé thăm.

Bắt đầu từ đỉnh v, thực hiện DFS trên đồ thị. Khi ghé thăm một đỉnh, đánh dấu nó là đã ghé thăm trong vector **visited**.

Sau khi hoàn thành DFS, tất cả các đỉnh được ghé thăm và có thể đến được từ đỉnh v sẽ thuộc cùng một thành phần liên thông mạnh.

Thực hiện DFS trên đồ thị để tính thời gian hoàn thành (finish time) cho mỗi đỉnh. Lưu thời gian hoàn thành vào một vector **finishTime**.

Tính đồ thị chuyển vị (transpose graph) của đồ thị gốc (đảo chiều của tất cả các cạnh).

Khởi tạo lại vector **visited** để theo dõi các đỉnh đã được ghé thăm.

Duyệt qua các đỉnh theo thứ tự giảm dần của thời gian hoàn thành (lấy từ vector **finishTime**). Nếu đỉnh chưa được ghé thăm, thực hiện DFS từ đỉnh đó.

Mỗi lần thực hiện DFS sẽ tạo ra một thành phần liên thông mạnh của đồ thị gốc.

11. Đường đi Hamilton trong đồ thị có hướng phi chu trình

#include <iostream>

#include <vector>

#include <queue>

#include <stack>

using namespace std;

class Graph {

private:

int V;

vector<vector<int>> adj;

public:

Graph(int vertices) {

V = vertices;

adj.resize(V);

}

void addEdge(int u, int v) {

adj[u].push\_back(v);

}

vector<int> topologicalSort() {

vector<int> inDegree(V, 0);

for (int u = 0; u < V; u++) {

for (int v : adj[u]) {

inDegree[v]++;

}

}

queue<int> q;

for (int u = 0; u < V; u++) {

if (inDegree[u] == 0) {

q.push(u);

}

}

vector<int> topoOrder;

while (!q.empty()) {

int u = q.front();

q.pop();

topoOrder.push\_back(u);

for (int v : adj[u]) {

inDegree[v]--;

if (inDegree[v] == 0) {

q.push(v);

}

}

}

return topoOrder;

}

bool hasUniqueDirectedPath() {

vector<int> topoOrder = topologicalSort();

vector<int> orderIndex(V);

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

orderIndex[topoOrder[i]] = i;

}

for (int u = 0; u < V; u++) {

for (int v : adj[u]) {

if (orderIndex[u] >= orderIndex[v]) {

return false;

}

}

}

return true;

}

};

int main() {

Graph g(6);

g.addEdge(0, 1);

g.addEdge(1, 2);

g.addEdge(2, 3);

g.addEdge(3, 4);

g.addEdge(4, 5);

if (g.hasUniqueDirectedPath()) {

cout << "There exists a unique directed path through each vertex." << endl;

} else {

cout << "There does not exist a unique directed path through each vertex." << endl;

}

return 0;

}

12. Thứ tự tô pô duy nhất

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

class Graph {

private:

int V;

vector<vector<int>> adj;

public:

Graph(int vertices) {

V = vertices;

adj.resize(V);

}

void addEdge(int u, int v) {

adj[u].push\_back(v);

}

bool hasUniqueTopologicalOrder() {

vector<int> inDegree(V, 0);

for (int u = 0; u < V; u++) {

for (int v : adj[u]) {

inDegree[v]++;

}

}

queue<int> q;

int count = 0;

for (int u = 0; u < V; u++) {

if (inDegree[u] == 0) {

q.push(u);

count++;

}

}

while (!q.empty()) {

int u = q.front();

q.pop();

for (int v : adj[u]) {

inDegree[v]--;

if (inDegree[v] == 0) {

q.push(v);

count++;

}

}

}

return (count == V); // If count equals the number of vertices, there is a unique topological order

}

};

int main() {

Graph g(6);

g.addEdge(0, 1);

g.addEdge(1, 2);

g.addEdge(2, 3);

g.addEdge(3, 4);

g.addEdge(4, 5);

if (g.hasUniqueTopologicalOrder()) {

cout << "There exists a unique topological order." << endl;

} else {

cout << "There does not exist a unique topological order." << endl;

}

return 0;

}

13. Đếm đồ thị có hướng

Một đồ thị có hướng *V* đỉnh không chứa cạnh song song khi và chỉ khi mỗi cặp đỉnh (*u*,*v*) đều có thể có cạnh *u*→*v* hoặc *v*→*u* (hoặc cả hai), nhưng không thể có cả hai chiều đồng thời.

Để tạo ra một đồ thị như vậy, ta cần quyết định hướng của cạnh cho mỗi cặp đỉnh (*u*,*v*). Mỗi cặp đỉnh đều có 2 lựa chọn (điều này tương ứng với việc có cạnh *u*→*v*, *v*→*u*, hoặc cả hai). Với *V* đỉnh, ta có *V*×*V*=*V2* cặp đỉnh, và mỗi cặp đỉnh đều có 2 lựa chọn. Do đó, tổng số đồ thị khác nhau có thể được tạo ra là 2^(V^2)

14. Đếm đồ thị có hướng phi chu trình(DAG)

Mỗi đỉnh trong DAG có thể kết nối với bất kỳ đỉnh nào khác mà không tạo thành chu trình. Cụ thể, cho mỗi cặp đỉnh (*u*,*v*) với u != v ta có thể có một trong những trường hợp sau:

1. Có cạnh *u*→*v*: *V* lựa chọn.
2. Không có cạnh giữa *u* và *v*.

Vì mỗi cặp đỉnh đều có 2 lựa chọn, và có *V* cặp đỉnh (đối với *V* đỉnh), số lượng DAG có hướng có thể được xây dựng là 2*V*×(*V*−1)

15. Sắp xếp tô pô dùng hàng đợi

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

using namespace std;

void topologicalSort(vector<vector<int>>& graph, vector<int>& indegree) {

int V = graph.size();

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

for (int j : graph[i]) {

indegree[j]++;

}

}

queue<int> q;

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

if (indegree[i] == 0) {

q.push(i);

}

}

while (!q.empty()) {

int u = q.front();

q.pop();

cout << u << " ";

for (int v : graph[u]) {

indegree[v]--;

if (indegree[v] == 0) {

q.push(v);

}

}

}

}

int main() {

int V, E;

cout << "Nhap so dinh V va so canh E: ";

cin >> V >> E;

vector<vector<int>> graph(V);

vector<int> indegree(V, 0);

cout << "Nhap " << E << " canh cua do thi:\n";

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

int u, v;

cin >> u >> v;

graph[u].push\_back(v);

}

cout << "Thu tu sap xep topo: ";

topologicalSort(graph, indegree);

return 0;

}