# A,Đồ thị vô hướng

## 1, Chu trình Euler và chu trình Hamilton

-Sử dụng code C++ (bai1.cpp) để chạy test ta thấy rằng cả 3 đồ thị nếu không có chu trình euler và chu trình Hamilton

## 2, Đếm đồ thị

Có bao nhiêu đồ thị vô hướng khác nhau có V đỉnh và E cạnh (không có cạnh song song)?

Số lượng đồ thị vô hướng khác nhau có V đỉnh và E cạnh (không có cạnh song song) có thể được tính bằng công thức "C(V, E) \* 2^(E - V + 1)", trong đó C(V, E) là hệ số nhị thức và có giá trị là tổ hợp chập V lấy E.

Để tính toán số lượng này, chúng ta cần xác định E cạnh nào sẽ được chọn từ tổng số C (V, E) cách chọn cạnh từ V đỉnh. Sau đó, mỗi cạnh có thể có hai trạng thái: có hoặc không có trong đồ thị. Do không có cạnh song song, nên số lượng cạnh tối đa là V - 1.

Vì vậy, tổng số đồ thị vô hướng khác nhau có V đỉnh và E cạnh (không có cạnh song song) là

C (V, E) \* 2 ^ (E - V + 1).

## 3, Phát hiện cạnh song song

Thuật toán sau đây sử dụng một loạt các vòng lặp để duyệt qua đỉnh và cạnh của đồ thị, và áp dụng các điều kiện để xác định xem cặp cạnh nào là cặp cạnh song song.

Hàm đếm số cạnh song song có thể được thực hiện theo các bước sau

1. Duyệt qua từng đỉnh u của đồ thị:

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

Bắt đầu vòng lặp, duyệt qua từng đỉnh của đồ thị.

2. Duyệt qua từng đỉnh kề v của đỉnh u:

for (int v: graph[u]) {

Với mỗi đỉnh u, duyệt qua danh sách kề của nó để xem có cạnh nào kề với nó hay không.

3. Kiểm tra điều kiện u < v:

if (u < v) {

Điều này đảm bảo rằng mỗi cặp cạnh chỉ được kiểm tra một lần. Ngăn chặn việc kiểm tra cặp cạnh hai lần với thứ tự ngược lại (v, u).

4. Tạo một set chứa các đỉnh kề của `u` (`dinhKeU`):

unordered\_set<int> dinhKeU(graph[u].begin(), graph[u].end());

Sử dụng `unordered\_set` để dễ dàng kiểm tra sự hiện diện của một đỉnh trong danh sách kề của `u`.

5. Duyệt qua từng đỉnh kề w của đỉnh v:

for (int w: graph[v]) {

Với mỗi đỉnh kề của v, kiểm tra xem cạnh (u, v) và (v, w) có phải là cạnh song song không.

6. Kiểm tra có cạnh song song không:

if (w != u && dinhKeU.find(w) == dinhKeU.end()) {

- w != u: Đảm bảo đỉnh w không trùng với u.

- dinhKeU.find(w) == dinhKeU.end(): Kiểm tra xem w có kề với u hay không. Nếu không, thì cạnh (u, v) và (v, w) là cạnh song song.

7. Tăng biến đếm nếu cạnh song song được tìm thấy:

soCanhSongSong++;

8. Kết quả:

return soCanhSongSong;

Trả về số cạnh song song được đếm trong đồ thị.

## 4, Chu trình lẻ

Để chứng minh rằng một đồ thị là đồ thị hai mầu (bipartite) khi và chỉ khi nó không chứa chu trình độ dài lẻ, ta sẽ sử dụng phương pháp phản chứng như gợi ý.

Giả sử rằng có một đồ thị G có chu trình độ dài lẻ và vẫn được tô màu theo hai mầu sao cho không có cạnh nối giữa hai đỉnh cùng mầu. Chúng ta sẽ chứng minh rằng G không thể là một đồ thị hai mầu.

Xét một chu trình độ dài lẻ bất kỳ trong G và xác định hai đỉnh bất kỳ trên chu trình đó. Gọi hai đỉnh này là u và v. Vì G là đồ thị hai mầu, ta gán màu khác nhau cho u và v.

Trên chu trình đó, xét hai đỉnh kề nhau, gọi chúng là x và y. Theo giả thiết, hai đỉnh x và y cùng thuộc cùng một mầu. Nhưng khi đó, hai đỉnh x và y được nối bởi một cạnh, điều này mâu thuẫn với giả thiết rằng không có cạnh nối giữa hai đỉnh cùng mầu.

