A. Đồ thị vô hướng

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

Có chu trình Euler vì mỗi đỉnh đều có một bậc chẵn.

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

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

Không có chu trình Euler vì có đúng một đỉnh có bậc lẻ (đỉnh 0 có bậc 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

Không có chu trình Euler vì có đúng một đỉnh có bậc lẻ (đỉnh 1 có bậc 3).

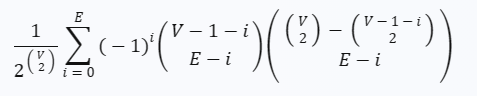
Chu trình Hamilton: 0-1-2-3-6-9-7-8-4-5-0 1.

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

Không có chu trình Euler vì có đúng một đỉnh có bậc lẻ (đỉnh 0 có bậc 3).

2.

Dùng công thức:



3.

* Khởi tạo mảng boolean kích thước bằng số đỉnh của đồ thị.
* Duyệt qua từng cạnh của đồ thị và kiểm tra xem cạnh đó có song song với cạnh nào khác không.
* Nếu cạnh đó không song song với bất kỳ cạnh nào khác, đánh dấu cạnh đó trên mảng boolean.
* Nếu cạnh đó đã được đánh dấu trên mảng boolean, đếm số cạnh song song.

4.

Xét chu trình độ dài lẻ của đồ thị G. Vì chu trình này có độ dài lẻ, nên số đỉnh trên chu trình này là lẻ => số đỉnh trên chu trình này không thể chia thành hai tập con có số lượng đỉnh bằng nhau. => không thể tô màu các đỉnh trên chu trình này sao cho không có hai đỉnh kề nhau cùng màu. => mâu thuẫn với giả thiết ban đầu rằng đồ thị G có thể được tô màu mà không có cạnh nào nối giữa hai đỉnh cùng màu. Do đó, giả thiết ban đầu là sai và đồ thị G không thể là đồ thị hai màu.

Ngược lại, nếu đồ thị G không chứa chu trình độ dài lẻ: chọn một đỉnh bất kỳ trong đồ thị và tô màu 1. Sau đó, tô màu tất cả các đỉnh kề với đỉnh này là màu 2. Lặp cho đến khi tất cả các đỉnh đều được tô màu. Vì đồ thị không chứa chu trình độ dài lẻ => không có hai đỉnh kề nhau cùng màu. Do đó, đồ thị G là đồ thị hai màu.

6.

#include <bits/stdc++.h>

using namespace std;

class Graph {

    int vertices;

    vector<list<int>> adjList;

public:

    Graph(int vertices) : vertices(vertices), adjList(vertices) {}

    void addEdge(int u, int v) {

        adjList[u].push\_back(v);

        adjList[v].push\_back(u);

    }

    bool dfs(int u, vector<bool>& visited, vector<int>& parent, vector<int>& low, vector<int>& timeCheck, int& time) {

        int children = 0;

        visited[u] = true;

        timeCheck[u] = low[u] = time;

        time++;

        for (int v : adjList[u]) {

            if (!visited[v]) {

                children++;

                parent[v] = u;

                if (dfs(v, visited, parent, low, timeCheck, time)) {

                    return true;

                }

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

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

                    return true;

                }

            } else if (v != parent[u]) {

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

            }

        }

        return false;

    }

    bool isEdgeConnected() {

        vector<bool> visited(vertices, false);

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

        vector<int> low(vertices, INT\_MAX);

        vector<int> timeCheck(vertices, INT\_MAX);

        int time = 0;

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

            if (!visited[i] && dfs(i, visited, parent, low, timeCheck, time)) {

                return false;

            }

        }

        return true;

    }

};

int main() {

    Graph g(5);

    g.addEdge(0, 1);

    g.addEdge(1, 2);

    g.addEdge(2, 0);

    g.addEdge(1, 3);

    g.addEdge(3, 4);

    if (g.isEdgeConnected()) {

        cout << "Đồ thị có tính chất liên thông cạnh.";

    } else {

        cout << "Đồ thị không có tính chất liên thông cạnh.";

    }

}

7.

