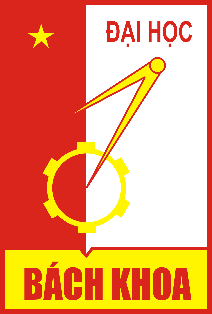
**TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI**

**VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG**

──────── \* ───────



BÁO CÁO MÔN HỌC

**Phân tích và thiết kế thuật toán**

**Tên đề tài : CHU TRÌNH ĐỒ THỊ**

SV thực hiện:

1. **Trần Thanh Tú**
2. **Vũ Tiến Hùng**

GVHD: **TS. ĐỖ PHAN THUẬN**

**HÀ NỘI 12-2019**

Nội dung

[I. Lý thuyết về đồ thị 5](#_Toc27427520)

[1. Khái niệm đồ thị 5](#_Toc27427521)

[2. Phân loại đồ thị: 5](#_Toc27427522)

[2.1 Đơn đồ thị vô hướng: 5](#_Toc27427523)

[2.2 Đa đồ thị vô hướng 5](#_Toc27427524)

[2.3 Giả đồ thị vô hướng 6](#_Toc27427525)

[2.4 Đơn đồ thị có hướng 6](#_Toc27427526)

[2.5 Đa đồ thị có hướng 6](#_Toc27427527)

[3. Bậc của đỉnh 7](#_Toc27427528)

[3.1 Định nghĩa 1 7](#_Toc27427529)

[3.2 Định nghĩa 2 7](#_Toc27427530)

[3.3 Định lý 1 7](#_Toc27427531)

[3.4 Định nghĩa 3 7](#_Toc27427532)

[3.5 Định nghĩa 4 7](#_Toc27427533)

[3.6 Định lý 2 8](#_Toc27427534)

[4. Đường đi, chu trình, đồ thị liên thông 8](#_Toc27427535)

[4.1 Định nghĩa 1: Đường đi 8](#_Toc27427536)

[4.2 Định nghĩa 2: Liên thông 8](#_Toc27427537)

[4.3 Định nghĩa 3 9](#_Toc27427538)

[4.4 Định nghĩa 4 9](#_Toc27427539)

[4.5 Định nghĩa 5 9](#_Toc27427540)

[4.6 Định nghĩa 6 9](#_Toc27427541)

[5. Biểu diễn đồ thị trên máy tinh 9](#_Toc27427542)

[5.1 Ma trận kề, ma trận trọng số: 9](#_Toc27427543)

[5.2 Ma trận liên thuộc đỉnh-cạnh: 11](#_Toc27427544)

[5.3 Danh sách cạnh: 11](#_Toc27427545)

[5.4 Danh sách kề: 12](#_Toc27427546)

[6. Các thuật toán tìm kiếm trên đồ thị 12](#_Toc27427547)

[6.1 Duyệt đồ thị theo chiều sâu (DFS-Depth First Search) 12](#_Toc27427548)

[6.2 Duyệt đồ thị theo chiều rộng: 12](#_Toc27427549)

[6.3 Tìm đường đi giữa hai đỉnh: 13](#_Toc27427550)

[7. Đồ thị Euler và đồ thị Hamilton 13](#_Toc27427551)

[7.1 Đồ thị Euler 13](#_Toc27427552)

[7.2 Đồ thị Hamilton 15](#_Toc27427553)

[8. Chu trình âm và thuật toán Bellman-ford 16](#_Toc27427554)

[8.1 Sơ lược về Bellman-ford 16](#_Toc27427555)

[8.2 Tử tưởng thuật toán 16](#_Toc27427556)

[8.3 Nội dung thuật toán 17](#_Toc27427557)

[8.4 Chứng minh tính đúng đắn 17](#_Toc27427558)

[8.5 Ứng dụng trong định tuyến 18](#_Toc27427559)

[9. Sắp xếp Topo (Topological sorting) 19](#_Toc27427560)

[9.1 Khái niệm 19](#_Toc27427561)

[9.2 Ví dụ 19](#_Toc27427562)

[9.3 Các thuật toán 20](#_Toc27427563)

[II. Trình bày bài toán 21](#_Toc27427564)

[1. Bài toán 1 21](#_Toc27427565)

[1.1. Nội dung bài toán 22](#_Toc27427566)

[1.2. Lý thuyết, phương pháp sử dụng để giải quyết bài toán 23](#_Toc27427567)

[1.3. Thuật toán để giải bài toán 23](#_Toc27427568)

[2. Bài toán 2 24](#_Toc27427569)

[2.1 Giới thiệu bài toán 24](#_Toc27427570)

[2.2 Lý thuyết giải bài toán 24](#_Toc27427571)

[2.3 Phương pháp giải bài toán 26](#_Toc27427572)

[2.4 Chứng minh tính đúng đắn: 26](#_Toc27427573)

[2.5 Đánh giá thời gian tính: 27](#_Toc27427574)

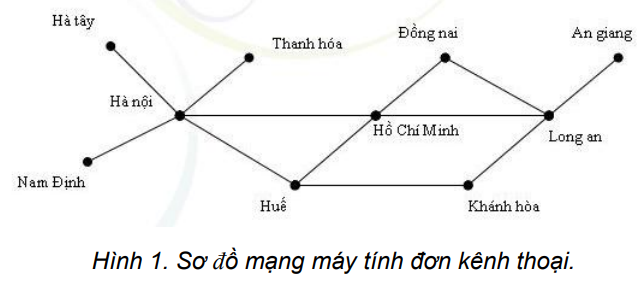
# Lý thuyết về đồ thị

## Khái niệm đồ thị

* Đồ thị là một cấu trúc rời rạc bao gồm các đỉnh và các cạnh nối các đỉnh này.
* Phân biệt các loại đồ thị khác nhau bởi kiểu và số lượng cạnh nối hai đỉnh nào đó của đồ thị.

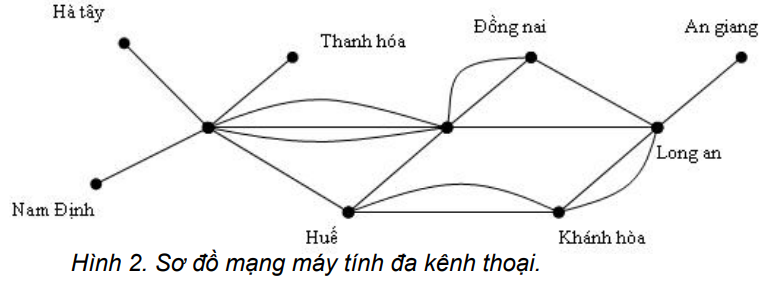
## Phân loại đồ thị:

### Đơn đồ thị vô hướng:

* Đơn đồ thị vô hướng G = (V,E) bao gồm V là tập các đỉnh khác rỗng, và E là tập các cặp không có thứ tự gồm hai phần tử khác nhau của V gọi là các cạnh.

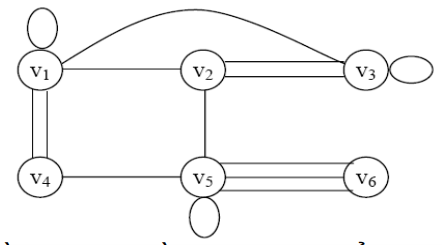
### Đa đồ thị vô hướng

* Đa đồ thị vô hướng G= (V, E) bao gồm V là tập các đỉnh khác rỗng, và E là tập các cặp không có thứ tự gồm hai phần tử khác nhau của V gọi là các cạnh. Hai cạnh e1 và e2 được gọi là cạnh lặp (bội hay song song) nếu chúng cùng tương ứng với một cặp đỉnh.
* Mỗi đơn đồ thị là đa đồ thị, nhưng không phải đa đồ thị nào cũng là đơn đồ thị, vì trong đa đồ thị có thể có hai (hoặc nhiều hơn) cạnh nối một cặp đỉnh nào đó.