Do đó, giả thiết ban đầu là sai và ta có thể kết luận rằng nếu một đồ thị không chứa chu trình độ dài lẻ, thì nó là một đồ thị hai mầu.

Tương tự, chúng ta có thể chứng minh ngược lại rằng nếu một đồ thị là đồ thị hai mầu, thì nó không chứa chu trình độ dài lẻ.

Vì vậy, ta có thể kết luận rằng một đồ thị là đồ thị hai mầu khi và chỉ khi nó không chứa chu trình độ dài lẻ.

## 5, Biconnected

Một đồ thị được gọi là biconnected nếu mỗi cặp đỉnh đều được nối với nhau bởi hai đường đi không giao nhau. Trong đồ thị liên thông, điểm articulation là đỉnh mà khi xóa nó và các cạnh kề sẽ làm đồ thị mất tính liên thông. Hãy chứng minh rằng một đồ thị bất kì mà không có điểm articulation là đồ thị biconnected.

Chứng minh:

Để chứng minh rằng một đồ thị bất kỳ mà không có điểm articulation là đồ thị biconnected, ta sẽ sử dụng phản chứng.

Giả sử rằng đồ thị G không có điểm articulation, nhưng không phải là đồ thị biconnected. Điều này có nghĩa là tồn tại một cặp đỉnh s và t trong G sao cho không có đường đi nào nối giữa chúng mà không đi qua một điểm articulation.

Giả sử ta có một đường đi P nối giữa s và t trong G. Vì G không biconnected, ta có thể chia đường đi P thành hai phần, gọi chúng là P1 và P2, sao cho không có đỉnh nào thuộc P1 nối với đỉnh nào thuộc P2 (nghĩa là không có cạnh nào cắt qua P1 và P2).

Giả sử x là một đỉnh thuộc P1 và y là một đỉnh thuộc P2. Khi đó, ta có đường đi từ s đến x qua P1, sau đó từ y đến t qua P2. Nhưng không có cạnh nối giữa các đỉnh thuộc P1 và P2, điều này mâu thuẫn với giả thiết rằng không có đường đi nào nối giữa s và t mà không đi qua một điểm articulation.

Do đó, giả thiết ban đầu là sai và ta có thể kết luận rằng một đồ thị bất kỳ mà không có điểm articulation là đồ thị biconnected.

Vì vậy, ta có thể chứng minh rằng một đồ thị không có điểm articulation là đồ thị biconnected.

Hoặc ta có thể nhận thấy các đỉnh nằm cùng trên 1 chu trình đơn sẽ cùng thuộc 1 thành phần song liên thông (biconnected) vì cho dù có xóa cạnh nào đi nữa thì vẫn còn 1 đường đi giữa 2 cặp đỉnh bất kì trong 1 chu trình đơn do trong 1 chu trình đơn của 1 đồ thị vô hướng, luôn tồn tại nhiều hơn 1 đường đi giữa 2 cặp đỉnh bất kì. Do đó, nếu 1 đồ thị là đồ thị biconnected, thì cả đồ thị đó sẽ là 1 chu trình đơn. Theo như đề bài, nếu không có điểm articulation (đồ thị có xóa 1 cạnh bất kì nào đi nữa thì cũng vẫn liên thông) => Đồ thị là 1 chu trình đơn. Mà do đồ thị là 1 chu trình đơn nên nó sẽ là 1 đồ thị Biconnected => Điều phải chứng minh.

# B, Đồ thị có hướng

## 8, Sắp xếp tô pô và BFS

**Thứ tự Tô-pô** của một đồ thị có hướng là một thứ tự sắp xếp của các đỉnh sao cho với mọi cung từ đỉnh u đến đỉnh v trong đồ thị, u luôn nằm trước v.

* Dễ nhận thấy rằng, một đồ thị **tồn tại thứ tự Tô-pô** thì đồ thị đó phải là đồ thị có hướng và không có chu trình *(Directed Acyclic Graph - DAG)*. Vì nếu đồ thị có chứa một chu trình, v1v2…vnv1 thì v2 phải đứng sau v1 theo thứ tự Tô-pô, v3 phải đứng sau v2, v.v…; nhưng sau đó v1 phải đứng sau v1, dẫn đến kết luận vô lý khi v1phải đứng sau chính nó trong danh sách.

