Github: https://github.com/Clakulis/samsung

1.

Đồ thị thứ nhất:

* Chu trình Euler: Có, vì mọi đỉnh đều có bậc chẵn (0, 2, 2, 4, 4, 4, 4, 4).
* Chu trình Hamilton: Không, không có chu trình Hamilton.

Đồ thị thứ hai:

* Chu trình Euler: Có, vì mọi đỉnh đều có bậc chẵn (4, 4, 4, 4, 4, 4, 4, 4).
* Chu trình Hamilton: Có, vì có thể tìm thấy chu trình Hamilton.

Đồ thị thứ ba:

* Chu trình Euler: Có, vì mọi đỉnh đều có bậc chẵn (4, 4, 4, 4, 4, 4, 4, 4).
* Chu trình Hamilton: Có, vì có thể tìm thấy chu trình Hamilton.

Đồ thị thứ tư:

* Chu trình Euler: Không, vì có 4 đỉnh có bậc lẻ (1, 1, 1, 1).
* Chu trình Hamilton: Không có thông tin nào về việc có chu trình Hamilton hay không.

2.

Số đồ thị vô hướng khác nhau có V đỉnh và E cạnh: (V-1)^(E-1)\*V!/(V-E)!

3.

Function CountParallelEdges(graph\_edges):

parallel\_edges\_count = 0

For i from 0 to len(graph\_edges) - 1:

For j from i + 1 to len(graph\_edges):

edge1 = graph\_edges[i]

edge2 = graph\_edges[j]

If set(edge1).isdisjoint(edge2) and not (edge1[0] == edge2[0] or edge1[1] == edge2[1] or edge1[0] == edge2[1] or edge1[1] == edge2[0]):

parallel\_edges\_count += 1

Return parallel\_edges\_count

4.

Chứng minh rằng một đồ thị là đồ thị hai mầu khi và chỉ khi nó không chứa chu trình độ dài lẻ bằng phương pháp phản chứng:

**(1) Đồ thị hai mầu => Không chứa chu trình độ dài lẻ:**

Giả sử đồ thị hai mầu *G* chứa chu trình độ dài lẻ. Điều này sẽ dẫn đến việc các đỉnh trên chu trình cùng phải thuộc một mầu, điều này xung đột với giả sử ban đầu về *G* là đồ thị hai mầu.

**(2) Không chứa chu trình độ dài lẻ => Đồ thị là đồ thị hai mầu:**

Giả sử *G* không chứa chu trình độ dài lẻ. Chúng ta có thể tô mầu các đỉnh xen kẽ sao cho không có hai đỉnh liền kề nào cùng mầu. Nếu *G* chứa chu trình, thì đỉnh cuối cùng của chu trình sẽ kề với đỉnh đầu tiên, và chúng sẽ thuộc mầu khác nhau. Điều này chỉ có thể xảy ra nếu chu trình có độ dài chẵn.

5.

Giả sử đồ thị *G* không có điểm articulation. Chúng ta muốn chứng minh rằng *G* là đồ thị biconnected.

1. **Mọi cặp đỉnh đều được nối bởi hai đường đi không giao nhau:**
   * Cho một cặp đỉnh *s* và *t* trong *G*.
   * Nếu có đỉnh articulation nào trên đường đi *s* đến *t*, xóa điểm đó vẫn để lại đường đi khác nối *s* và *t*, vì điểm đó không là điểm articulation trên đường đi khác.
   * Do đó, ta có thể xây dựng hai đường đi không giao nhau nối *s* và *t*.
2. **�*G* không có điểm articulation nên *G* là đồ thị biconnected:**
   * Nếu *G* có điểm articulation, việc xóa điểm này có thể làm đồ thị mất tính liên thông.
   * Giả sử *G* không có điểm articulation, điều này có nghĩa là khi xóa bất kỳ đỉnh nào, đồ thị vẫn liên thông.
   * Do đó, mọi cặp đỉnh đều được nối bởi hai đường đi không giao nhau, suy ra *G* là đồ thị biconnected.

6.

Function IsEdgeConnected(graph):

visited = [False] \* len(graph)

for vertex in graph:

if not visited[vertex]:

if not IsBridge(graph, vertex, visited):

return False

return True

Function IsBridge(graph, start\_vertex, visited):

visited[start\_vertex] = True

for neighbor in graph[start\_vertex]:

if not visited[neighbor]:

if not IsBridge(graph, neighbor, visited):

return False

return not any(visited)

7.

Function FloodFill(image, start\_pixel, new\_color):

start\_color = image[start\_pixel[0]][start\_pixel[1]]

If start\_color == new\_color:

Return image

Function DFS(pixel):

x, y = pixel

If 0 <= x < len(image) and 0 <= y < len(image[0]) and image[x][y] == start\_color:

image[x][y] = new\_color

DFS((x + 1, y))

DFS((x - 1, y))

DFS((x, y + 1))

DFS((x, y - 1))

DFS(start\_pixel)

Return image

8.

Thuật toán mô tả chạy BFS và sắp xếp đỉnh dựa trên khoảng cách từ nguồn không đảm bảo kết quả là một thứ tự tô pô chính xác vì:

1. **BFS không đảm bảo thứ tự tô pô:** BFS duyệt qua các đỉnh theo từng "tầng" nhưng không đảm bảo thứ tự chính xác giữa các đỉnh cùng một "tầng".
2. **Mỗi đỉnh có thể có nhiều hơn một đỉnh cha:** Việc này làm cho việc chọn thứ tự giữa các đỉnh cùng một "tầng" trở nên không chắc chắn.
3. **Thứ tự từng "tầng" không đảm bảo thứ tự tô pô toàn cục:** BFS sắp xếp các đỉnh theo từng "tầng" nhưng không đảm bảo thứ tự chính xác giữa các "tầng" khác nhau.