### Giả đồ thị vô hướng

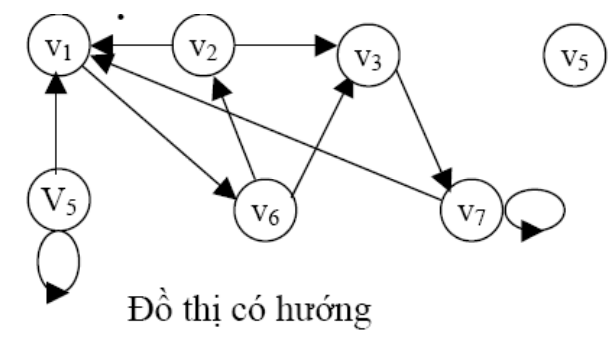
* Giả đồ thị vô hướng G = (V, E) bao gồm V là tập các đỉnh khác rỗng và E là tập các cặp không có thứ tự gồm hai phần tử (không nhất thiết phải khác nhau) của V gọi là cạnh.
* Với v Є V, nếu (v,v) Є E thì ta nói có một khuyên tại đỉnh v.



* Nhận xét: giả đồ thị là loại đồ thị vô hướng tổng quát nhất vì nó có thể chứa các khuyên và các cạnh lặp. Đa đồ thị là loại đồ thị vô hướng có thể chứa cạnh bội nhưng không thể có các khuyên, còn đơn đồ thị là loại đồ thị vô hướng không chứa cạnh bội hoặc các khuyên.

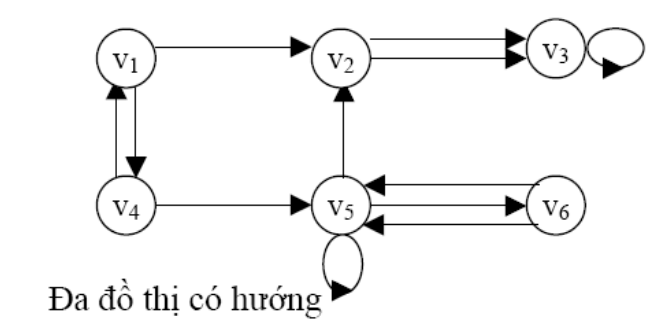
### Đơn đồ thị có hướng

* Đơn đồ thị có hướng G = (V, E) bao gồm V là tập các đỉnh khác rỗng và E là tập các cặp có thứ tự gồm hai phần tử khác nhau của V gọi là các cung.



### Đa đồ thị có hướng

* Đa đồ thị có hướng G = (V, E) bao gồm V là tập các đỉnh khác rỗng và E là tập các cặp có thứ tự gồm hai phần tử khác nhau của V gọi là các cung. Hai cung e1, e2 tương ứng với cùng một cặp đỉnh được gọi là cung lặp.



## Bậc của đỉnh

### Định nghĩa 1

* Hai đỉnh u và v trong đồ thị (vô hướng) G=(V,E) được gọi là liền kề nếu (u,v) Є E. Nếu e = (u,v) thì e gọi là cạnh liên thuộc với các đỉnh u và v. Cạnh e cũng được gọi là cạnh nối các đỉnh u và v. Các đỉnh u và v gọi là các điểm đầu mút của cạnh e.

### Định nghĩa 2

* Bậc của đỉnh v trong đồ thị G=(V,E), ký hiệu deg(v), là số các cạnh liên thuộc với nó, riêng khuyên tại một đỉnh được tính hai lần cho bậc của nó.
* Đỉnh v gọi là đỉnh treo nếu deg(v)=1 và gọi là đỉnh cô lập nếu deg(v)=0.

### Định lý 1

* Giả sử G = (V, E) là đồ thị vô hướng với m cạnh. Khi đó tổng bậc của tất cả các đỉnh bằng hai lần số cạnh.
* **Hệ quả:** Trong đồ thị vô hướng, số đỉnh bậc lẻ (nghĩa là có bậc là số lẻ) là một số chẵn.

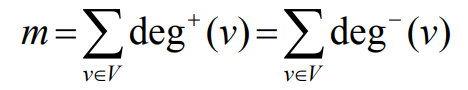
### Định nghĩa 3

* Nếu e = (u, v) là cung của đồ thị có hướng G thì ta nói hai đỉnh u và v là kề nhau, và nói cung (u, v) nối đỉnh u với đỉnh v hoặc cũng nói cung này là đi ra khỏi đỉnh u và vào đỉnh v. Đỉnh u(v) sẽ được gị là đỉnh đầu (cuối) của cung (u,v).

### Định nghĩa 4

* Ta gọi bán bậc ra (bán bậc vào) của đỉnh v trong đồ thị có hướng là số cung của đồ thị đi ra khỏi nó (đi vào nó) và ký hiệu là deg+(v) (deg- (v))

### Định lý 2

* Cho G =(V, E) là một đồ thị có hướng. Khi đó:

**Chứng minh:** Kết quả có ngay là vì mỗi cung được tính một lần cho đỉnh đầu và một lần cho đỉnh cuối.

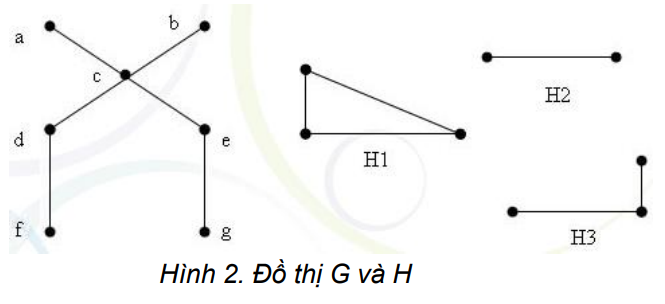
## Đường đi, chu trình, đồ thị liên thông

### Định nghĩa 1: Đường đi

* Đường đi độ dài n từ đỉnh u đến đỉnh v, trong đó n là số nguyên dương, trên đồ thị vô hướng G = (V, E) là dãy x0, x1,…, xn-1, xn; trong đó u = x0, v = xn, (xi , xi+1)∈ E, i = 0, 1, 2,…, n-1. Đường đi nói trên còn có thể biểu diễn dưới dạng dãy các cạnh: (x0, x1), (x1, x2), …, (xn-1, xn)
* Đỉnh u gọi là đỉnh đầu, còn đỉnh v gọi là đỉnh cuối của đường đi.
* Đường đi có đỉnh đầu trùng với đỉnh cuối (tức là u = v) được gọi là chu trình.
* Đường đi hay chu trình được gọi là đơn nếu như không có cạnh nào bị lặp lại.

### Định nghĩa 2: Liên thông

* Đồ thị vô hướng G = (V, E) được gọi là liên thông nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ của nó.



Đồ thị G là liên thông, còn đồ thị H là không liên thông.

### Định nghĩa 3

* Ta gọi đồ thị con của đồ thị G = (V, E) là đồ thị H = (W, F), trong đó W⊆V và F⊆E
* Trong trường hợp đồ thị là không liên thông, nó sẽ rã ra thành một số đồ thị con liên thông không có đỉnh chung. Những đồ thị con liên thông như vậy ta sẽ gọi là các thành phần liên thông của đồ thị.

