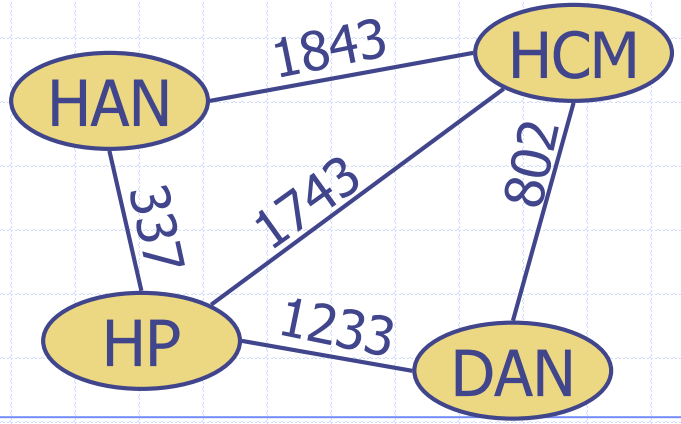
CHƯƠNG 7

Đồ thị và các thuật toán đồ thị



NỘI DUNG

7.1. Đồ thị

Đồ thị vô hướng, Đồ thị có hướng,Tính liên thông của đồ thị

7.2. Biểu diễn đồ thị

Biểu diễn đồ thị bởi ma trận, Danh sách kề, Danh sách cạnh

7.3. Các thuật toán duyệt đồ thị

Thuật toán tìm kiếm theo chiều sâu, Thuật toán tìm kiếm theo chiều rộng

7.4. Một số ứng dụng của tìm kiếm trên đồ thị

Bài toán đường đi, Bài toán liên thông,

Đồ thị không chứa chu trình và bài toán sắp xếp tôpô, Bài toán tô màu đỉnh đồ thị

7.5. Bài toán cây khung nhỏ nhất

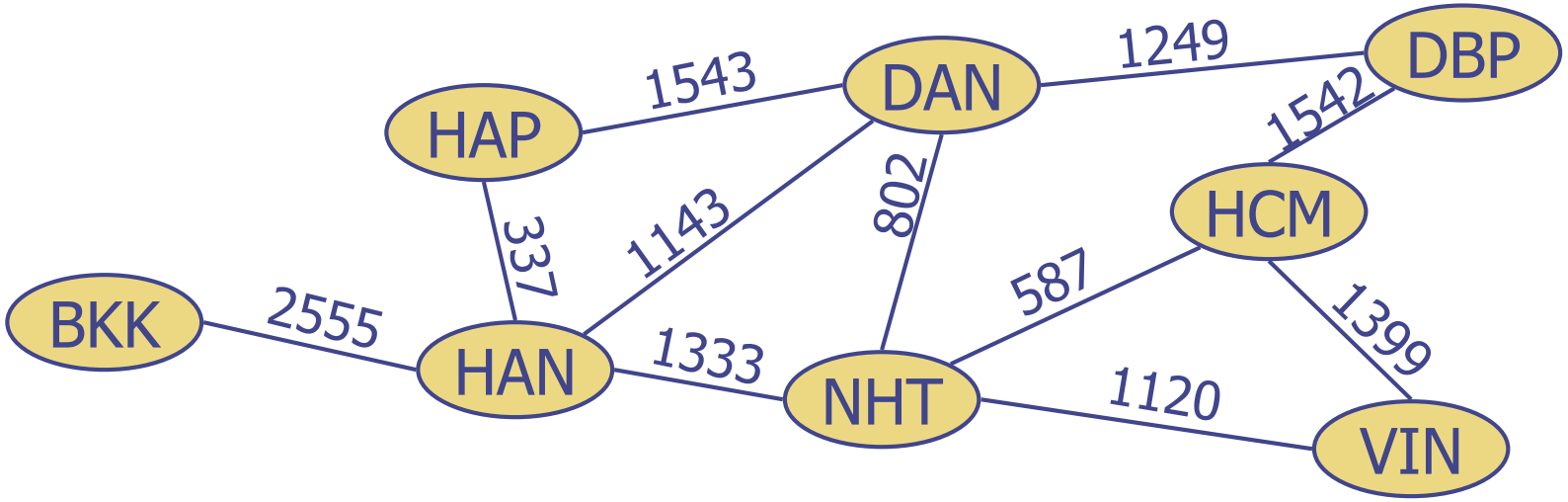
Thuật toán Kruscal, Cấu trúc dữ liệu biểu diễn phân hoạch,

7.6. Bài toán đường đi ngắn nhất

Thuật toán Dijkstra, Cài đặt thuật toán với các cấu trúc dữ liệu

7.1. Đồ thị

- Đồ thị là cặp (***V, E***), trong đó
  - ***V*** là tập đỉnh
  - ***E*** là họ các cặp đỉnh gọi là các cạnh
- Ví dụ:
  - Các đỉnh là các sân bay
  - Các cạnh thể hiện đường bay nối hai sân bay
  - Các số trên cạnh có thể là chi phí (thời gian, khoảng cách)

