A-1

+) Chu trình euler: đầu tiên ta đi chứng minh định lý sau: một độ thị có chu trình euler thì bậc của tất cả các đỉnh đều là bậc chẵn!

Thật vật, giả sử có một đỉnh bậc chẵn trong đồ thị có chu trình euler, thì mỗi một cạnh đi đến đỉnh đó phải có một cạnh ra khỏi đỉnh đó để đảm bảo điều kiện mỗi cạnh chỉ được đi đến 1 lần, khi chạy qua đỉnh lẻ cuối cùng của đỉnh đó thì sẽ không đi đâu được nữa. vậy một đồ thị có chu trình euler thì tất cả các đỉnh.

Xét 4 đồ thị đã cho:

-) đồ thị 1:

Đỉnh 0: 3

Đỉnh 1: 3

Đỉnh 2: 3

Đỉnh 3: 3

Đỉnh 4: 3

Đỉnh 5: 3

Đỉnh 6: 3

Đỉnh 7: 3

Đỉnh 8: 2

Đỉnh 9: 3

Có 9 đỉnh có bậc lẻ => không có chu trình euler.

-) đồ thị 2:

Đỉnh 0: 3

Đỉnh 1: 2

Đỉnh 2: 2

Đỉnh 3: 4

Đỉnh 4: 2

Đỉnh 5: 4

Đỉnh 6: 4

Đỉnh 7: 2

Đỉnh 8: 4

Đỉnh 9: 2

Có 1 đỉnh có bậc lẻ => không có chu trình euler

-) 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

Đỉnh 0: 3

Đỉnh 1: 3

Đỉnh 2: 3

Đỉnh 3: 3

Đỉnh 4: 3

Đỉnh 5: 3

Đỉnh 6: 3

Đỉnh 7: 3

Đỉnh 8: 3

Đỉnh 9: 3

Có 9 đỉnh bậc lẻ => không có chu trình eluler

-) 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

Đỉnh 0: 3

Đỉnh 1: 3

Đỉnh 2: 3

Đỉnh 3: 3

Đỉnh 4: 3

Đỉnh 5: 3

Đỉnh 6: 3

Đỉnh 7: 3

Đỉnh 8: 3

Đỉnh 9: 3

Có 9 đỉnh bậc lẻ => không có chu trình euler

+) chu trình haminlton: để kiểm tra có chu trình haminlton hay không ta sử dụng thuật toán sau:

#include <iostream>

#include <vector>

using namespace std;

bool is\_valid(int v, int pos, const vector<vector<int>>& graph, vector<int>& path) {

if (graph[path[pos - 1]][v] == 0)

return false;

return find(path.begin(), path.end(), v) == path.end();

}

bool hamiltonian\_cycle\_util(const vector<vector<int>>& graph, vector<int>& path, int pos) {

if (pos == graph.size()) {

if (graph[path[pos - 1]][path[0]] == 1)

return true;

else

return false;

}

for (int v = 1; v < graph.size(); ++v) {

if (is\_valid(v, pos, graph, path)) {

path[pos] = v;

if (hamiltonian\_cycle\_util(graph, path, pos + 1))

return true;

path[pos] = -1;

}

}

return false;

}

void hamiltonian\_cycle(const vector<vector<int>>& graph) {

vector<int> path(graph.size(), -1);

path[0] = 0;

if (!hamiltonian\_cycle\_util(graph, path, 1)) {

cout << "Khong ton tai chu trinh Hamilton." << endl;

return;

}

cout << "Chu trinh Hamilton: ";

for (int vertex : path)

cout << vertex << " ";

cout << path[0] << endl;

}

int main() {

// Đồ thị được cho

vector<vector<int>> graph = {

….

};

hamiltonian\_cycle(graph);

return 0;

}

* Đồ thị 1 có chu trình hamilton: 0 2 5 9 6 7 8 4 1 3 0;
* Đồ thị 2 có chu trình hamilton: 0 1 3 4 8 7 6 9 5 2 0;
* Đồ thị 3 có chu trình hamilton: 0 1 2 9 5 8 4 7 6 3 0;
* Đồ thị 4 không có chu trình haminlton.

A-2

Do có V đỉnh nên ta có số cạnh không song song nối được giữa tất cả các định của đồ thị là: cạnh. Mà đồ thị đã cho có E cạnh nên ta có số đồ thị thoả mãn yêu cầu đề bài là: ( đồ thị).

A-3

Thuật toán tuyến tính để đếm số cạnh song song trong đồ thị

1. Bắt đầu từ đỉnh đầu tiên.

2. Duyệt qua tất cả các cạnh của đỉnh đó.

3. Đối với mỗi cạnh, kiểm tra xem cạnh đó có giống với bất kỳ cạnh nào khác không. Nếu có, đó là một cạnh song song.