Vì vậy, trong khi BFS có thể được sử dụng để sắp xếp các đỉnh dựa trên khoảng cách từ nguồn, nó không đảm bảo kết quả là một thứ tự tô pô chính xác cho toàn bộ đồ thị. Để đảm bảo thứ tự tô pô, cần sử dụng các thuật toán sắp xếp tô pô như DFS kết hợp với các tiêu chí như thời điểm kết thúc duyệt.

9.

Function Fleury(graph, start\_vertex):

If Not IsEulerianCycle(graph):

Print("Không tồn tại chu trình Euler.")

Return

path = []

current\_vertex = start\_vertex

While graph[current\_vertex]:

next\_vertex = graph[current\_vertex].pop(0)

path.append((current\_vertex, next\_vertex))

current\_vertex = next\_vertex

Print("Chu trình Euler:", path)

Function IsEulerianCycle(graph):

If Not IsStronglyConnected(graph):

Return False

For each vertex in graph:

If len(graph[vertex]) != len(GetInDegree(graph, vertex)):

Return False

Return True

Function IsStronglyConnected(graph):

visited = [False] \* len(graph)

DFS(graph, 0, visited)

If any(not visited[i] for i in range(len(graph))):

Return False

reversed\_graph = GetTranspose(graph)

visited = [False] \* len(graph)

DFS(reversed\_graph, 0, visited)

Return not any(not visited[i] for i in range(len(graph)))

Function DFS(graph, vertex, visited):

visited[vertex] = True

For each neighbor in graph[vertex]:

If not visited[neighbor]:

DFS(graph, neighbor, visited)

Function GetTranspose(graph):

transposed\_graph = {}

For each vertex in graph:

For each neighbor in graph[vertex]:

Add edge to transposed\_graph: (neighbor, vertex)

Return transposed\_graph

Function GetInDegree(graph, vertex):

in\_degree = [0] \* len(graph)

For each vertex in graph:

For each neighbor in graph[vertex]:

in\_degree[neighbor] += 1

Return in\_degree

10.

Function DFS(v, visited, orderStack):

Mark vertex v as visited

For each adjacent vertex k of v:

If vertex k is not visited, recursively call DFS(k, visited, orderStack)

Push v onto the orderStack

Function findSCCs(startVertex):

Initialize an array visited with false values for all vertices

Initialize an empty orderStack

Call DFS(startVertex, visited, orderStack)

Reverse the edges of the graph

Initialize a new array visited with false values for all vertices

Initialize an array sccs to store SCCs

For each vertex v in orderStack:

If vertex v is not visited, call DFS(v, visited, scc) to build a Strongly Connected Component

Return the array of SCCs (sccs)

11.

Function hasHamiltonianPath(graph):

Sort vertices topologically using DFS

For each consecutive pair of vertices:

If no directed edge between current and next, return False

Return True

Topological sort:

Stack orderStack, Array visited

For each vertex v:

If v not visited, DFS(v, visited, orderStack)

Return orderStack

Function DFS(v, visited, orderStack):

Mark v as visited

For each adjacent vertex k of v:

If k not visited, recursively call DFS(k, visited, orderStack)

Push v onto orderStack

12.

Function hasUniqueTopologicalOrder(graph):

If hasHamiltonianPath(graph):

Return True

For each consecutive pair of vertices in topological order:

Swap their positions

If modified order is a valid topological order:

Return False

Swap back to original positions

Return True

Function hasHamiltonianPath(graph):

Implement algorithm to check for Hamiltonian path (not provided here)

Function swapConsecutivePairs(order, i):

Swap order[i] and order[i+1]

Function isTopologicalOrder(graph, order):

Implement algorithm to check if 'order' is a valid topological order for 'graph' (not provided here)

13.

Để chứng minh rằng có tất cả 2^(V\*(V-1)/2) đồ thị có hướng với *V* đỉnh không chứa cạnh song song, ta sử dụng các bước sau:

1. Mỗi đỉnh có thể kết nối với *V*−1 đỉnh khác, vì không chứa cạnh song song.
2. Chọn các cạnh một cách độc lập cho từng đỉnh.

Với mỗi đỉnh, có *V*−1 cách để kết nối với đỉnh khác. Do đó, tổng số cách chọn cạnh cho *V* đỉnh là (V-1)^V. Tuy nhiên, mỗi cạnh được tính hai lần trong số này (một lần cho mỗi đỉnh mà nó kết nối). Do đó, số đồ thị có hướng khác nhau là 2^(V\*(V-1)/2).

Vậy nên, có 2^(V\*(V-1)/2) đồ thị có hướng với *V* đỉnh không chứa cạnh song song.

14.  
Số lượng đồ thị có hướng không chứa chu trình với *V* đỉnh có thể được tính bằng công thức V^(VC2), trong đó VC2 là số cách chọn 2 đỉnh từ *V* đỉnh.

Do đó, có V^(VC2) đồ thị có hướng phi chu trình chứa *V* đỉnh.

15.

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

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

int V = graph.size();

queue<int> q;

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

for (int neighbor : graph[i])

indegree[neighbor]++;

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

if (indegree[i] == 0)

q.push(i);

while (!q.empty()) {

int u = q.front();

q.pop();

result.push\_back(u);

for (int neighbor : graph[u])

if (--indegree[neighbor] == 0)

q.push(neighbor);

}

}

int main() {

int V, E;

cin >> V >> E;

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

vector<int> indegree(V, 0);

vector<int> result;

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

int u, v;

cin >> u >> v;

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

}

topologicalSort(graph, indegree, result);

for (int vertex : result)

cout << vertex << " ";

return 0;

}