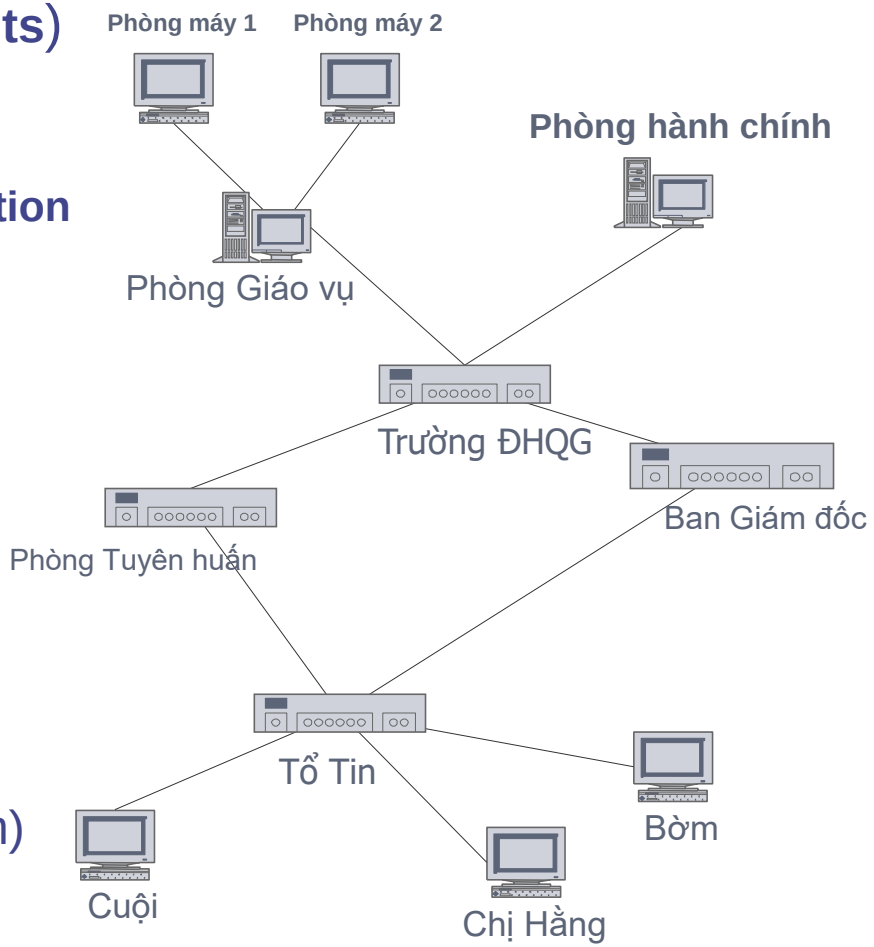


Các kiểu cạnh

- Cạnh có hướng (Directed edge)
  - Cặp có thứ tự gồm hai đỉnh (u,v)
  - Đỉnh u là đỉnh đầu
  - Đỉnh v là đỉnh cuối
  - Ví dụ, đường bay
- Cạnh vô hướng (Undirected edge)
  - Cặp không có thứ tự gồm 2 đỉnh (u,v)
  - Ví dụ, tuyến bay
- Đồ thị có hướng (digraph)
  - Các cạnh có hướng
  - Ví dụ, mạng truyền tin
- Đồ thị vô hướng (Undirected graph)
  - Các cạnh không có hướng
  - Ví dụ, mạng tuyến bay

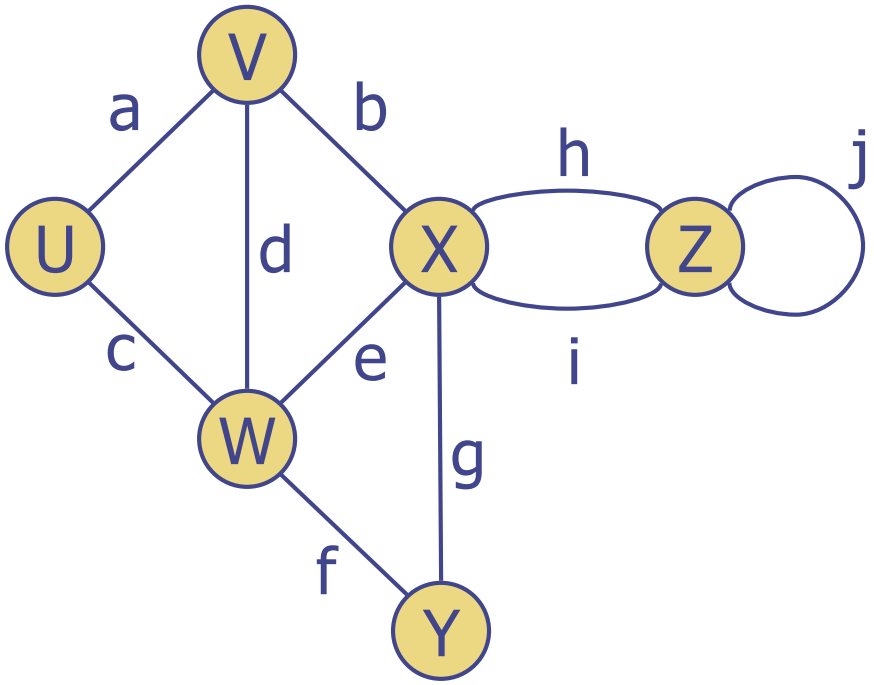
Ứng dụng

- Mạch lôgic (**Electronic circuits**)
  - Mạch in
  - Mạch tích hợp
- Mạng giao thông (**Transportation networks**)
  - Mạng xa lộ
  - Mạng tuyến bay
- Mạng máy tính (**Computer networks**)
  - Mạng cục bộ
  - Internet
  - Web
- Cơ sở dữ liệu (Databases)
  - Sơ đồ quan hệ thực thể (Entity-relationship diagram)



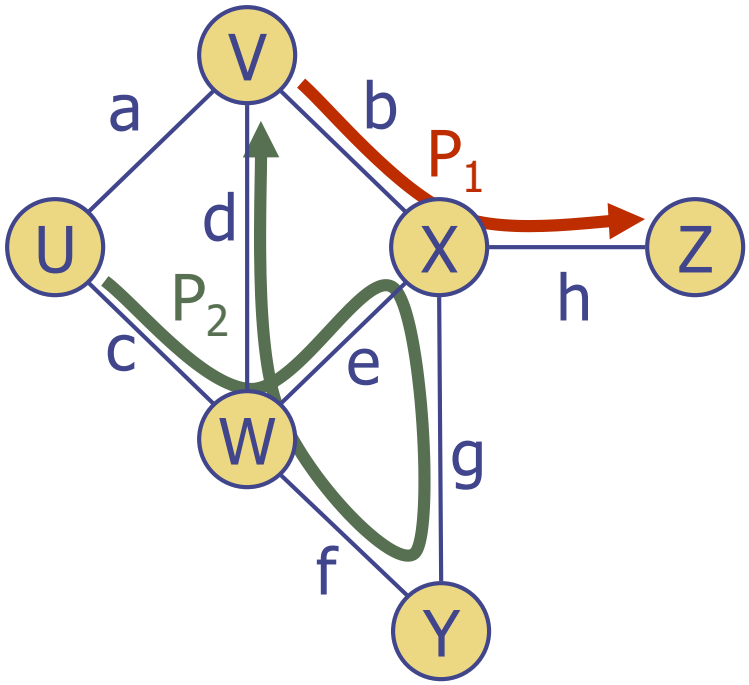
Thuật ngữ

- Đầu mút của cạnh
  - U và V là các đầu mút của cạnh a
- Cạnh kề với đỉnh
  - a, d, và b kề với đỉnh V
- Đỉnh kề
  - U và V là kề nhau
- Bậc của đỉnh
  - X có bậc 5
- Cạnh lặp
  - h và i là các cạnh lặp
- Khuyên
  - j là khuyên