### Định nghĩa 4

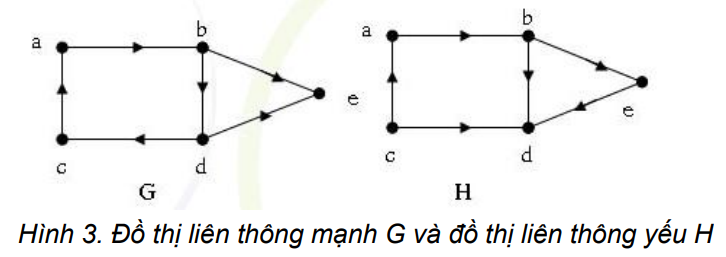
* Đỉnh v được gọi là đỉnh rẽ nhánh nếu việc loại bỏ v cùng với các cạnh liên thuộc với nó khỏi đồ thị làm tăng số thành phần liên thông của đồ thị.
* Cạnh e được gọi là cầu nếu việc loại bỏ nó khỏi đồ thị làm tăng số thành phần liên thông của đồ thị.

### Định nghĩa 5

* Đồ thị có hướng G = (V, A) được gọi là liên thông mạnh nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ của nó.

### Định nghĩa 6

* Đồ thị có hướng G = (V, A) được gọi là liên thông yếu nếu đồ thị vô hướng tương ứng với nó là vô hướng liên thông.

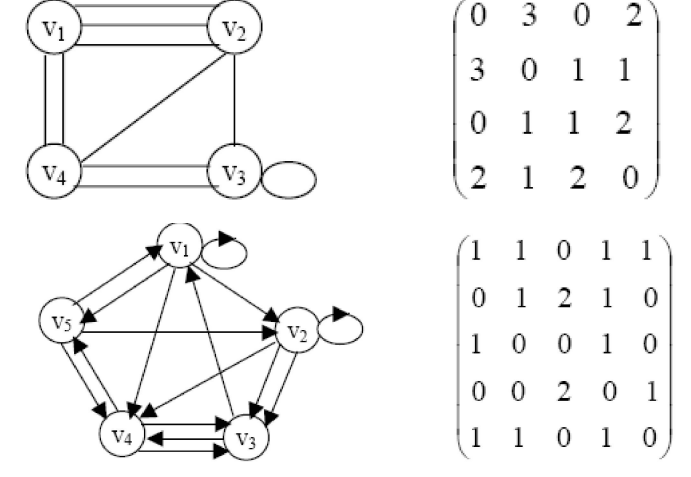


## Biểu diễn đồ thị trên máy tinh

### Ma trận kề, ma trận trọng số:

5.1.1 Ma trận kề:

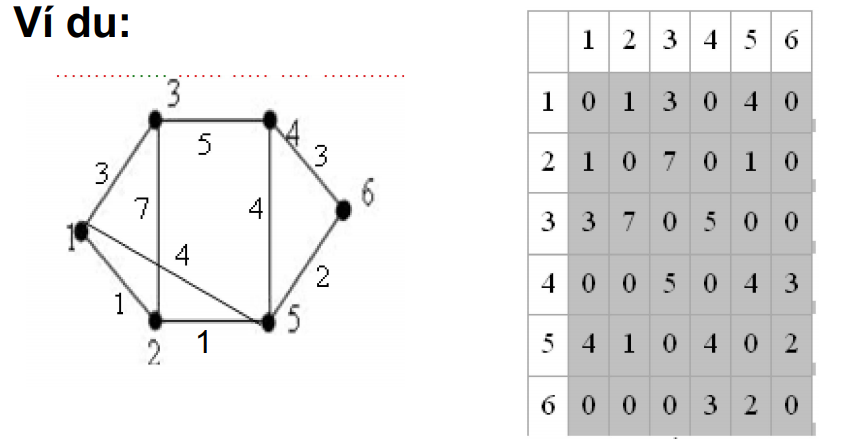
* Ma trận A={ai,j : i,j=1, 2,. . . ,n} với ai,j=0, nếu (i,j) ∉ E và ai,j=1, nếu (i,j)ЄE, i, j=1, 2,. . .,n gọi là ma trận kề của đồ thị G.
* Ví dụ:



5.1.2 Ma trận trọng số

* Đồ thị có trọng số là đồ thị mà mỗi cạnh (i,j) có một giá trị c(i,j) gọi là trọng số của cạnh.
* Để biểu diễn đồ thị ta sử dụng ma trận trọng số C= {c[i,j], i,j=1, 2,. . .,n}
* Với c[i,j]= c(i,j) nếu (i,j) Є E và c[i,j]=θ nếu (i,j) ∉ E

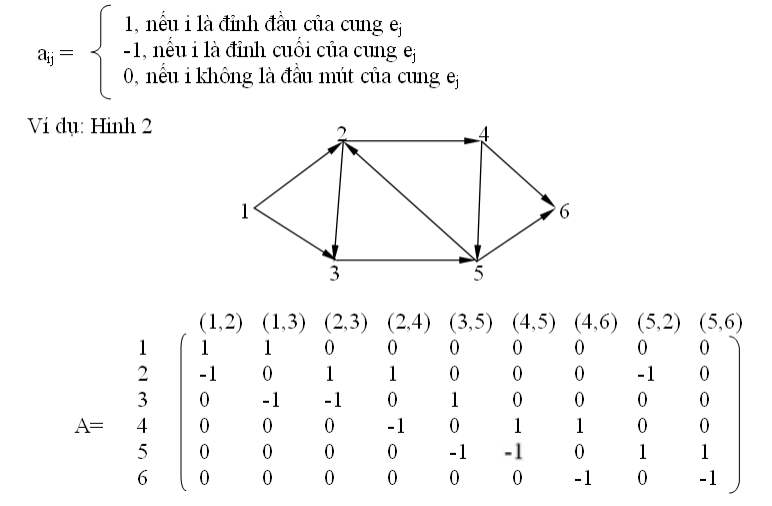
Trong đó số θ có thể được đặt bằng một trong các giá trị sau: 0, +∞, -∞.



* **Ưu điểm** lớn nhất của phương pháp biểu diễn đồ thị bằng ma trận kề (hoặc ma trận trọng số) là để trả lời câu hỏi: Hai đỉnh u,v có kề nhau trên đồ thị hay không, chúng ta chỉ phải thực hiện một phép so sánh.
* **Nhược điểm** lớn nhất của phương pháp này là: không phụ thuộc vào số cạnh của đồ thị, ta luôn phải sử dụng n2 đơn vị bộ nhớ để lưu trữ ma trận kề của nó.

### Ma trận liên thuộc đỉnh-cạnh:

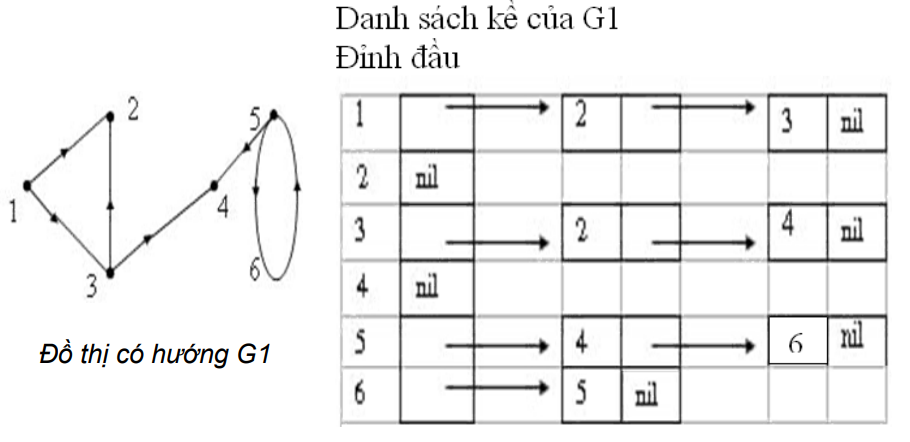
* Xét G=(V, E) là đơn đồ thị có hướng. Ma trận liên thuộc đỉnhcạnh có dạng:



### Danh sách cạnh:

* Trong trường hợp đồ thị thưa (đồ thị có số cạnh m thoả mãn bất đẳng thức: m<6n) biểu diễn đồ thị dưới dạng d/s cạnh.
* Lưu trữ danh sách tất cả các cạnh (cung) của đồ thị. + Một cạnh (cung) e=(x,y) của đồ thị tương ứng với hai biến Dau[e], Cuoi[e].
* Để lưu trữ đồ thị ta cần sử dụng 2m đơn vị bộ nhớ.
* **Nhược điểm:** để tìm các đỉnh kề với một đỉnh cho trước phải làm m phép so sánh (khi duyệt qua danh sách tất cả các cạnh của đồ thị).