Do vậy, nếu đồ thị có chu trình, thì không thể tạo ra một thứ tự tô pô hợp lệ. Và việc chỉ sử dụng thuật toán BFS và sắp xếp các đỉnh theo khoảng cách không đảm bảo rằng đồ thị sẽ không có chu trình. Do đó, thứ tự được tạo ra không phải là một thứ tự tô pô hợp lệ.

## 10, Liên thông mạnh

-ĐỘ phức tạp tuyến tính:

Thuật toán Tarjan để tìm thành phần liên thông mạnh:

1. Khởi tạo các biến và cấu trúc dữ liệu:

- Mảng `index`: Lưu trữ thứ tự thăm của mỗi đỉnh.

- Mảng `lowLink`: Lưu trữ giá trị thấp nhất của các đỉnh có thể đạt được từ các đỉnh thuộc cùng một thành phần liên thông mạnh.

- Mảng `onStack`: Đánh dấu xem một đỉnh có đang nằm trên ngăn xếp hay không.

- Ngăn xếp (`stack`): Lưu trữ các đỉnh đang được duyệt qua trong quá trình DFS.

2. Thực hiện DFS trên tất cả các đỉnh chưa được thăm:

- Mỗi khi gặp một đỉnh chưa được thăm, đánh dấu nó và thực hiện DFS từ đỉnh đó.

- Trong quá trình DFS, cập nhật `index` và `lowLink` của các đỉnh.

- Nếu một đỉnh chưa được thăm và đang trên ngăn xếp (`onStack`), cập nhật `lowLink` của đỉnh hiện tại dựa trên `index` của đỉnh đó.

3. Kiểm tra và xử lý thành phần liên thông mạnh:

- Khi kết thúc DFS từ một đỉnh, kiểm tra xem có thành phần liên thông mạnh mới được tìm thấy hay không.

- Nếu có, lấy các đỉnh từ ngăn xếp cho đến khi gặp đỉnh hiện tại để tạo thành một thành phần liên thông mạnh.

Dưới đây là một đoạn mã giả của thuật toán:

function Tarjan(graph):

index = {} // Lưu trữ thứ tự thăm của mỗi đỉnh

lowLink = {} // Lưu trữ giá trị thấp nhất của các đỉnh

onStack = {} // Đánh dấu xem một đỉnh có đang trên ngăn xếp hay không

stack = [] // Ngăn xếp để lưu trữ các đỉnh đang được duyệt qua

result = [] // Danh sách để lưu trữ các thành phần liên thông mạnh

function dfs(v):

index[v] = lowLink[v] = currentIndex

currentIndex++

stack.push(v)

onStack[v] = true

for each neighbor of v:

if neighbor is not visited:

dfs(neighbor)

lowLink[v] = min(lowLink[v], lowLink[neighbor])

else if neighbor is on stack:

lowLink[v] = min(lowLink[v], index[neighbor])

// Kiểm tra và xử lý thành phần liên thông mạnh

if lowLink[v] == index[v]:

component = [] // Thành phần liên thông mạnh mới

while True:

u = stack.pop()

onStack[u] = false

component.push(u)

if u == v:

break

result.push(component)

currentIndex = 0

for each vertex in graph:

if vertex is not visited:

dfs(vertex)

return result // Danh sách các thành phần liên thông mạnh

Trong đoạn mã trên, `index`, `lowLink`, và `onStack` là các mảng để lưu trữ thông tin về các đỉnh trong quá trình thực hiện DFS. `stack` là ngăn xếp được sử dụng trong quá trình duyệt DFS để xác định thành phần liên thông mạnh. `result` là danh sách cuối cùng chứa các thành phần liên thông mạnh.

-ĐỘ phức tạp bậc 2:

Thuật toán Kosaraju:

Duyệt đồ thị bằng thuật toán DFS (Depth-First Search) và lưu lại thứ tự các đỉnh khi hoàn thành việc duyệt DFS (thứ tự dựa trên thời điểm kết thúc duyệt DFS của mỗi đỉnh).

Tạo đồ thị đảo ngược (reverse graph) bằng cách đảo hướng của tất cả các cạnh trong đồ thị ban đầu. Điều này có thể được thực hiện bằng cách tạo một đồ thị mới với cùng số đỉnh, nhưng các cạnh có hướng ngược lại so với đồ thị gốc.

Khởi tạo một danh sách hoặc một stack rỗng để lưu trữ các thành phần liên thông mạnh.