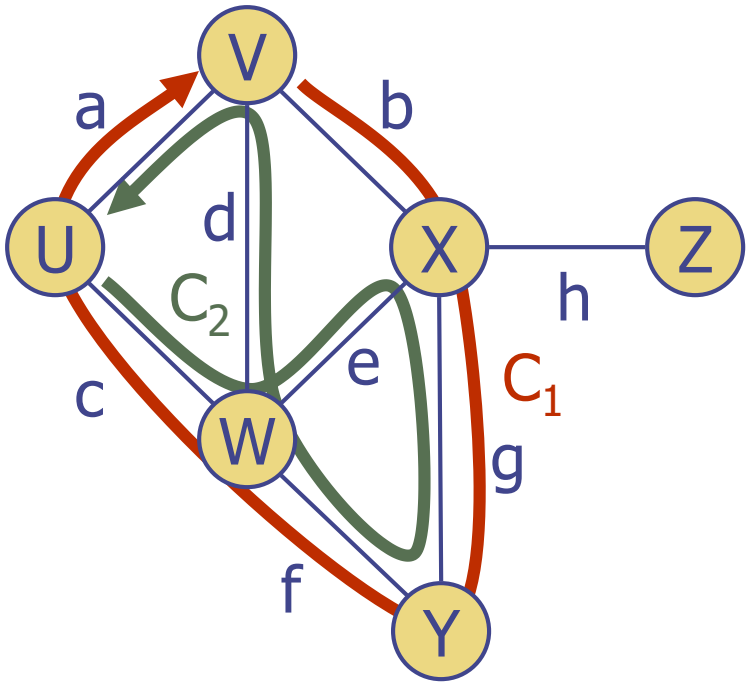
Thuật ngữ (tiếp tục)

- Đường đi
  - Dãy các đỉnh (hoặc dãy các cạnh)
  - Hai đỉnh liên tiếp là có cạnh nối
  - Đỉnh đầu và đỉnh cuối
- Đường đi đơn
  - Các đỉnh trên đường đi là phân biệt
- Ví dụ
  - P_1= V,X,Z (dãy cạnh: b, h) là đường đi đơn
  - P_2= U,W,X,Y,W,V) (P_2=c,e,g,f,d) là đường đi không là đơn



Thuật ngữ (tiếp)

- Chu trình
  - Đường đi gồm các cạnh phân biệt có đỉnh đầu trùng đỉnh cuối
- Chu trình đơn
  - Ngoại trừ đầu trùng cuối, không còn hai đỉnh nào giống nhau
- Ví dụ
  - C_1= V,X,Y,W,U là chu trình đơn
  - C_2=U,W,X,Y,W,V là chu trinh không là đơn



Tính chất

Tính chất 1

$\sum_v \deg(v) = 2m$

CM: mỗi cạnh được đếm 2 lần

Tính chất 2

Trong đơn đồ thị vô hướng (đồ thị không có cạnh lặp và khuyên)

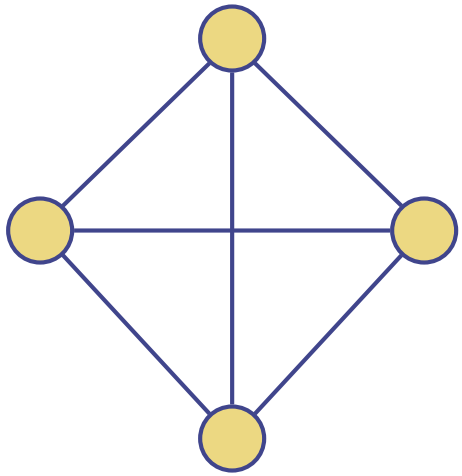
$m \le n\,(n-1)/2$

CM: mỗi đỉnh có bậc không quá $(n-1)$

Tương tự có những cận cho đồ thị có hướng

Ký hiệu

n	số đỉnh
m	số cạnh
$\deg(v)$	bậc của đỉnh v



Ví dụ

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Graph ADT

- Các phép toán cơ bản (Basic Graph operations)
  - khởi tạo/create (số đỉnh, isDirected)
  - huỷ/destroy
  - nhận số cạnh / get number of edges
  - nhận số đỉnh / get number of vertices
  - cho biết đồ thị là có hướng hay vô hướng / tell whether graph is directed or undirected
  - bổ sung cạnh / insert an edge
  - loại bỏ cạnh / remove an edge
  - có cạnh nối giữa hai đỉnh / tell whether an edge exists between two vertices
  - duyệt các đỉnh kề của một đỉnh cho trước / An iterator that process all vertices adjacent to a given vertex

Các bài toán xử lý đồ thị

- Tính giá trị của một số đặc trưng số của đồ thị (số liên thông, sắc số, ...)
- Tìm một số tập con cạnh đặc biệt (chẳng hạn, cặp ghép, bè, chu trình, cây khung, ...)
- Tìm một số tập con đỉnh đặc biệt (chẳng hạn, phủ đỉnh, phủ cạnh, tập độc lập,...)
- Trả lời truy vấn về một số tính chất của đồ thị (liên thông, phẳng, ...)
- Các bài toán tối ưu trên đồ thị: Cây khung nhỏ nhất, đường đi ngắn nhất, luồng cực đại trong mạng, ...
- ...

