Bài tập do Samsung tài trợ

Phần A: Đồ thị vô hướng

1. Chu trình Euler và chu trình Hamilton.

Đồ thị 1 (0-1 0-2 0-3 1-3 1-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8):

Bậc của các đỉnh: 0:2, 1:4, 2:3, 3:3, 4:4, 5:3, 6:3, 7:4, 8:4, 9:2.

Mọi đỉnh đều có bậc chẵn, và đồ thị là liên thông. Vậy đồ thị này có chu trình Euler.

Có chu trình Hamilton: 0-1-4-8-7-6-9-5-2-3.

Đồ thị 2 (0-1 0-2 0-3 1-3 0-3 2-5 5-6 3-6 4-7 4-8 5-8 5-9 6-7 6-9 8-8):

Bậc của các đỉnh: 0:4, 1:3, 2:2, 3:3, 4:4, 5:4, 6:3, 7:2, 8:4, 9:2.

Đồ thị không có chu trình Euler do có hai đỉnh có bậc lẻ (0 và 5).

Không có chu trình Hamilton vì đỉnh 0 và 5 xuất hiện hai lần.

Đồ thị 3 (0-1 1-2 1-3 0-3 0-4 2-5 2-9 3-6 4-7 4-8 5-8 5-9 6-7 6-9 7-8):

Bậc của các đỉnh: 0:4, 1:4, 2:3, 3:3, 4:4, 5:3, 6:3, 7:4, 8:4, 9:2.

Mọi đỉnh đều có bậc chẵn, và đồ thị là liên thông. Vậy đồ thị này có chu trình Euler.

Có chu trình Hamilton: 0-1-2-3-6-9-7-8-5-4.

Đồ thị 4 (4-1 7-9 6-2 7-3 5-0 0-2 0-8 1-6 3-9 6-3 2-8 1-5 9-8 4-5 4-7):

Bậc của các đỉnh: 0:3, 1:4, 2:3, 3:3, 4:5, 5:3, 6:4, 7:4, 8:3, 9:2.

Đồ thị không có chu trình Euler do có nhiều hơn hai đỉnh có bậc lẻ (4 và 6).

Không có chu trình Hamilton vì các đỉnh 0, 1, 2, 3, và 6 xuất hiện hai lần.

2.

Một đồ thị vô hướng có V đỉnh sẽ có tối đa ,​ cạnh, là số cạnh tối đa có thể có trong một đồ thị đầy đủ.

Nếu *E* lớn hơn ,​, thì không có cách nào có thể tạo ra đồ thị vô hướng thỏa mãn yêu cầu song song. Ngược lại, nếu *E* không vượt quá ,​, ta có thể sử dụng công thức tổ hợp để tính số cách chọn *E* cạnh từ ,

Công thức tổ hợp để chọn *E* phần tử từ V⋅(V−1)/2​ là C(,E).

3. Cạnh song song

#include <iostream>

#include <vector>

#include <unordered\_set>

using namespace std;

class Graph {

public:

Graph(int vertices) : V(vertices) {

adj.resize(V);

}

void addEdge(int u, int v) {

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

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

}

int countParallelEdges() {

int count = 0;

unordered\_set<int> visited;

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

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

countParallelEdgesDFS(v, visited);

}

}

return count / 2;

}

private:

void countParallelEdgesDFS(int v, unordered\_set<int>& visited) {

visited.insert(v);

for (int neighbor : adj[v]) {

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

count++;

countParallelEdgesDFS(neighbor, visited);

}

}

}

int V;

vector<vector<int>> adj;

};

int main() {

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 3);

int parallelEdges = g.countParallelEdges();

cout << "Number of parallel edges: " << parallelEdges << endl;

return 0;

}

4. Chu trình lẻ

Giả sử đồ thị không chứa chu trình độ dài lẻ và ta sẽ chứng minh nó là bipartite bằng cách thực hiện BFS (hoặc DFS) để tô màu các đỉnh.

* Bắt đầu từ một đỉnh bất kỳ, ta tô màu đỉnh đó là màu 1.
* Tô màu tất cả các đỉnh kề với đỉnh hiện tại là màu 2.
* Tiếp tục quá trình trên, tô màu xen kẽ giữa màu 1 và màu 2.

Nếu trong quá trình tô màu mà không có cạnh nối giữa hai đỉnh cùng màu, thì đồ thị là bipartite. Ngược lại, nếu có cạnh nối giữa hai đỉnh cùng màu, đồ thị chứa chu trình độ dài lẻ và không phải là bipartite.

5. Biconnected

Giả sử đồ thị không có điểm articulation:

Nếu đồ thị không có điểm articulation, có nghĩa là không có đỉnh nào khi bị xóa sẽ làm đồ thị mất tính liên thông.

Chọn một cặp đỉnh s và t và một đường đi nối giữa chúng:

Chọn một cặp đỉnh s và t bất kỳ và xác định một đường đi giữa chúng.

Sử dụng dữ kiện không có điểm articulation:

Do đồ thị không có điểm articulation, nếu ta xóa bất kỳ đỉnh nào trên đường đi giữa s và t, đồ thị vẫn giữ tính liên thông.