Duyệt đồ thị đảo ngược (reverse graph) bằng thuật toán DFS theo thứ tự ngược lại của thứ tự đã lưu ở bước1.

Mỗi lần duyệt DFS trong bước 4 hoàn thành việc duyệt một thành phần liên thông mạnh. Ghi lại các đỉnh trong thành phần liên thông mạnh đó và thêm nó vào danh sách hoặc stack đã khởi tạo ở bước 3.

Trả về danh sách hoặc stack chứa các thành phần liên thông mạnh của đồ thị.

Nếu đồ thị được biểu diễn dưới dạng danh sách kề thì độ phức tạp thời gian là O(V+E) nhưng nếu dưới dạng ma trận kề thì sẽ là bậc 2 tương ứng O(V^2)

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

Để xác định xem có tồn tại một đường đi qua mỗi đỉnh đúng một lần trong một đồ thị có hướng phi chu trình (DAG), ta có thể sử dụng thuật toán sắp xếp tô pô (topological sort) và kiểm tra xem có cạnh nối giữa mỗi cặp đỉnh liên tiếp trong thứ tự tô pô hay không. Dưới đây là mô tả thuật toán:

1. Thực hiện thuật toán sắp xếp tô pô để tạo ra một thứ tự tô pô của các đỉnh trong đồ thị phi chu trình. Điều này có thể được thực hiện bằng cách sử dụng thuật toán DFS hoặc thuật toán Kahn's để tìm kiếm các đỉnh không có cạnh vào (in-degree = 0) và loại bỏ chúng khỏi đồ thị.

2. Kiểm tra xem có cạnh nối giữa mỗi cặp đỉnh liên tiếp trong thứ tự tô pô hay không. Để làm điều này, duyệt qua từng đỉnh trong thứ tự tô pô và kiểm tra xem có cạnh nối từ đỉnh hiện tại đến đỉnh tiếp theo trong thứ tự tô pô hay không. Nếu không có cạnh nối, thì đồ thị không thỏa mãn yêu cầu và không tồn tại đường đi thỏa mãn đi qua mỗi đỉnh đúng một lần.

3. Nếu tất cả các cạnh nối giữa các cặp đỉnh liên tiếp đều tồn tại, thì đồ thị thỏa mãn yêu cầu và tồn tại một đường đi có hướng đi qua mỗi đỉnh đúng một lần.

Thuật toán trên có độ phức tạp thời gian tuyến tính, vì ta chỉ cần thực hiện thuật toán sắp xếp tô pô một lần (độ phức tạp O(V + E)) và duyệt qua từng cạnh một lần (độ phức tạp O(E)). Tổng cộng, độ phức tạp thời gian là O(V + E).

## 12, Thứ tự Tô pô duy nhất

Áp dụng gợi ý được cho sẵn: một đồ thị có hướng có thứ tự tô pô duy nhất khi và chỉ khi có một cạnh có hướng nối giữa mỗi cặp đỉnh liên tiếp trong thứ tự tô pô (nghĩa là khi đồ thị có một đường đi Hamilton). Nếu đồ thị có hướng có nhiều thứ tự tô pô thì thứ tự tô pô thứ hai có thể thu được từ thứ tự thứ nhất bằng cách đổi vị trí một cặp đỉnh liên tiếp.

Ta sẽ tìm và kiểm tra xem liệu 1 đồ thị có đường đi Hamilton hay không, thông qua việc sử dụng hướng tiếp cận “backtracking” trong việc thử nghiệm tất cả các cách sắp xếp khác nhau của đỉnh

Bước 1: Khởi tạo 1 mảng path với tất cả giá trị ban đầu là -1

Bước 2: Xây dựng hàm canAdd kiểm tra xem liệu có thể thêm đỉnh v vào vị trí pos trong đường đi hay không:

* Kiểm tra xem có cạnh nối path[pos-1] đến v hay không
* Kiểm tra xem liệu v đã được chọn hay chưa

Bước 3: Xây dựng hàm HamiltonPath

* Nếu pos bằng NumV (số lượng đỉnh trong đồ thị)
* Duyệt qua tất cả các đỉnh từ 1 đến NumV

+ Kiểm tra xem liệu có thể thêm đỉnh đó vào vị trí pos hay không

+ Nếu có thể, them đỉnh đó vào path và đệ quy cho pos + 1

+ Nếu không thể, duyệt thứ tự đỉnh kế tiếp

Bước 4:

* Gọi hàm HamiltonPath với pos ban đầu là 1
* Trả về true nếu có đường đi Hamilton cũng như thứ tự topo duy nhất
* Ngược lại, trả về false

Mã giả như sau:

function hasHamiltonianPath(graph):

numV = number of vertices in graph

path = array of size numVertices, all initialized to -1

function canAdd(v, pos, path):

if graph[path[pos-1]][v] == 0:

return false

for i in range(pos):

if path[i] == v:

return false

return true

function HamiltonPath (pos):

if pos == numV:

return true

for v in range(1, numV):

if canAdd(v, pos, path):

path[pos] = v

if HamiltonPath(pos + 1):

return true

path[pos] = -1

return false

if HamiltonPath (1):

print ("Có thứ tự topo duy nhất.")

else:

print ("Không có thứ tự topo duy nhất.")

graph = adjacency matrix of the graph

hasHamiltonianPath(graph)

## 13, Đếm đồ thị có hướng

Để chứng minh rằng có tất cả 2^(V^2) đồ thị V đỉnh có hướng không chứa cạnh song song, ta sẽ sử dụng một phép đếm đơn giản.

Gợi ý: Có bao nhiêu đồ thị có hướng chứa V đỉnh và E cạnh?

Để xây dựng một đồ thị có hướng chứa V đỉnh và E cạnh, ta có thể xem mỗi cạnh trong đồ thị có thể có hai trạng thái: tồn tại hoặc không tồn tại. Do đó, với E cạnh, ta có 2^E cách để xây dựng các cạnh trong đồ thị.

Với mỗi cách xây dựng các cạnh, mỗi cạnh có thể có hai hướng: từ đỉnh A đến đỉnh B hoặc từ đỉnh B đến đỉnh A. Do đó, với mỗi cách xây dựng các cạnh, ta có 2^E x 2^E = 2^(2E) cách để xây dựng đồ thị có hướng.

Tuy nhiên, số lượng cạnh E không phải là một giá trị cố định. Để tính toán số đồ thị có hướng không chứa cạnh song song, ta cần xem xét tất cả các giá trị E từ 0 đến V^2.

Với mỗi giá trị E, có C (V^2, E) cách để chọn E cạnh từ tổng số V^2 cạnh có thể có trong đồ thị V đỉnh. Do đó, tổng số đồ thị có hướng không chứa cạnh song song là:

Σ C (V^2, E) \* 2^(2E), trong đó, Σ là tổng từ E = 0 đến E = V^2.

Tuy nhiên, công thức trên dường như rất phức tạp để tính toán. Thay vào đó, ta có thể sử dụng một phép đếm khác để giải quyết vấn đề này.

Gọi G là một đồ thị có hướng không chứa cạnh song song với V đỉnh. Ta xem xét mỗi cạnh trong G và đánh dấu hướng của nó. Với mỗi cạnh, ta có hai cách đánh dấu hướng: hướng từ đỉnh A đến đỉnh B hoặc hướng từ đỉnh B đến đỉnh A. Do đó, với V cạnh, ta có 2^V cách để đánh dấu hướng cho các cạnh trong G.

Vì vậy, tổng số đồ thị có hướng không chứa cạnh song song là:

2^V \* 2^V \* ... \* 2^V (V lần)

= (2^V) ^ V = 2 ^ (V \* V) = 2^(V^2)

Vậy, có tất cả 2^(V^2) đồ thị V đỉnh có hướng không chứa cạnh song song.

## 14, Đếm đồ thị có hướng phi chu trình

Để tính số lượng đồ thị có hướng phi chu trình có V đỉnh, chúng ta có thể sử dụng công thức dựa trên đặc điểm của DAG.

Mỗi DAG có thể được tạo ra bằng cách chọn một số cạnh từ tất cả các cạnh có thể có giữa các đỉnh. Đối với một DAG có V đỉnh, số lượng cạnh tối đa có thể là V x (V-1)/2 vì mỗi đỉnh có thể có cạnh đến mọi đỉnh khác, nhưng không có chu trình nên không có cạnh từ một đỉnh đến chính nó.

Vì vậy, để tính số lượng đồ thị có hướng phi chu trình với V đỉnh, chúng ta có thể sử dụng công thức:

do mỗi cạnh có thể hoặc không có trong đồ thị, và có V x (V-1)/2 cạnh có thể xuất hiện.