### Danh sách kề:



## Các thuật toán tìm kiếm trên đồ thị

### Duyệt đồ thị theo chiều sâu (DFS-Depth First Search)

Procedure DFS(v); (\*tim kiem theo chieu sau bat dau tu dinh v; cac bien Chuaxet, Ke la bien toan cuc\*)

Begin

Tham\_dinh(v); Chuaxet[v]:=false;

For u Є Ke(v) do

If Chuaxet[u] then DFS(u);

End; (\*dinh v da duyet xong\*)

Khi đó, tìm kiếm theo chiều sâu trên đồ thị được thực hiện nhờ thuật toán sau:

Begin (\*Khoi tao tat ca cac dinh cua do thi\*)

for v Є V do Chuaxet[v]:=true;

for v Є V do If Chuaxet[v] then DFS(v);

End.

### Duyệt đồ thị theo chiều rộng:

Procedure BFS(v);(\*BFS bat dau tu dinh v, cac bien Chuaxet, Ke la bien cuc bo\*)

Begin QUEUE:=Ø; QUEUE⇐ v; (\*ket qua nap vao QUEUE\*) Chuaxet[v]:=false;

While QUEUE<> Ø do

Begin

p ⇐ QUEUE; (\*lay p tu QUEUE:\*) Tham\_dinh(p);

For u Є Ke(v) do

If Chuaxet[u] then Begin QUEUE ⇐ u; Chuaxet[u]:=false; End;

End; End;

Khi đó, tìm kiếm theo chiều rộng trên đồ thị được thực hiện nhờ thuật toán sau:

Begin for f Є V do Chuaxet[v]:=true; (\*Khoi tao cac dinh cua do thi la chua xet\*) for v Є V do

if Chuaxet[v] then BFS(v);

End.

### Tìm đường đi giữa hai đỉnh:

Khi đó, thủ tục DFS(v) cần sửa câu lệnh if trong nó như sau:

If Chuaxet[u] then

Begin

Truoc[u]:=v;

DFS(u);

End;

Thủ tục BFS(v) cần sửa đổi câu lện if trong nó như sau:

If Chuaxet [u] then

Begin

QUEUE u;

Chuaxet[u]:=false;

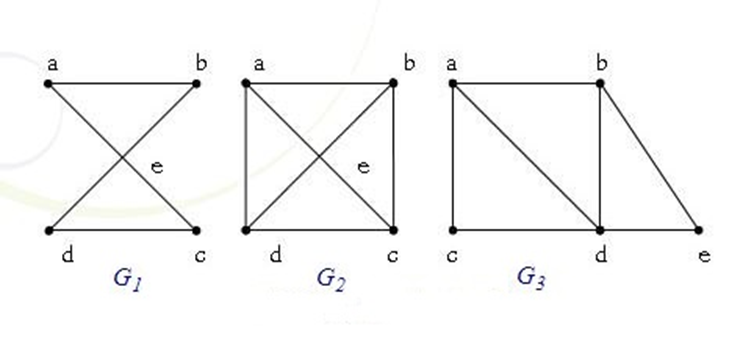
Truoc[u]:=p;

End;

## Đồ thị Euler và đồ thị Hamilton

### Đồ thị Euler

7.1.1. Định nghĩa:

* Đường đi qua mỗi cạnh của đồ thị đúng một lần được gọi là đường đi Euler.
* Chu trình qua mỗi cạnh của đồ thị đúng một lần được gọi là chu trình Euler.
* Đồ thị được gọi là đồ thị Euler nếu nó có chu trình Euler, và gọi là đồ thị nửa Euler nếu nó có đường đi Euler
* **Nhận xét:** mọi đồ thị Euler luôn là nửa Euler, nhưng điều ngược lại không luôn đúng.
* Đồ thị G1 là đồ thị Euler vì nó có chu trình Euler a, e, c, d, e, b, a.
* Đồ thị G2 không có chu trình cũng như đường đi Euler.
* Đồ thị G3 không có chu trình Euler nhưng nó có đường đi Euler a, c, d, e, b, d, a, b, vì thế G3 là đồ thị nửa Euler.

7.1.2. Định lý 1 (Euler):

* G là đồ thị vô hướng liên thông. G là đồ thị Euler ⇔ mọi đỉnh của G đều có bậc chẵn.
* **Bổ đề:** Nếu bậc của mỗi đỉnh của đồ thị G không nhỏ hơn 2 thì G chứa chu trình.
* **Hệ quả:** Đồ thị vô hướng liên thông G là nửa Euler khi và chỉ khi nó có không quá 2 đỉnh bậc lẻ.

7.1.3. Thuật toán Flor:

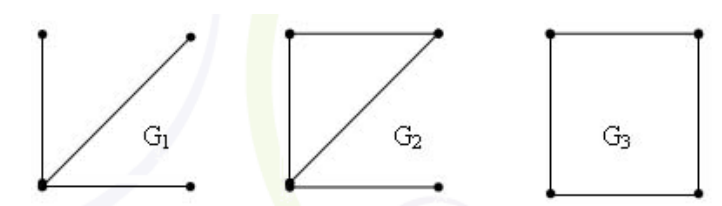
* Xuất phát từ một đỉnh u nào đó của G ta đi theo các cạnh của nó một cách tuỳ ý chỉ cần tuân thủ 2 qui tắc sau:
  + Xoá bỏ cạnh đã đi qua đồng thời xoá bỏ đỉnh cô lập tạo thành.
  + Ở mỗi bước ta chỉ đi qua cầu khi không còn cách lựa chọn nào khác.

7.1.4 Tìm chu trình và đường đi euler bằng Stack

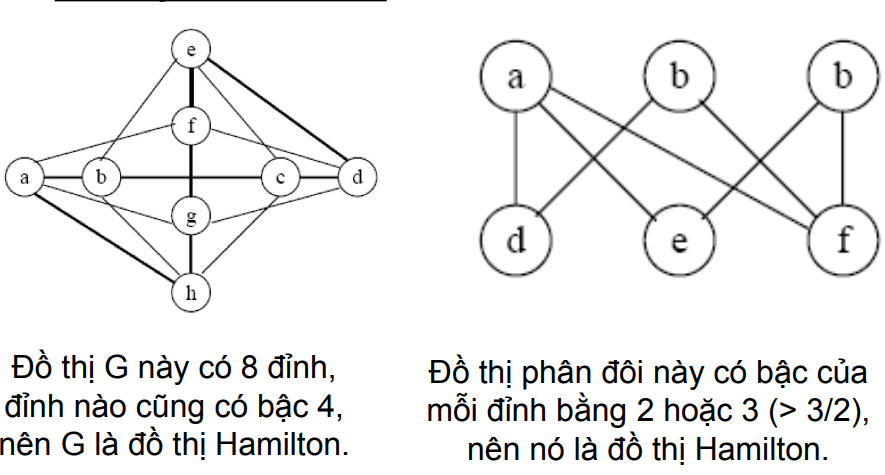
* **Thuật toán tìm chu trình euler**
* B1: Tạo một mảng S để ghi đường đi và một stack T để xếp các đỉnh ta sẽ xét. Xếp vào đó một đỉnh tuỳ ý u nào đó của đồ thị, nghĩa là đỉnh u sẽ được xét đầu tiên.
* B2: Xét đỉnh trên cùng của ngăn xếp, giả sử đỉnh đó là đỉnh v và thực hiện:
  + Nếu v là đỉnh cô lập thì lấy v khỏi ngăn xếp và đưa vào S;
  + Nếu v là liên thông với đỉnh x thì xếp x vào ngăn xếp T sau đó xoá bỏ cạnh (v, x);
* B3: Quay lại bước 2 cho tới khi ngăn xếp rỗng. Kết quả chu trình Euler được chứa trong S theo thứ tự ngược lại.
* **Thuật toán tìm đường đi euler**
* B1: Xác định điểm xuất phát của đường đi từ đỉnh bậc lẻ này và kết thúc ở đỉnh bậc lẻ khác.
* B2: Nạp đỉnh bậc lẻ vừa xác định vào stack, sau đó thực hiện như thuật toán tìm chu trình euler.