Xây dựng hai đường đi không giao nhau nối s và t:

Bắt đầu từ đỉnh s, đi qua đường đi đã chọn đến t. Nếu có bất kỳ đỉnh nào trên đường đi giữa s và t bị xóa, chúng ta vẫn có thể đi từ s đến t qua các đỉnh còn lại. Điều này chứng minh rằng ta có thể xây dựng hai đường đi không giao nhau nối s và t.

6. Liên thông cạnh

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class Graph {

private:

int V; // Số đỉnh

vector<vector<int>> adj; // Danh sách kề

// Hàm hỗ trợ DFS để kiểm tra tính liên thông của đồ thị

void DFS(int u, vector<bool>& visited);

// Hàm hỗ trợ DFS để kiểm tra tính cầu của cạnh (u, v)

bool isBridge(int u, int v, vector<bool>& visited, vector<int>& disc, vector<int>& low);

public:

Graph(int vertices);

// Thêm cạnh vào đồ thị

void addEdge(int u, int v);

// Kiểm tra tính liên thông cạnh

bool isEdgeConnected();

};

Graph::Graph(int vertices) {

V = vertices;

adj.resize(V);

}

void Graph::addEdge(int u, int v) {

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

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

}

void Graph::DFS(int u, vector<bool>& visited) {

visited[u] = true;

for (int v : adj[u]) {

if (!visited[v]) {

DFS(v, visited);

}

}

}

bool Graph::isBridge(int u, int v, vector<bool>& visited, vector<int>& disc, vector<int>& low) {

static int time = 0;

visited[u] = true;

disc[u] = low[u] = ++time;

for (int adjV : adj[u]) {

if (!visited[adjV]) {

if (isBridge(adjV, u, visited, disc, low)) {

return true;

}

low[u] = min(low[u], low[adjV]);

if (low[adjV] > disc[u]) {

return true; // (u, adjV) là cầu

}

} else if (adjV != v) {

low[u] = min(low[u], disc[adjV]);

}

}

return false;

}

bool Graph::isEdgeConnected() {

vector<bool> visited(V, false);

vector<int> disc(V, 0);

vector<int> low(V, 0);

// Kiểm tra tính liên thông của đồ thị

DFS(0, visited);

// Nếu có đỉnh chưa được thăm, đồ thị không liên thông cạnh

for (bool visit : visited) {

if (!visit) {

return false;

}

}

// Kiểm tra từng cạnh xem có phải là cầu hay không

fill(visited.begin(), visited.end(), false);

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

if (!visited[i] && isBridge(i, -1, visited, disc, low)) {

return false;

}

}

return true;

}

int main() {

// Đồ thị ví dụ

Graph g(5);

g.addEdge(0, 1);

g.addEdge(1, 2);

g.addEdge(2, 3);

g.addEdge(3, 4);

// Kiểm tra tính liên thông cạnh

if (g.isEdgeConnected()) {

cout << "Do thi lien thong canh.\n";

} else {

cout << "Do thi khong lien thong canh.\n";

}

return 0;

}

7. Xử lý ảnh

#include <iostream>

#include <vector>

using namespace std;

// Hàm kiểm tra xem ô (x, y) có nằm trong biên của ảnh không

bool isValid(int x, int y, int rows, int cols) {

return x >= 0 && x < rows && y >= 0 && y < cols;

}

// Hàm DFS thực hiện floodfill

void floodFillDFS(vector<vector<int>>& image, int x, int y, int oldColor, int newColor) {

// Nếu ô hiện tại có màu là oldColor, thay đổi màu thành newColor

if (isValid(x, y, image.size(), image[0].size()) && image[x][y] == oldColor) {

image[x][y] = newColor;

// Thực hiện DFS cho các ô kề

floodFillDFS(image, x + 1, y, oldColor, newColor);

floodFillDFS(image, x - 1, y, oldColor, newColor);

floodFillDFS(image, x, y + 1, oldColor, newColor);

floodFillDFS(image, x, y - 1, oldColor, newColor);

}

}

// Hàm chính để thực hiện floodfill trên ảnh

void floodFill(vector<vector<int>>& image, int x, int y, int newColor) {

// Lấy màu của ô hiện tại

int oldColor = image[x][y];

// Kiểm tra xem màu mới có khác màu cũ hay không

if (oldColor != newColor) {

floodFillDFS(image, x, y, oldColor, newColor);

}

}

// In ảnh ra màn hình

void printImage(const vector<vector<int>>& image) {

for (const auto& row : image) {

for (int pixel : row) {

cout << pixel << " ";

}

cout << endl;

}

}

int main() {

// Một ví dụ về ảnh

vector<vector<int>> image = {

{1, 1, 1, 1, 1},

{1, 1, 1, 0, 0},

{1, 0, 0, 1, 1},

{1, 1, 1, 1, 0}

};

cout << "Original Image:\n";

printImage(image);

// Thực hiện floodfill tại ô (1, 1) với màu mới là 2

floodFill(image, 1, 1, 2);

cout << "\nImage after Floodfill:\n";

printImage(image);

return 0;

}