NỘI DUNG

7.1. Đồ thị

7.2. Biểu diễn đồ thị

7.3. Các thuật toán duyệt đồ thị

7.4. Một số ứng dụng của tìm kiếm trên đồ thị

7.5. Bài toán cây khung nhỏ nhất

7.6. Bài toán đường đi ngắn nhất

7.2. Biểu diễn đồ thị

- Biểu diễn đồ thị bởi ma trận,
- Danh sách kề,
- Danh sách cạnh

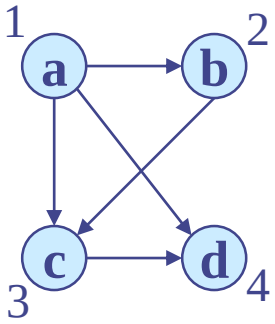
Biểu diễn đồ thị

- Có nhiều cách biểu diễn,
- Việc lựa chọn cách biểu diễn phụ thuộc vào từng bài toán cụ thể cần xét, từng thuật toán cụ thể cần cài đặt.
- Có hai vấn đề chính cần quan tâm khi lựa chọn cách biểu diễn:
  - Bộ nhớ mà cách biểu diễn đó đòi hỏi
  - Thời gian cần thiết để trả lời các truy vấn thường xuyên đối với đồ thị trong quá trình xử lý đồ thị:
    - Chẳng hạn:
      - Có cạnh nối hai đỉnh u, v ?
      - Liệt kê các đỉnh kề của đỉnh v ?

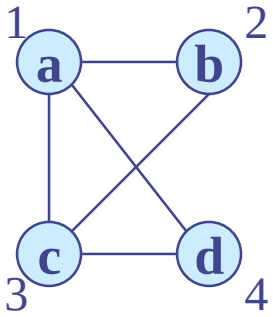
Ma trận kề (Adjacency Matrix)

- $n \times n$ ma trận A.
- Các đỉnh được đánh số từ 1 đến $|V|$ theo 1 thứ tự nào đó.
- A xác định bởi:

$$A[i, j] = a_{ij} = \begin{cases} 1 & \text{nêu } (i, j) \in E \\ 0 & \text{nêu trái lai} \end{cases}$$



	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

$A = A^T$ đối với đồ thị vô hướng.

Ma trận kề

- Chú ý về sử dụng ma trận kề:
  - Dòng toàn không ~đỉnh cô lập.
  - M[i, i] = 1 $\leftrightarrow$ khuyên (self-loop)
- Bộ nhớ (Space)
  - $|V|^2$ bits
  - $(|V|^2 + |V|)/2$ (nếu là đồ thị vô hướng nhưng khó cài đặt).
  - Các thông tin bổ sung, chẳng hạn chi phí trên cạnh, cần được cất giữ dưới dạng ma trận.
- Thời gian trả lời các truy vấn
  - Hai đỉnh *i* và *j* có kề nhau? O(1)
  - Bổ sung hoặc loại bỏ cạnh O(1)
  - Bổ sung đỉnh: tăng kích thước ma trận
  - Liệt kê các đỉnh kề của *v* : O(|*V*|) (ngay cả khi *v* là đỉnh cô lập).

Ma trận trọng số

- Trong trường hợp đồ thị có trọng số trên cạnh, thay vì ma trận kề, để biểu diễn đồ thị ta sử dụng ma trận trọng số

$$C = c[i, j],\ i, j = 1, 2, ..., n,$$

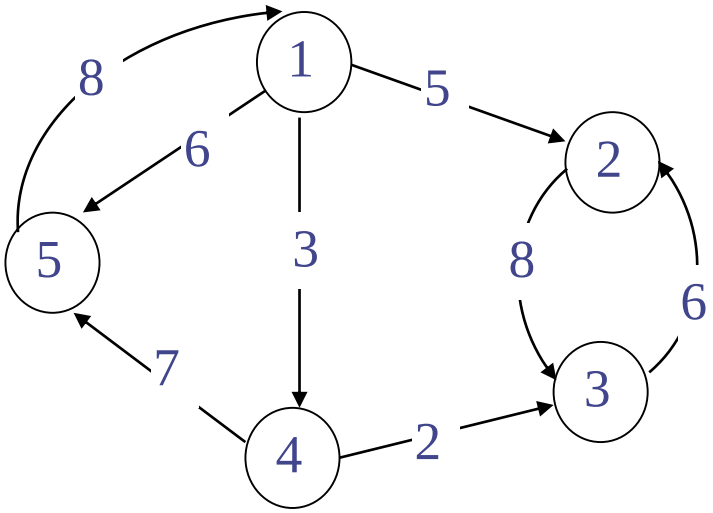
với

$$c[i, j] = \begin{cases} c(i, j), & \text{if } (i, j) \in E \\ \theta, & \text{if } (i, j) \notin E, \end{cases}$$

trong đó θ là giá trị đặc biệt để chỉ ra một cặp (i,j) không là cạnh, tùy từng trường hợp cụ thể, có thể được đặt bằng một trong các giá trị sau: 0, $+\infty$, $-\infty$.

Ma trận trọng số

- Ví dụ

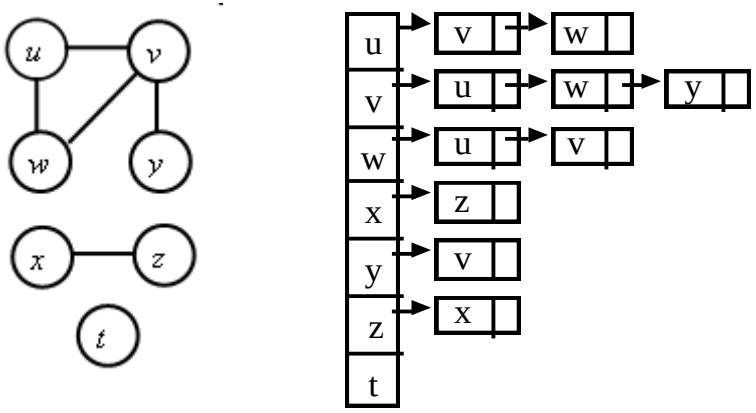


$$A = \begin{bmatrix} \theta & 5 & \theta & 3 & 6 \\ \theta & \theta & 8 & \theta & \theta \\ \theta & 6 & \theta & \theta & \theta \\ \theta & \theta & 2 & \theta & 7 \\ 8 & \theta & \theta & \theta & \theta \end{bmatrix}$$