4. Tiếp tục điều này cho tất cả các đỉnh và cạnh trong đồ thị.

#include <iostream>

#include <vector>

#include <unordered\_set>

using namespace std;

int countParallelEdges(const vector<vector<int>>& graph) {

int parallelEdges = 0;

for (int i = 0; i < graph.size(); ++i) {

unordered\_set<int> visitedVertices;

const vector<int>& edges = graph[i];

for (int j = 0; j < edges.size(); ++j) {

int currentEdge = edges[j];

if (visitedVertices.find(currentEdge) != visitedVertices.end()) {

++parallelEdges;

}

else {

visitedVertices.insert(currentEdge);

}

}

}

return parallelEdges;

}

int main() {

vector<vector<int>> graph = {

{1, 2, 3},

{0, 2, 3},

{0, 1, 3},

{0, 1, 2}

};

int result = countParallelEdges(graph);

cout << result << endl;

return 0;

}

A-4

+) Đầu tiên ta sẽ chứng minh một đồ thị 2 màu thì không có chu trình lẻ.

Thật vậy giả sử có đồ thị 2 màu có chu trình lẻ và đồ thị có 2 màu là x và y thì giả sử chu trình là các đỉnh như sau ABCDA. Do đồ thị 2 màu nên 2 đỉnh liên tiếp trong chu trình phải khác màu, từ đó theo chu trình trên thì A ở đầu chu trình nếu xác định có màu x thì cuối chu trình xác định có màu y với bất kỳ chu trình lẻ nào => vô lý => nếu một đồ thị là đồ thị 2 màu thì không có chu trình lẻ.

+) Tiếp đây ta sẽ chứng minh một đồ thị không có chu trình lẻ là đồ thị 2 màu.

* Với đồ thị không có chu trình thì dễ dàng chứng minh được nó là đồ thị 2 màu vì với một điểm bất kỳ trên đồ thị chọn cho nó một màu các đỉnh nối với nó là màu còn lại và cứ nối lần lượt hết các đỉnh theo quy tắc đó ta được đồ thị 2 màu.
* Với đồ thị có chu trình chẵn thì trong một chu trình chẵn, thì cũng chọn trước một điểm bắt đầu có màu x đỉnh tiếp nối với x có màu y, rồi cứ nối tiếp như vậy thì khi quay về đỉnh bắt đầu sẽ lại có màu x bởi vì số đỉnh trong chu kỳ là chẵn.
* Vậy một đồ thị là đồ thị hai màu khi và chỉ khi nó không chứa một chu trình lẻ.

A-5

Giả sử ta có một đồ thị G không có điểm articulation nào. Thì với hai đỉnh s và t bất kỳ thuộc đồ thị G thì do đồ thị liên thông nên tồn tại đường đi từ s tới t. Trên đường đi đó nếu ta xoá một đỉnh bất kỳ thì đồ thị vẫn còn tính liên thông ( do ko có điểm articulation nào). Từ đó suy ra luôn tồn tại một đường đi khác từ s tới t mà không cắt đường đi trên => G là độ thị Biconnected (ĐPCM).

A-6

#include <iostream>

#include <vector>

using namespace std;

class Graph {

public:

Graph(int vertices);

void addEdge(int u, int v);

bool isEdgeConnectedGraph();

private:

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

int vertices;

vector<vector<int>> adjList;

};

Graph::Graph(int vertices) {

this->vertices = vertices;

adjList.resize(vertices);

}

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

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

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

}

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

static int time = 0;

int children = 0;

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

visited[u] = true;

for (int v : adjList[u]) {

if (!visited[v]) {

children++;

DFS(v, u, disc, low, visited);

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

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

cout << "Cau found: " << u << " " << v << endl;

}

}

else if (v != parent) {

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

}

}

}

bool Graph::isEdgeConnectedGraph() {

vector<int> disc(vertices, -1);

vector<int> low(vertices, -1);

vector<bool> visited(vertices, false);

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

if (!visited[i]) {

DFS(i, -1, disc, low, visited);

}

}

return true;

}

int main() {

Graph graph(5);

graph.addEdge(0, 1);

graph.addEdge(1, 2);

graph.addEdge(2, 0);

graph.addEdge(1, 3);

graph.addEdge(3, 4);

if (graph.isEdgeConnectedGraph()) {

cout << "Do thi lien thong canh." << endl;

}

else {

cout << "Do thi khong lien thong canh." << endl;

}

return 0;

}

A-7

#include <iostream>

#include <vector>

using namespace std;

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

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

}

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

if (!isValid(x, y, image.size(), image[0].size()) || image[x][y] != originalColor) {

return;

}

image[x][y] = newColor;

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

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

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

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

}

void floodFill(vector<vector<int>>& image, int startX, int startY, int newColor) {

if (!isValid(startX, startY, image.size(), image[0].size())) {

return;

}

int originalColor = image[startX][startY];

if (newColor != originalColor) {

floodFillDFS(image, startX, startY, newColor, originalColor);

}

}

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

for (const auto& row : image) {

for (int pixel : row) {

cout << pixel << " ";

}

cout << endl;

}

}

int main() {

int rows, cols;

cin >> rows >> cols;

vector<vector<int>> image(rows, vector<int>(cols));

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

for (int j = 0; j < cols; ++j) {

cin >> image[i][j];

}

}

int startX, startY, newColor;

cin >> startX >> startY >> newColor;

floodFill(image, startX, startY, newColor);

return 0;

}