### Đồ thị Hamilton

* Đường đi qua tất cả các đỉnh của đồ thị mỗi đỉnh đúng một lần được gọi là đường đi Hamilton.
* Chu trình bắt đầu từ một đỉnh v nào đó qua tất cả các đỉnh còn lại mỗi đỉnh đúng một lần rồi quay trở về v được gọi là chu trình Hamilton.
* Đồ thị G được gọi là đồ thị Hamilton nếu nó chứa chu trình Hamilton và gọi là đồ thị nửa Hamilton nếu nó có đường đi Hamilton.
* Ví dụ 3. Trong hình: Đồ thị G3 là Hamilton, G2 là nửa Hamilton còn G1 không là nửa Hamilton.



* Cho đến nay việc tìm một tiêu chuẩn nhận biết đồ thị Hamilton vẫn còn là mở.
* Phần lớn các phát biểu đều có dạng "nếu G có số cạnh đủ lớn thì G là Hamilton".
* Định lý:
* **Định lý 1** (Dirak 1952). Đơn đồ thị vô hướng G với n>2 đỉnh, mỗi đỉnh có bậc không nhỏ hơn n/2 là đồ thị Hamilton.
* **Định lý 2** Nếu G là đồ thị phân đôi với hai tập đỉnh là V1, V2 có số đỉnh cùng bằng n (n ≥ 2) và bậc của mỗi đỉnh lớn hơn n/2 thì G là một đồ thị Hamilton.
* **Định lý 3**. Giả sử G là đồ có hướng liên thông với n đỉnh. Nếu deg+(v)≥n/2, deg–(v) ≥ n/2, ∀v thì G là Hamilton.



## Chu trình âm và thuật toán Bellman-ford

### Sơ lược về Bellman-ford

* Thuật toán Bellman-Ford là một thuật toán tính các đường đi ngắn nhất nguồn đơn trong một đồ thị có hướng có trọng số (trong đó một số cung có thể có trọng số âm).
* Thuật toán Dijkstra giải cùng bài toán này tuy nhiên Dijkstra có thời gian chạy nhanh hơn đơn giản là đòi hỏi trọng số của các cung phải có giá trị không âm.
* Độ phức tạp của thuật toán Bellman\_ford : O(V\*E) với V là số đỉnh và E là số cạnh của đồ thị

### Tử tưởng thuật toán

* Bước 1 : khởi tạo ∏(0, x) = 0, ∏(0, i) = +∞, ∀i≠x và k=1
* Bước 2 : Với mỗi i ∈ X ta đặt:

∏(k,i)=min({∏(k-1,i)}∪{∏(k-1,j)+L[j][i]})

* Bước 3 : Nếu ∏(k,i)=∏(k-1,i) với i∈X thì ∏(k,i) chính là độ dài đường đi ngắn nhất từ x đến i. Ngược lại nếu k<n thì tăng k=k+1 và trở lại bước 2; nếu k=n thì dừng vì từ x đi tới được 1 mạch âm.
* Ưu điểm :

- Từ 1 đỉnh xuất phát nhìn hình ta có thế suy ra đường đi ngắn nhất từ đỉnh đó tới các đỉnh khác mà không cần làm lại từ đầu.

- Ví dụ: Từ đỉnh 1 ta có thể tìm đường đi ngắn nhất từ 1->3 và 1->4 mà không cần làm lại.

### Nội dung thuật toán

**function** BellmanFord(danh\_sách\_đỉnh, danh\_sách\_cung, nguồn)

// hàm yêu cầu đồ thị đưa vào dưới dạng một danh sách đỉnh, một danh sách cung

// hàm tính các giá trị khoảng\_cách và đỉnh\_liền\_trước của các đỉnh,

// sao cho các giá trị đỉnh\_liền\_trước sẽ lưu lại các đường đi ngắn nhất.

// bước 1: khởi tạo đồ thị

**for each** v in danh\_sách\_đỉnh:

**if** v is nguồn **then** khoảng\_cách(v):= 0

**else** khoảng\_cách(v):= vô cùng

đỉnh\_liền\_trước(v):= null

// bước 2: kết nạp cạnh

**for** i **from** 1 **to** size(danh\_sách\_đỉnh)-1:

**for each** (u,v) **in** danh\_sách\_cung:

**if** khoảng\_cách(v) > khoảng\_cách(u) + trọng\_số(u,v):

khoảng\_cách(v):= khoảng\_cách(u) + trọng\_số(u,v)

đỉnh\_liền\_trước(v):= u

// bước 3: kiểm tra chu trình âm

**for each** (u,v) **in** danh\_sách\_cung:

if khoảng\_cách(v) > khoảng\_cách(u) + trọng\_số(u,v):

error "Đồ thị chứa chu trình âm"

### Chứng minh tính đúng đắn

Tính đúng đắn của thuật toán có thể được chứng minh bằng quy nạp. Thuật toán có thể được phát biểu chính xác theo kiểu quy nạp như sau:

* **Bổ đề**. Sau i lần lặp vòng for:
* Nếu Khoảng\_cách(u) không có giá trị vô cùng lớn, thì nó bằng độ dài của một đường đi nào đó từ s tới u;
* Nếu có một đường đi từ s tới u qua nhiều nhất i cung, thì Khoảng\_cách(u) có giá trị không vượt quá độ dài của đường đi ngắn nhất từ s tới u qua tối đa i cung.
* **Chứng minh**
  + Trường hợp cơ bản: Xét i = 0 và thời điểm trước khi vòng for được chạy lần đầu tiên. Khi đó, với đỉnh nguồn khoảng\_cách(nguồn) = 0, điều này đúng. Đối với các đỉnh u khác, khoảng\_cách(u) = vô cùng, điều này cũng đúng vì không có đường đi nào từ nguồn đến u qua 0 cung.
  + Trường hợp quy nạp:
  + Chứng minh câu 1. Xét thời điểm khi khoảng cách tới một đỉnh được cập nhật bởi công thức khoảng\_cách(v):= khoảng\_cách(u) + trọng\_số(u,v). Theo giả thiết quy nạp, khoảng\_cách(u) là độ dài của một đường đi nào đó từ nguồn tới u. Do đó, khoảng\_cách(u) + trọng\_số(u,v) là độ dài của đường đi từ nguồn tới u rồi tới v.
  + Chứng minh câu 2: Xét đường đi ngắn nhất từ nguồn tới u qua tối đa i cung. Giả sử v là đỉnh liền ngay trước u trên đường đi này. Khi đó, phần đường đi từ nguồn tới v là đường đi ngắn nhất từ nguồn tới v qua tối đa i-1 cung. Theo giả thuyết quy nạp, khoảng\_cách(v) sau i-1 vòng lặp không vượt quá độ dài đường đi này. Do đó, trọng\_số(v,u) + khoảng\_cách(v) có giá trị không vượt quá độ dài của đường đi từ s tới u. Trong lần lặp thứ i, khoảng\_cách(u) được lấy giá trị nhỏ nhất của khoảng\_cách(v) + trọng\_số(v,u) với mọi v có thể. Do đó, sau i lần lặp, khoảng\_cách(u) có giá trị không vượt quá độ dài đường đi ngắn nhất từ nguồn tới u qua tối đa i cung.
  + Khi i bằng số đỉnh của đồ thị, mỗi đường đi tìm được sẽ là đường đi ngắn nhất toàn cục, trừ khi đồ thị có chu trình âm. Nếu tồn tại chu trình âm mà từ đỉnh nguồn có thể đi đến được thì sẽ không tồn tại đường đi nhỏ nhất (vì mỗi lần đi quanh chu trình âm là một lần giảm trọng số của đường).