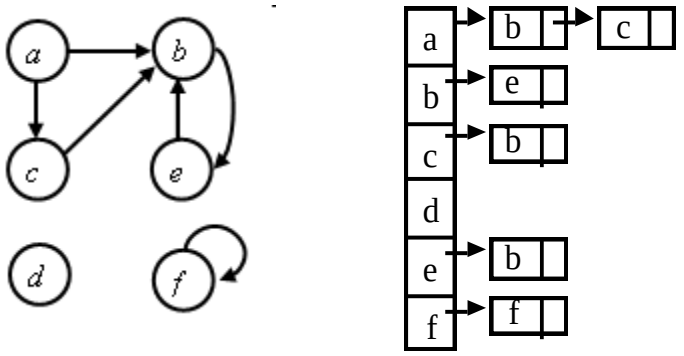
Danh sách kề (Adjacency List)

- **Danh sách kề:** Với mỗi đỉnh *v* cất giữ danh sách các đỉnh kề của nó.
  - Là mảng *Adj* gồm |*V*| danh sách.
  - Mỗi đỉnh có một danh sách.
  - Với mỗi $u \in V$, *Adj*[*u*] bao gồm tất cả các đỉnh kề của *u*.
- Ví dụ

Đồ thị vô hướng

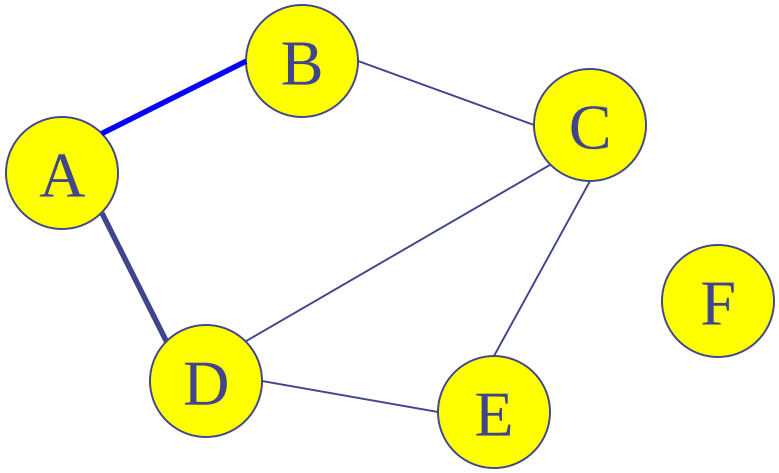


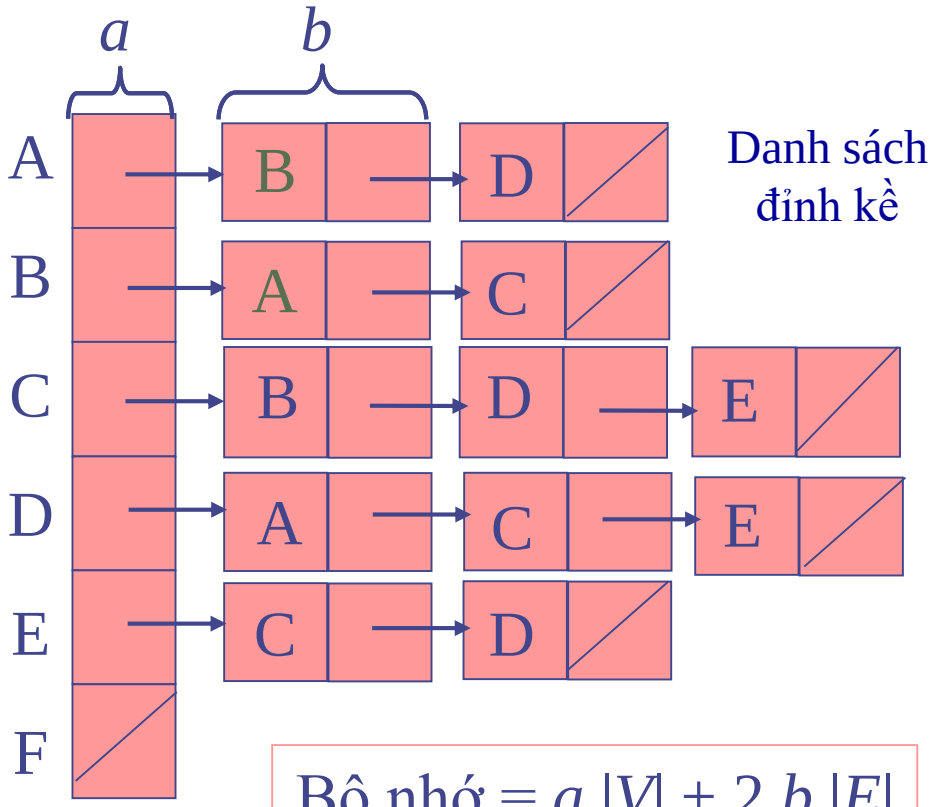
Đồ thị có hướng



Danh sách kề của đồ thị vô hướng

Với mỗi $v \in V$, Ke(v) = danh sách các đỉnh u: $(v, u) \in E$

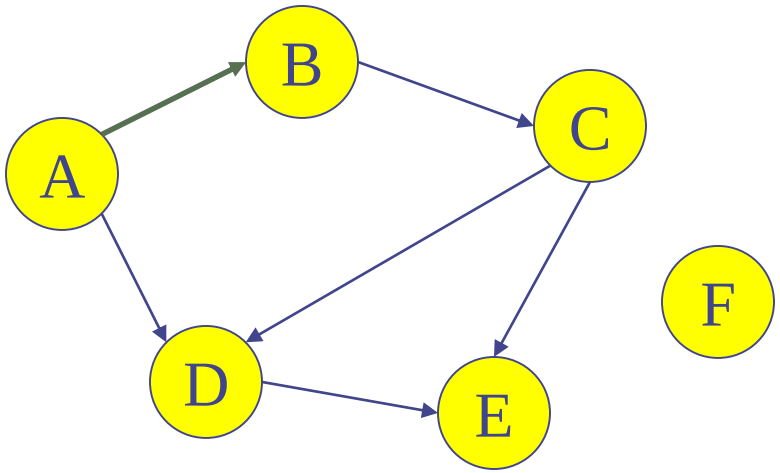


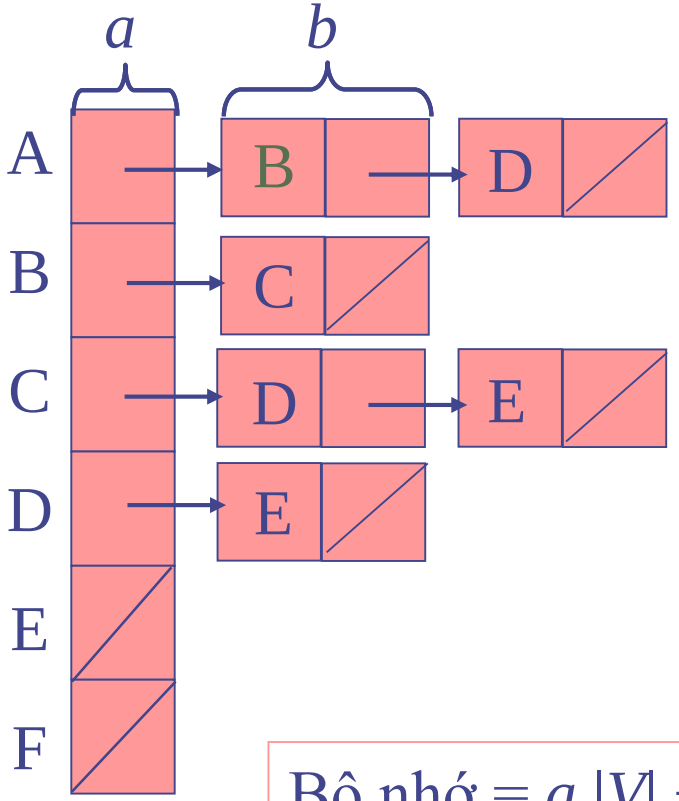


Bộ nhớ = $a\,|V| + 2\,b\,|E|$

Danh sách kề của đồ thị có hướng

Với mỗi $v \in V$, $\text{Ke}(v) = \{ u: (v, u) \in E \}$





Bộ nhớ = $a\,|V| + b\,|E|$

Yêu cầu bộ nhớ

- Tổng cộng bộ nhớ: $\Theta(|V|+|E|)$
- Thường là nhỏ hơn nhiều so với $|V|^2$, nhất là đối với đồ thị thưa (sparse graph).
- Đồ thị thưa là đồ thị mà $|E| = k\,|V|$ với $k < 10$.
- ***Chú ý:***
  - ***Phần lớn các đồ thị trong thực tế ứng dụng là đồ thị thưa!***
