



Dudu Service Maker

Link submit: <https://www.urionlinejudge.com.br/judge/en/problems/view/1610>

Solution:

	Sử dụng visited và inPath	Sử dụng visited cải tiến
C++	https://ideone.com/DoxvEK	https://ideone.com/ZbDiSl
Java	https://ideone.com/TbJSu5	https://ideone.com/Y1TGNm
Python	https://ideone.com/kF4C2r	https://ideone.com/CB2LDb

Tóm tắt đề:

Cho một danh sách gồm N tài liệu cần chuẩn bị để hoàn thành nhiệm vụ. Trong đó một số tài liệu thì lại cần có tài liệu khác đi kèm. Hãy xác định xem danh sách này có chứa chu trình – tức tài liệu A cần có tài liệu B nào đó, đồng thời B lại phải cần có A – hay không. Hay nói cách khác là cho đồ thị có hướng N đỉnh, xác định xem có tồn tại chu trình trong đồ thị hay không.

Input:

Dòng đầu tiên chứa số nguyên T là số lượng test case của đồ thị ($T \leq 100$).

Mỗi test case bắt đầu bằng hai số nguyên N, M ($1 \leq N \leq 10^4, 1 \leq M \leq 3 \cdot 10^4$) là số lượng tài liệu (số đỉnh) và số mối quan hệ (số cạnh của đồ thị). M dòng tiếp theo, mỗi dòng chứa hai số nguyên A, B ($1 \leq A, B \leq N, A \neq B$) – tài liệu A phụ thuộc vào tài liệu B (hay có cạnh từ A đến B).

Output:

Với mỗi test case, nếu không tồn tại chu trình thì in ra “NAO”, ngược lại in ra “SIM”.

Ví dụ:

3	NAO
2 1	SIM
1 2	SIM
2 2	
1 2	
2 1	
4 4	
2 3	
3 4	
4 2	
1 3	

Giải thích ví dụ:

Gồm 3 đồ thị:

Đồ thị 1 gồm 2 đỉnh và 1 cạnh (1, 2), không có cạnh từ 2 về 1 nên không tạo thành chu trình. Kết quả là NAO.

Đồ thị 2 gồm 2 đỉnh và 2 cạnh, tồn tại chu trình 1-2 nên kết quả là SIM.

Đồ thị 3 gồm 4 đỉnh và 4 cạnh, tồn tại chu trình 2-3-4 nên kết quả là SIM.

Hướng dẫn giải:

Thấy rằng, đồ thị chỉ có thể tồn tại chu trình chỉ khi nào có một cạnh nối từ u đến một đỉnh v nào đó được thăm trước đó, đồng thời từ v phải đến được u . Như vậy, ý tưởng giải cơ bản sẽ là sử dụng DFS (theo phương pháp đệ quy) để duyệt qua từng đỉnh, với mỗi đỉnh u đang xét, ta duyệt qua từng đỉnh v kề với u :

- Nếu v chưa thăm thì ta duyệt DFS(v).
- Nếu v thăm rồi, thì lúc này sử dụng mảng path để kiểm tra xem từ v có đến được u hay không, nếu đến được thì chúng ta có chu trình.

Tuy nhiên, với cách này, thì khi sử dụng mảng path để truy vết đường đi và kiểm tra thì trong trường hợp xấu nhất sẽ tốn $O(N)$, tổng quát lên với mỗi đỉnh như vậy thì là $O(N^2)$. Như vậy ta cần phải có một cách nào đó để kiểm tra nhanh xem từ v có đến được u hay không.

Lúc này ta nhận xét, nếu từ v đến được u , thì v sẽ nằm trên đường đi từ gốc DFS đến u . Như vậy, nếu mình đánh dấu lại các đỉnh thuộc đường đi từ gốc đến u , thì có thể kiểm tra nhanh v có thuộc đường đi đó hay không, đồng nghĩa với việc từ v có đến được u hay không. Lúc này ta sẽ chỉnh sửa như sau: thay vì dùng mảng path, ta dùng một mảng là inPath với $\text{inPath}[i] = \text{true}$ nếu i nằm trên đường đi từ gốc DFS đến đỉnh u đang xét, ngược lại $\text{inPath}[i] = \text{false}$. Khi mình duyệt xong DFS(u), thì lúc trở về đỉnh cha của u , chắc chắn u không nằm trên đường đi từ gốc đến cha của u , nên cần gán lại $\text{inPath}[u] = \text{false}$ trước khi thoát khỏi DFS(u).

Ngoài ra, còn một cách xử lý nữa là sử dụng mảng visited, nhưng thay vì lúc này chỉ đánh dấu 0/1 (false/true) thì lúc này mình đánh dấu 3 giá trị nhằm mục đích sử dụng nó để thực hiện chức năng của cả 2 mảng visited và path ở cách trên:

- $\text{visited}[u] = 0$ nếu u chưa được duyệt (tức $\text{visited}[u] = \text{false}$ và $\text{inPath}[u] = \text{false}$ theo cách vừa trình bày).
- $\text{visited}[u] = 1$ nếu u đã được duyệt và ta đang duyệt các đỉnh kề với u ($\text{visited}[u] = \text{true}$ và $\text{inPath}[u] = \text{true}$).

- $visited[u] = 2$ nếu u đã được duyệt và đã duyệt xong các đỉnh kề với u ($visited[u] = false$ và $inPath[u] = false$).

Độ phức tạp: $O(V + E)$ với V , E lần lượt là số lượng đỉnh và số lượng cạnh của đồ thị. Tuy nhiên, cách sử dụng mảng $visited$ 3 giá trị sẽ ít tốn bộ nhớ hơn.

Big-O Coding