### Ứng dụng trong định tuyến

* Một biến thể phân tán của thuật toán Bellman-Ford được dùng trong các giao thức định tuyến vector khoảng cách, chẳng hạn giao thức RIP (Routing Information Protocol). Đây là biến thể phân tán vì nó liên quan đến các nút mạng (các thiết bị định tuyến) trong một hệ thống tự chủ (autonomous system), ví dụ một tập các mạng IP thuộc sở hữu của một nhà cung cấp dịch vụ Internet (ISP).
* Thuật toán gồm các bước sau:
  + Mỗi nút tính khoảng cách giữa nó và tất cả các nút khác trong hệ thống tự chủ và lưu trữ thông tin này trong một bảng.
  + Mỗi nút gửi bảng thông tin của mình cho tất cả các nút lân cận.
  + Khi một nút nhận được các bảng thông tin từ các nút lân cận, nó tính các tuyến đường ngắn nhất tới tất cả các nút khác và cập nhật bảng thông tin của chính mình.
* Nhược điểm chính của thuật toán Bellman-Ford trong cấu hình này là:
  + Không nhân rộng tốt
  + Các thay đổi của tô-pô mạng không được ghi nhận nhanh do các cập nhật được lan truyền theo từng nút một.
  + Đếm dần đến vô cùng (nếu liên kết hỏng hoặc nút mạng hỏng làm cho một nút bị tách khỏi một tập các nút khác, các nút này vẫn sẽ tiếp tục ước tính khoảng cách tới nút đó và tăng dần giá trị tính được, trong khi đó còn có thể xảy ra việc định tuyến thành vòng tròn)

## Sắp xếp Topo (Topological sorting)

### Khái niệm

* Trong khoa học máy tính, 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 u đến v trong đồ thị, u luôn nằm trước v.
* Thuật toán để tìm thứ tự tô pô gọi là thuật toán sắp xếp tô pô. Thứ tự tô pô tồn tại khi và chỉ khi đồ thị không có chu trình (viết tắt là DAG - tiếng Anh directed acyclic graph).
* Đồ thị có hướng không có chu trình luôn có ít nhất một thứ tự tô pô, và có thuật toán để tìm thứ tự tô pô trong thời gian tuyến tính.

### Ví dụ

* Một ứng dụng kinh điển của thứ tự tô pô là lập kế hoạch cho một chuỗi các công việc. Các thuật toán sắp xếp tô pô được nghiên cứu lần đầu tiên vào những năm 1960 trong phương pháp PERT cho việc lập kế hoạch trong quản lý dự án. Các công việc được đại diện bởi các đỉnh đồ thị. Đồ thị có cung từ x đến y nếu công việc x phải hoàn thành trước khi công việc y bắt đầu (chẳng hạn như khi giặt quần áo, việc giặt phải hoàn thành trước khi bắt đầu phơi khô). Khi đó, một thứ tự tô pô tương ứng với một thứ tự thực hiện các công việc.
* Trong khoa học máy tính, các ứng dụng tương tự phát sinh trong lập kế hoạch thực thi lệnh, xác định thứ tự biên dịch trong makefile, xác định quan hệ phụ thuộc giữa các biểu tượng trong chương trình liên kết.

### Các thuật toán

* Các thuật toán sắp xếp tô pô thường có thời gian tuyến tính trong số nút cộng với số cung ({\displaystyle O(|V|+|E|)}{\displaystyle O(|V|+|E|)}).
* Một trong những thuật toán này, phát hiện bởi Kahn năm 1962[1], hoạt động bằng cách lần lượt chọn các đỉnh theo thứ tự đúng như thứ tự tô pô. Đầu tiên, xác định một danh sách các "nút bắt đầu" không có cung vào và chèn chúng vào một tập S. Trong một đồ thị có hướng không có chu trình, luôn có ít nhất một nút như vậy. Sau đó:

*L* ← danh sách rỗng (cuối cùng sẽ chứa danh sách đã sắp xếp)

*S* ← tập hợp các nút không có cung vào

**while** *S* khác rỗng **do**

loại bỏ một nút *n* từ *S*

chèn *n* vào *L*

**for each** nút *m* sao cho có cung *e* từ *n* đến *m* **do**

loại bỏ cung *e* từ đồ thị

**if** *m* không có cung vào **then**

chèn *m* vào *S*

**if** đồ thị vẫn còn cung **then**

thông báo lỗi (đồ thị có ít nhất một chu trình)

**else**

thông báo thứ tự tô pô là *L*

* Nếu đồ thị là một DAG, danh sách L luôn chứa một thứ tự hợp lệ(thứ tự tô pô không nhất thiết là duy nhất). Nếu đồ thị không là DAG thì nó có ít nhất một chu trình và do đó không thể có thứ tự tô pô.
* Lưu ý rằng S có thể lựa chọn phần tử n một cách tùy ý. Tùy thuộc vào thứ tự các nút n được loại bỏ từ S, một thứ tự tô pô khác nhau được tạo ra. Một biến thể của thuật toán của Kahn sử dụng thứ tự từ điển cho việc lựa chọn n là một thành phần quan trọng của thuật toán Coffman-Graham cho lập kế hoạch song song và vẽ đồ thị lớp.
* Có một thuật toán khác cho sắp xếp tô pô dựa trên tìm kiếm theo chiều sâu. Đối với thuật toán này, các cung chỉ theo hướng ngược lại so với thuật toán trước: có một cung từ x đến y nếu công việc x phụ thuộc vào công việc y (nói cách khác, nếu công việc y phải hoàn thành trước khi công việc x có thể bắt đầu). Thuật toán duyệt qua các nút của đồ thị, trong một trật tự tùy ý, và thực hiện tìm kiếm theo chiều sâu cho đến khi tìm đến một nút đã được thăm:

*L* ← danh sách rỗng (cuối cùng sẽ chứa thứ tự sắp xếp)

*S* ← tập hợp các nút không có cung vào

**for each** nút *n* trong *S* **do**

thăm(n)

**function** thăm(nút ​​*n*)

**if** chưa thăm *n* **then**

đánh dấu *n* là đã thăm

**for each** nút *m* sao cho có cung từ n đến m **do**

thăm(m)

chèn *n* vào *L*

* **Lưu ý**: mỗi nút n được thêm vào danh sách L chỉ sau khi đã thăm tất cả các nút khác mà n phụ thuộc vào(tất cả các nút hậu duệ của n trong đồ thị). Cụ thể là, khi thuật toán thêm nút n, nó đảm bảo rằng tất cả các nút n phụ thuộc vào đã có trong danh sách L: chúng đã được thêm vào L hoặc do lời gọi đệ quy đến thăm(), hoặc do một lời gọi đến thăm() từ trước. Vì mỗi cung và mỗi nút được thăm một lần, thuật toán chạy trong thời gian tuyến tính. Lưu ý rằng mã giả trên không thể phát hiện trường hợp lỗi khi đồ thị có chu trình. Thuật toán có thể được thay đổi để phát hiện chu trình bằng cách kiểm tra có nút nào được thăm nhiều hơn một lần trong bất kỳ một chuỗi lồng nhau của các lời gọi đệ quy đến thăm(). Thuật toán này có thể đã được mô tả lần đầu tiên bởi Tarjan năm 1976[2].