  - ***Cách biểu diễn này được sử dụng nhiều nhất trong ứng dụng***

Ngã 6 Bắc Ninh

Biểu diễn đồ thị bởi danh sách kề

- Thời gian trả lời các truy vấn:
  - Thêm cạnh $O(1)$
  - Xoá cạnh Duyệt qua danh sách kề của mỗi đầu mút.
  - Thêm đỉnh Phụ thuộc vào cài đặt.
  - Liệt kê các đỉnh kề của v: O(<số đỉnh kề>) (tốt hơn ma trận kề)
  - Hai đỉnh i, j có kề nhau? Tìm kiếm trên danh sách: Θ(degree(i)). Đánh giá trong tình huống tồi nhất là $O(|V|)$ => không hiệu quả (tồi hơn ma trận kề)

Cấu trúc dữ liệu

```
#define N ...; // số lượng đỉnh tối đa
struct NodeType{
            int v;  // đỉnh kề
            NodeType *Next;
        };
typedef struct NodeType AdjList;
AdjList *Ke[N];  // danh sách kề của đồ thị
```

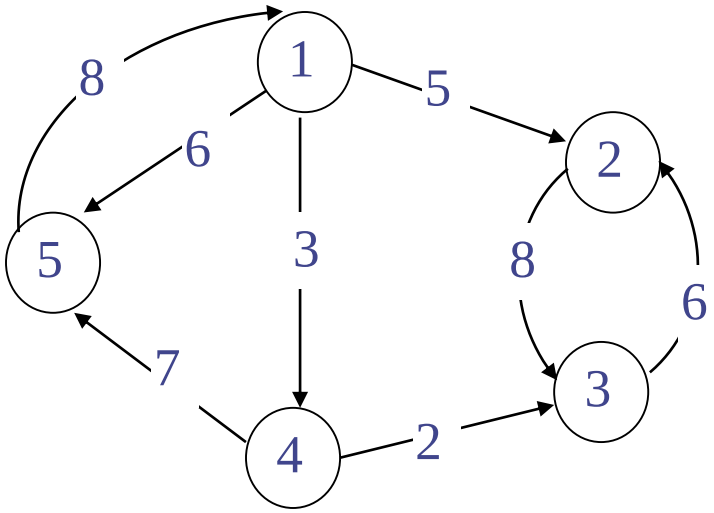
Danh sách cạnh (Edge List)

- Với mỗi cạnh e = (u, v) cất giữ

  dau[e]= u , cuoi[e] = v

- Nếu đồ thị có trọng số trên cạnh, thì cần có thêm một biến cất giữ c[e]
- **Đây là cách chuẩn bị dữ liệu cho các đồ thị thực tế**

Danh sách cạnh



e	dau[e]	cuoi[e]	c[e]
1	1	5	6
2	5	1	8
3	4	5	7
4	1	4	3
5	1	2	5
6	4	3	2
7	2	3	8
8	3	2	6