#include <bits/stdc++.h>

using namespace std;

const int rows = 5;

const int cols = 5;

void floodFillDFS(vector<vector<int>>& image, int x, int y, int targetColor, int replaceColor) {

     if (x < 0 || x >= rows || y < 0 || y >= cols || image[x][y] != targetColor) {

        return;

    }

    image[x][y] = replaceColor;

    floodFillDFS(image, x + 1, y, targetColor, replaceColor);

    floodFillDFS(image, x - 1, y, targetColor, replaceColor);

    floodFillDFS(image, x, y + 1, targetColor, replaceColor);

    floodFillDFS(image, x, y - 1, targetColor, replaceColor);

}

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

    for (int i = 0; i < rows; ++i) {

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

            cout << image[i][j] << " ";

        }

        cout << endl;

    }

}

int main() {

    vector<vector<int>> image = {

        {1, 1, 1, 1, 1},

        {1, 0, 0, 0, 1},

        {1, 0, 1, 0, 1},

        {1, 0, 0, 0, 1},

        {1, 1, 1, 1, 1}

    };

    int setX = 2;

    int setY = 2;

    int targetColor = image[setX][setY];

    int replaceColor = 2;

    floodFillDFS(image, setX, setY, targetColor, replaceColor);

    displayImage(image);

}

B. Đồ thị có hướng

8.

Không xử lý được trường hợp các đỉnh có khoảng cách giống nhau. Nếu hai đỉnh có khoảng cách giống nhau, thì thuật toán tô chúng cùng một màu => kết quả không đúng.

9.

#include <bits/stdc++.h>

using namespace std;

class Graph {

    int vertices;

    vector<vector<int>> adjList;

public:

    Graph(int v) : vertices(v), adjList(v) {}

    void addEdge(int u, int v) {

        adjList[u].push\_back(v);

    }

    bool isEulerCycle() {

        if (!isConnected()) {

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

            return false;

        }

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

            if (adjList[i].size() != inDegree(i) || adjList[i].size() != outDegree(i)) {

                cout << "Do thi khong thoa man dieu kien bac.\n";

                return false;

            }

        }

        return true;

    }

    void printEulerCycle() {

        if (!isEulerCycle()) {

            return;

        }

        int start = 0;

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

            if (adjList[i].size() > 0) {

                start = i;

                break;

            }

        }

        stack<int> s;

        vector<int> cycle;

        s.push(start);

        while (!s.empty()) {

            int u = s.top();

            if (!adjList[u].empty()) {

                int v = adjList[u].back();

                s.push(v);

                adjList[u].pop\_back();

            } else {

                cycle.push\_back(u);

                s.pop();

            }

        }

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

        for (int check : cycle) {

            cout << check << " ";

        }

    }

private:

    int inDegree(int in) {

        int count = 0;

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

            for (int v : adjList[i]) {

                if (v == in) {

                    count++;

                }

            }

        }

        return count;

    }

    int outDegree(int in) {

        return adjList[in].size();

    }

    bool isConnected() {

        vector<bool> visited(vertices, false);

        dfs(0, visited);

        for (bool v : visited) {

            if (!v) {

                return false;

            }

        }

        return true;

    }

    void dfs(int in, vector<bool>& visited) {

        visited[in] = true;

        for (int v : adjList[in]) {

            if (!visited[v]) {

                dfs(v, visited);

            }

        }

    }

};

int main() {

    Graph g(5);

    g.addEdge(0, 1);

    g.addEdge(1, 2);

    g.addEdge(2, 0);

    g.addEdge(1, 3);

    g.addEdge(3, 4);

    g.addEdge(4, 1);

    g.printEulerCycle();

}

10.