# Trình bày bài toán

## Bài toán 1

* Tên bài toán: Fox and Names (<https://codeforces.com/problemset/problem/510/C>)
* Sinh viên thực hiện: Vũ Tiến Hùng
* MSSV: 20161996

### Nội dung bài toán

* Time limit : 2s
* Memory limit : 256MB
* Nội dung tóm tắt:
  + Cho 1 danh sách các tên người theo một thứ tự nhất định (danh sách các tác giả của 1 bài báo). Và tác giả của bài báo đó cho rằng danh sách tên của các tác giả của bài báo đã được sắp xếp theo đúng thứ tự từ điển (đúng thứ tự bảng chữ cái aphabet).
  + Khi so sánh 2 tên s và t (tên s đứng trước tên t). Tìm kí tự (thứ i) đầu tiên khác nhau từ trái sang của 2 chuỗi này (si ≠ ti). Nếu không tìm được, chuỗi ngắn phải là chuỗi đứng trước, còn nếu tìm được thì kí tự si luôn phải đứng trước ti trong bảng chữ cái (vì s đứng trước t)
* Input:
  + dòng đầu tiên chứa n (1 ≤ n ≤ 100)
  + n dòng tiếp theo mỗi dòng chứa 1 tên (string) (1 ≤ |name| ≤ 100)
* Output:
  + Nếu có thứ tự bảng chứ cái latin thỏa mãn thứ tự những tên tác giả ở trên thì đưa ra bảng chứ latin đó
  + Nếu không có bảng chữ cái latin nào thỏa mãn được thứ tự các tên trên thì in ra ”Impossible”

### Lý thuyết, phương pháp sử dụng để giải quyết bài toán

* Xây dựng 1 đồ thị có hướng từ danh sách tên trong phần input
* Sử dụng phương pháp sắp xếp topo (topological sorting) để đưa ra thứ tự topo của các chữ cái trong đồ thị đã xây dựng (cụ thể sẽ được trình bày ở phần sau).

### Thuật toán để giải bài toán

* Bước 1: Xây dựng 1 đồ thị có hướng thỏa mãn thứ tự các tên người trong phần input
  + Duyệt lần lượt thứ tự các tên trong input. Với mỗi 2 tên name[i] và name[i + 1] liên tiếp, ta tiếp tục duyệt lần lượt (từ trái sang phải) từng kí tự thứ j của 2 tên này đến khi tìm thấy kí tự đầu tiên khác nhau của 2 tên thứ i và i + 1 (u = name[i][j] ≠ name[i+1][j] = v), ta lưu đồ thị dưới dạng danh sách kề bằng cách adj[u].push\_back(v), adj là vector để lưu trữ đồ thị (adj[u] chữa danh sách các đỉnh mà u có thể đi tới
  + Sau bước duyệt để xây dựng đồ thị trên ta thu được 1 danh sách kề adj chứ thông tin về đồ thị. Và tiến hành đưa ra thứ tự topo của các đỉnh trong đồ thị
  + Dùng cờ flag để đánh dấu trường hợp không tìm được 2 kí tự tương ứng khác nhau nào của 2 tên liên tiếp mà tên dài hơn lại đứng trước
* Bước 2: Đưa ra thứ tự topo của các đỉnh trong đồ thị

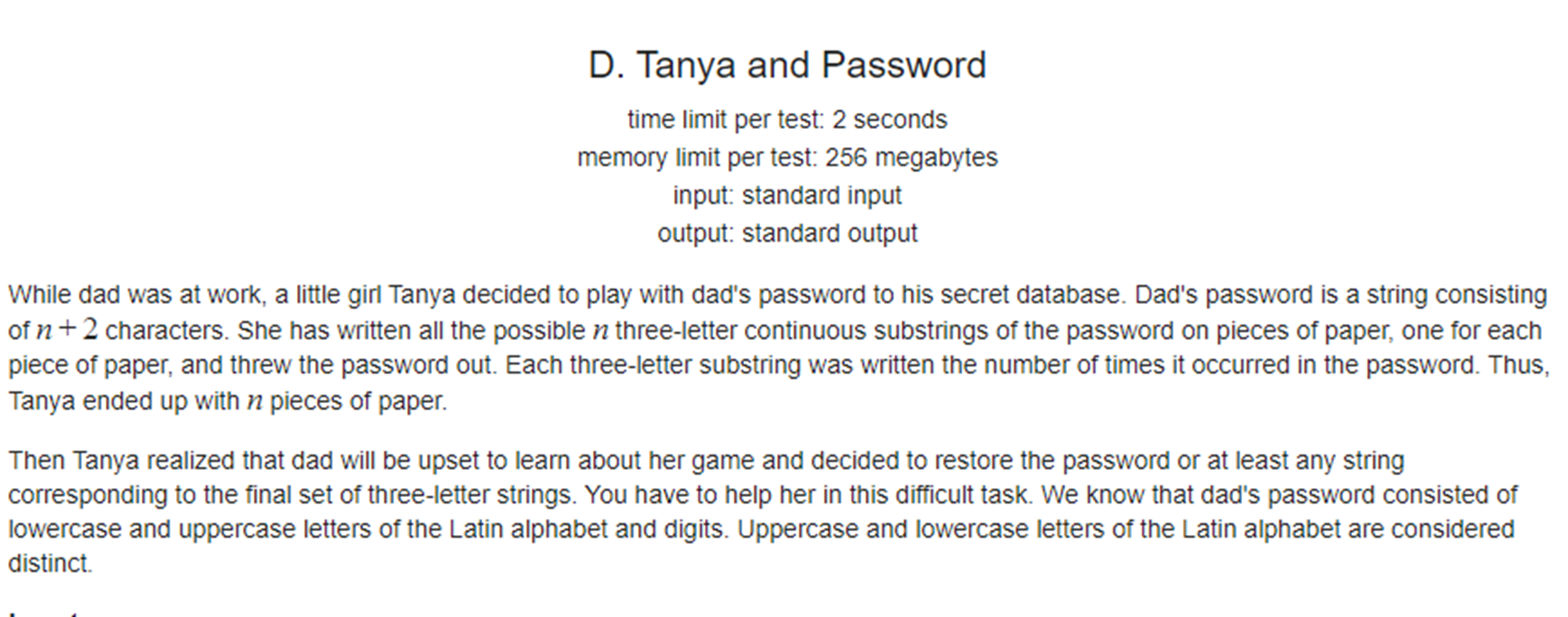
1. Bài toán 2

* Tên bài toán: Tanya and Password

<https://codeforces.com/contest/508/problem/D>

* Sinh viên thực hiện: Trần Thanh Tú
* MSSV: 20164487

### 2.1 Giới thiệu bài toán



Input:

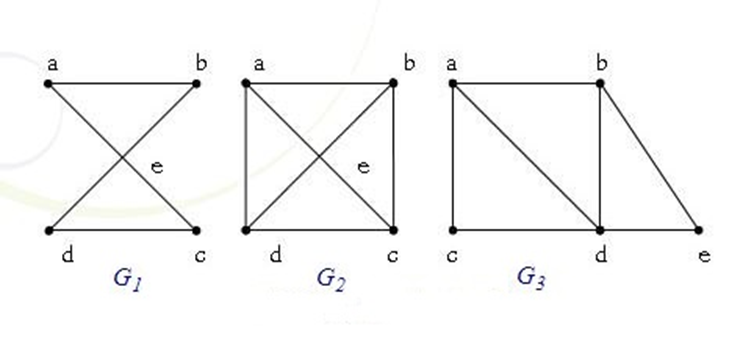
* Dòng đầu tiên là hằng số n (1<n<2\*10^5), số chuỗi con 3 ký tự.
* N dòng tiếp theo mỗi dòng chứa một chuỗi 3 ký tự, tạo thành từ chuỗi con của password. Mỗi ký tự là chữ thường, chữ in hoa và chữ số latinh.