Đánh giá thời gian thực hiện các thao tác

◆ n đỉnh, m cạnh ◆ đơn đồ thị vô hướng	Edge List	Adjacency List	Adjacency Matrix
Bộ nhớ	$n + m$	$n + m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

NỘI DUNG

7.1. Đồ thị

7.2. Biểu diễn đồ thị

7.3. Các thuật toán duyệt đồ thị

7.4. Một số ứng dụng của tìm kiếm trên đồ thị

7.5. Bài toán cây khung nhỏ nhất

7.6. Bài toán đường đi ngắn nhất

7.3. Các thuật toán duyệt đồ thị (Graph Searching)

- Tìm kiếm theo chiều rộng
- Tìm kiếm theo chiều sâu

Duyệt đồ thị

- Ta gọi ***duyệt đồ thị*** (Graph Searching hoặc Graph Traversal) là việc duyệt qua mỗi đỉnh và mỗi cạnh của đồ thị.
- Ứng dụng:
  - *Xây dựng các thuật toán khảo sát các tính chất của đồ thị;*
  - *Là thành phần cơ bản của nhiều thuật toán.*
- Cần xây dựng thuật toán hiệu quả để thực hiện việc duyệt đồ thị. Ta xét hai thuật toán duyệt cơ bản:
  - *Tìm kiếm theo chiều rộng* (*Breadth First Search – BFS*)
  - *Tìm kiếm theo chiều sâu* (*Depth First Search – DFS*)

Ý tưởng chung

- Trong quá trình thực hiện thuật toán, mỗi đỉnh ở một trong ba trạng thái:
  - *Chưa thăm (thể hiện bởi màu trắng),*
  - *Đã thăm nhưng chưa duyệt xong (thể hiện bởi màu xám)*
  - *Đã duyệt xong (thể hiện bởi màu đen).*
- Quá trình duyệt được bắt đầu từ một đỉnh *v* nào đó. Ta sẽ khảo sát các đỉnh đạt tới được từ *v*:
  - *Thoạt đầu mỗi đỉnh đều có màu trắng (chưa thăm - not visited).*
  - *Đỉnh đã được thăm sẽ chuyển thành màu xám (trở thành đã thăm nhưng chưa duyệt xong).*
  - *Khi tất cả các đỉnh kề của một đỉnh v là đã được thăm, đỉnh v sẽ có màu đen (đã duyệt xong).*

Thuật toán tìm kiếm theo chiều rộng

(BFS algorithm)

BFS

- **Input:** Đồ thị $G = (V, E)$, có hướng hoặc vô hướng, và đỉnh xuất phát $s \in V$.
- **Output:**

  Với mọi $v \in V$
  - $d[v]$ = khoảng cách từ s đến v.
  - $\pi[v]$ – đỉnh đi trước v trong đường đi ngắn nhất từ s đến v.
  - Xây dựng cây BFS gốc tại s chứa tất cả các đỉnh đạt đến được từ s.
- Ta sẽ sử dụng màu để ghi nhận trạng thái của đỉnh trong quá trình duyệt:
  - *Trắng* (*White*) – chưa thăm.
  - *Xám* (*Gray*) – đã thăm nhưng chưa duyệt xong.
  - *Đen* (*Black*) – đã duyệt xong

Tìm kiếm theo chiều rộng từ đỉnh s

BFS_Visit(s)

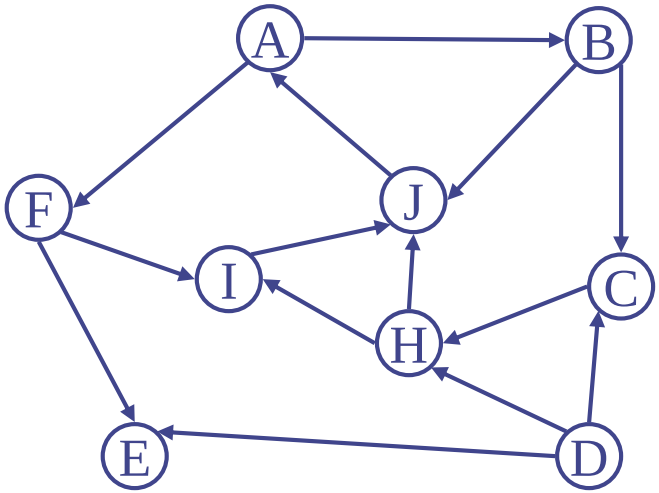
```
1.  for  u ∈ V – {s}
2          do color[u] ← white
3             d[u] ← ∝
4             π[u] ← NULL
5   color[s] ← gray
6   d[s] ← 0
7   π[s] ← NULL
8   Q ← ∅
9   enqueue(Q,s)
10  while Q ≠ ∅
11     do u ← dequeue(Q)
12        for  v ∈ Adj[u]
13           do if color[v] = white
14              then  color[v] ← gray
15                    d[v] ← d[u] + 1
16                    π[v] ← u
17                    enqueue(Q,v)
18        color[u] ← black
```

Thuật toán tìm kiếm theo chiều rộng trên đồ thị *G*

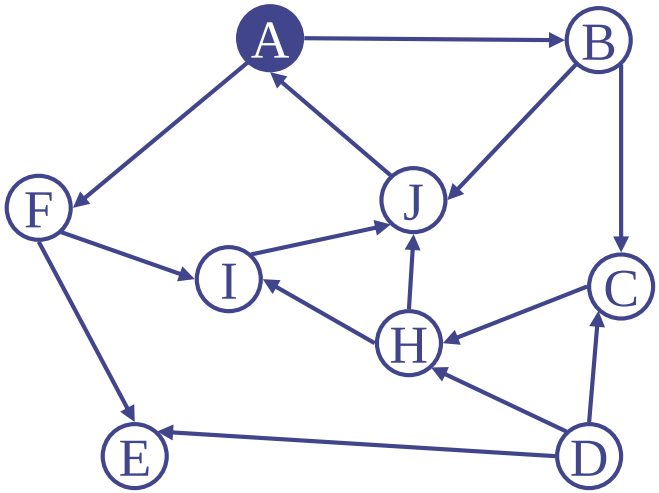
BFS(G)