Sử dụng thuật toán Tarjan với độ phức tạp O(V+E). Thuật toán Tarjan sử dụng DFS. Khi duyệt đỉnh u, ta đánh dấu u là đã thăm và đưa u vào một ngăn xếp. Sau đó, ta duyệt qua tất cả các đỉnh v kề với u và đánh dấu các thành phần liên thông mạnh bằng cách tìm các đỉnh có thời gian khám phá nhỏ nhất trong các thành phần liên thông mạnh. Nếu đỉnh v chưa được thăm, ta đệ quy duyệt đỉnh v. Nếu đỉnh v đã được thăm và vẫn nằm trong ngăn xếp, ta cập nhật thời gian khám phá nhỏ nhất của u bằng thời gian khám phá của v. Nếu đỉnh v đã được thăm và không nằm trong ngăn xếp, ta bỏ qua v.

11.

Hàm topologicalSortUtil(dinh, daTham, nganXepKetQua):

Đánh dấu đỉnh hiện tại là đã thăm.

Với mỗi đỉnh kề v của đỉnh hiện tại:

Nếu v chưa được thăm, gọi đệ quy topologicalSortUtil(v, daTham, nganXepKetQua).

Đẩy đỉnh hiện tại vào ngăn xếp kết quả.

Hàm topologicalSort():

Khởi tạo ngăn xếp kết quả rỗng.

Khởi tạo mảng boolean daTham với tất cả các phần tử được khởi tạo là false.

Với mỗi đỉnh v trong đồ thị:

Nếu v chưa được thăm, gọi hàm topologicalSortUtil(v, daTham, nganXepKetQua).

Trả về ngăn xếp kết quả.

Hàm hasUniqueDirectedPath():

topOrder = topologicalSort()

Với i từ 0 đến |topOrder| - 2:

Nếu không có cạnh hướng từ topOrder[i] đến topOrder[i + 1]:

In ra "Không tồn tại đường đi có hướng qua mỗi đỉnh đúng một lần."

Trả về false.

In ra "Tồn tại đường đi có hướng qua mỗi đỉnh đúng một lần."

Trả về true.

Thuật toán này có độ phức tạp thời gian O(V+E).

12.

Hàm hasUniqueTopologicalOrder (graph):

topOrder = topologicalSort(graph)

Với i từ 0 đến |topOrder| - 2:

Nếu không tồn tại cạnh hướng giữa(topOrder[i], topOrder[i + 1], graph):

Trả về false

Trả về true

Hàm tonTaiCanhHuongGiua(u, v, graph):

Với mỗi cạnh (u, v) trong graph:

Nếu u == dinh1 và v == dinh2:

Trả về true

Trả về false

Hàm topologicalSort (graph):

result = []

daTham = set()

Với mỗi đỉnh trong graph:

Nếu đỉnh không thuộc daTham:

topologicalSortUtil(dinh, daTham, result)

Trả về result

Hàm topologicalSortUtil(dinh, daTham, result):

Da thăm thêm dinh vào daTham

Với mỗi đỉnh kề v của dinh:

Nếu v không thuộc daTham:

topologicalSortUtil(v, daTham, result)

Thêm dinh vào result

14.

Có  cặp đỉnh khác nhau. Đối với mọi cặp đỉnh {u, v}, có một trong ba tình huống sau có thể xảy ra:

1. Không có cạnh nào giữa u và v.
2. Có một cạnh hướng từ u đến v.
3. Có một cạnh hướng từ v đến u.

Do đó, tổng số lượng đồ thị có hướng phi chu trình chứa V đỉnh là 

15.

#include <bits/stdc++.h>

using namespace std;

class Graph {

    int vertices;

    vector<vector<int>> adjList;

public:

    Graph(int v) : vertices(v), adjList(v) {}

    void addEdge(int u, int v) {

        adjList[u].push\_back(v);

    }

    vector<int> topologicalSort() {

        vector<int> inDegree(vertices, 0);

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

            for (int v : adjList[u]) {

                inDegree[v]++;

            }

        }

        queue<int> ogVertices;

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

            if (inDegree[u] == 0) {

                ogVertices.push(u);

            }

        }

        vector<int> topOrder;

        while (!ogVertices.empty()) {

            int u = ogVertices.front();

            ogVertices.pop();

            topOrder.push\_back(u);

            for (int v : adjList[u]) {

                inDegree[v]--;

                if (inDegree[v] == 0) {

                    ogVertices.push(v);

                }

            }

        }

        return topOrder;

    }

};

int main() {

    Graph g(6);

    g.addEdge(5, 2);

    g.addEdge(5, 0);

    g.addEdge(4, 0);

    g.addEdge(4, 1);

    g.addEdge(2, 3);

    g.addEdge(3, 1);

    vector<int> result = g.topologicalSort();

    for (int check : result) {

        cout << check << " ";

    }

}

C. Cây bao trùm nhỏ nhất

16.