Output:

* Nếu n chuỗi không thể tạo thành password thì in “NO”
* Nếu n chuỗi tạo thành password in “YES” và chuỗi password tạo thành.

### 2.2 Lý thuyết giải bài toán

* Chu trình và đường đi euler:
* Trong [lý thuyết đồ thị](https://vi.wikipedia.org/wiki/L%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B), một đường đi trong [đồ thị](https://vi.wikipedia.org/wiki/%C4%90%E1%BB%93_th%E1%BB%8B_(l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B)) G = (X, E) được gọi là đường đi Euler nếu nó đi qua tất cả các cạnh của đồ thị, mỗi cạnh đúng một lần.
* Đường đi Euler có đỉnh cuối cùng trùng với đỉnh xuất phát gọi là chu trình Euler.



* Đồ thị G1 là đồ thị Euler vì nó có chu trình Euler a, e, c, d, e, b, a.
* Đồ thị G2 không có chu trình cũng như đường đi Euler.
* Đồ thị G3 không có chu trình Euler nhưng nó có đường đi Euler a, c, d, e, b, d, a, b, vì thế G3 là đồ thị nửa Euler.
* Định lý Euler về chu trình và đường đi Euler:
* Đồ thị vô hướng liên thông G = (X, E) có chu trình Euler khi và chỉ khi G không có đỉnh bậc lẻ.
* Đồ thị vô hướng liên thông G = (X, E) có đường đi Euler khi và chỉ khi G có đúng hai đỉnh bậc lẻ. Nếu G có hai đỉnh bậc lẻ thì đường đi Euler có hai đầu đường đi nằm ở hai đỉnh bậc lẻ.
* Đồ thị có hướng liên thông G = (X, E) có chu trình Euler, khi đó số đỉnh bậc trong của G sẽ bằng số đỉnh bậc ngoài của G (d+(x) = d-(x),∀xϵ X).
* Thuật toán tìm chu trình euler

1. Tạo một mảng S để ghi đường đi và một stack T để xếp các đỉnh ta sẽ xét. Xếp vào đó một đỉnh tuỳ ý u nào đó của đồ thị, nghĩa là đỉnh u sẽ được xét đầu tiên.
2. Xét đỉnh trên cùng của ngăn xếp, giả sử đỉnh đó là đỉnh v và thực hiện:

* Nếu v là đỉnh cô lập thì lấy v khỏi ngăn xếp và đưa vào S;
* Nếu v là liên thông với đỉnh x thì xếp x vào ngăn xếp T sau đó xoá bỏ cạnh (v, x);

1. Quay lại bước 2 cho tới khi ngăn xếp rỗng. Kết quả chu trình Euler được chứa trong S theo thứ tự ngược lại.

* Thuật toán tìm đường đi euler
* Xác định điểm xuất phát của đường đi từ đỉnh bậc lẻ này và kết thúc ở đỉnh bậc lẻ khác.
* Nạp đỉnh bậc lẻ vừa xác định vào stack, sau đó thực hiện như thuật toán tìm chu trình euler.

### 2.3 Phương pháp giải bài toán

Với chuỗi ban đầu n+2 ký tự:



Tách thành:



Nếu coi mỗi chuỗi là một đỉnh, và mỗi 2 đỉnh kề nhau trên danh sách trên là một cạnh, thì bài toán được đưa về bài toán tìm đường đi Hamilton qua tất cả các đỉnh. => khó

Nếu ta coi mỗi đỉnh là một cạnh:



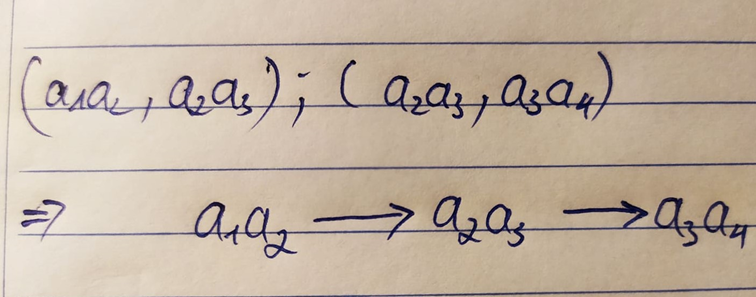
Tìm đường đi Euler từ đỉnh :  đến 

Lưu lại thứ tự các cạnh trên đường đi euler, từ đó chuỗi ban đầu sẽ là đỉnh đầu + ký tự thứ 3 của các đỉnh tiếp theo trong thứ tự đã tìm được.

* 1. Chứng minh tính đúng đắn:

Ta dễ dàng chứng minh tính đúng đắn của thuật toán nhờ quy nạp toán học:

Với n = 4 ta có:



Giả sử bài toán đúng với n=k, ta sẽ chứng minh nó cũng đúng với n=k+1

Thật vây, với n=k+1 ta có thêm cạnh 

Từ đường đi tại n=k vừa tìm được, ta thêm cạnh  vào đường đi, ta được lời giải bài toán với n=k+1.

* 1. Đánh giá thời gian tính:

Chương trình gồm 4 phần:

* Hàm void f() biểu diễn các đỉnh(chuỗi 3 ký tự) theo cơ số 62, thời gian tính O(3).

int f(string s) {

int ret = 0;

for(int i=0;i<3;++i){

ret \*= 62;

if(s[i] < 'A') {

ret += s[i] -'0';

} else if(s[i]<'a') {

ret += s[i]-'A'+10;

} else {

ret += s[i]-'a'+36;

}

}

return ret;

}

* Hàm dfs() để tìm đường đi euler từ đỉnh source thời gian tính O(N)

void dfs(int u) {

while(!adj[u].empty()){

int v = adj[u].back().first;

int e = adj[u].back().second;

adj[u].pop\_back();

T.push\_back(e);

vis[e] = 1;

dfs(v);

while(!T.empty()){

S.push\_back(T.back());

T.pop\_back();

if(S.back() == e) break;

}

}

}

* Vòng for khởi tạo và lưu trữ các cạnh thời gian tính O(N)

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

cin >> edge[i];

int p = f(edge[i]);

int a = p/62;

int b = p%(62\*62);

adj[a].push\_back(make\_pair(b, i));

indeg[b]++;

}

* Vòng for tìm source và sink thời gian tính O(62\*62)

source = -1; sink = -1;

bool foundsource = false;

bool foundsink = false;

for(int i=0;i<62\*62;++i){

int outdeg = adj[i].size();

if(outdeg > 0 && !foundsource){

source = i;

}

if(outdeg - indeg[i] == 1) {

if(foundsource) {

printf("NO\n"); return 0;

}

source = i;

foundsource = true;

}

else if(indeg[i] - outdeg == 1) {

if(foundsink) {

printf("NO\n"); return 0;

}

sink = i;

foundsink = true;

}

else if(indeg[i] - outdeg != 0) {

printf("NO\n"); return 0;

}

}

* In kết quả thời gian tính O(N)

while(!S.empty()){

if(!flag){

printf("%s", edge[S.back()].c\_str());

flag = true;

} else {

printf("%c", edge[S.back()][2]);

}

S.pop\_back();

}

* **=> thời gian tính của chương trình:** O(max(O(3)+O(N)+O(N)+O(N)+O(N))=O(N)