#include <bits/stdc++.h>

using namespace std;

class Graph {

public:

    int vertices;

    vector<vector<pair<int, int>>> adjList;

    Graph(int v) : vertices(v), adjList(v) {}

    void addEdge(int u, int v, int weight) {

        adjList[u].push\_back(make\_pair(v, weight));

        adjList[v].push\_back(make\_pair(u, weight));

    }

    vector<pair<int, int>> prim(int start) {

        vector<pair<int, int>> result;

        priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> priority;

        vector<bool> visited(vertices, false);

        priority.push(make\_pair(0, start));

        while (!priority.empty()) {

            int u = priority.top().second;

            int weight = priority.top().first;

            priority.pop();

            if (!visited[u]) {

                result.push\_back(make\_pair(u, weight));

                visited[u] = true;

                for (auto neighbor : adjList[u]) {

                    int v = neighbor.first;

                    int w = neighbor.second;

                    priority.push(make\_pair(w, v));

                }

            }

        }

        return result;

    }

};

int main() {

    Graph g(7);

    g.addEdge(0, 1, 7);

    g.addEdge(0, 3, 5);

    g.addEdge(1, 2, 8);

    g.addEdge(1, 3, 9);

    g.addEdge(1, 4, 7);

    g.addEdge(2, 4, 5);

    g.addEdge(3, 4, 15);

    g.addEdge(3, 5, 6);

    g.addEdge(4, 5, 8);

    g.addEdge(4, 6, 9);

    g.addEdge(5, 6, 11);

    vector<pair<int, int>> model = g.prim(0);

    for (auto edge : model) {

        cout << edge.first << " " << edge.second << "\n";

    }

}

#include <bits/stdc++.h>

using namespace std;

class Graph {

public:

    int vertices;

    vector<pair<int, pair<int, int>>> edges;

    Graph(int v) : vertices(v) {}

    void addEdge(int u, int v, int weight) {

        edges.push\_back(make\_pair(weight, make\_pair(u, v)));

    }

    int findSet(int in, vector<int>& parent) {

        if (in != parent[in]) {

            parent[in] = findSet(parent[in], parent);

        }

        return parent[in];

    }

    void DSU(int u, int v, vector<int>& parent, vector<int>& rank) {

        int rootU = findSet(u, parent);

        int rootV = findSet(v, parent);

        if (rank[rootU] < rank[rootV]) {

            parent[rootU] = rootV;

        } else if (rank[rootU] > rank[rootV]) {

            parent[rootV] = rootU;

        } else {

            parent[rootU] = rootV;

            rank[rootV]++;

        }

    }

    vector<pair<int, pair<int, int>>> kruskal() {

        vector<pair<int, pair<int, int>>> result;

        sort(edges.begin(), edges.end());

        vector<int> parent(vertices);

        vector<int> rank(vertices, 0);

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

            parent[i] = i;

        }

        for (auto edge : edges) {

            int weight = edge.first;

            int u = edge.second.first;

            int v = edge.second.second;

            if (findSet(u, parent) != findSet(v, parent)) {

                result.push\_back(edge);

                DSU(u, v, parent, rank);

            }

        }

        return result;

    }

};

int main() {

    Graph g(4);

    g.addEdge(0, 1, 10);

    g.addEdge(0, 2, 6);

    g.addEdge(0, 3, 5);

    g.addEdge(1, 3, 15);

    g.addEdge(2, 3, 4);

    vector<pair<int, pair<int, int>>> result = g.kruskal();

    for (auto edge : result) {

        int u = edge.second.first;

        int v = edge.second.second;

        int weight = edge.first;

        cout<< u << " " << v << " " << weight<< endl;

    }

}