1. **for** $u \in V$
2. **do** $color[u] \leftarrow$ white
3. $\pi[u] \leftarrow$ NULL
5. **for** $u \in V$
6. **do if** $color[u]$ = white
7. **then BFS-Visit(*u*)**

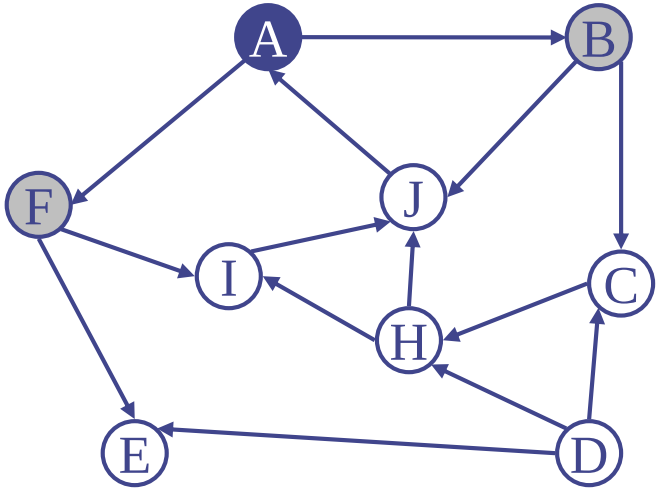
Ví dụ: Thực hiện BFS(A)



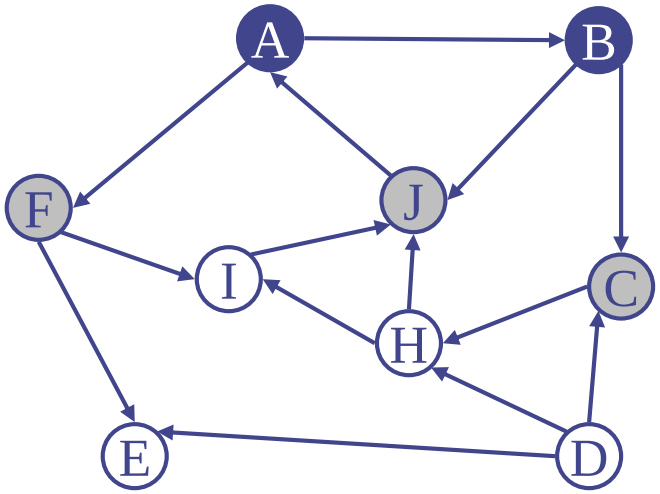
Q = {A}



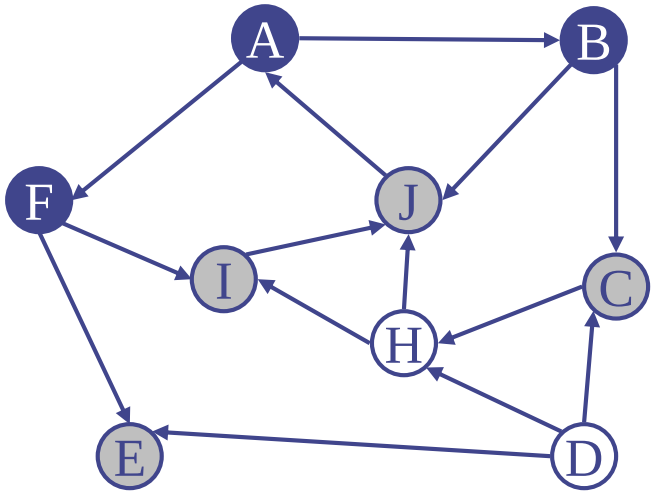
Q = {B,F}



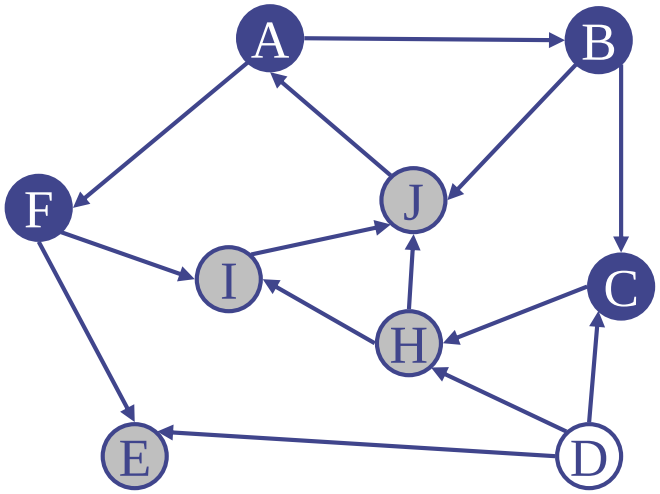
Q = {F,C,J}



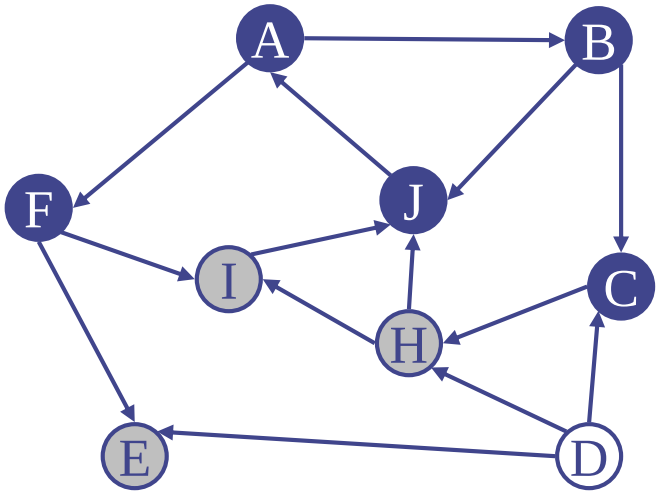
Q = {C,J,E,I}



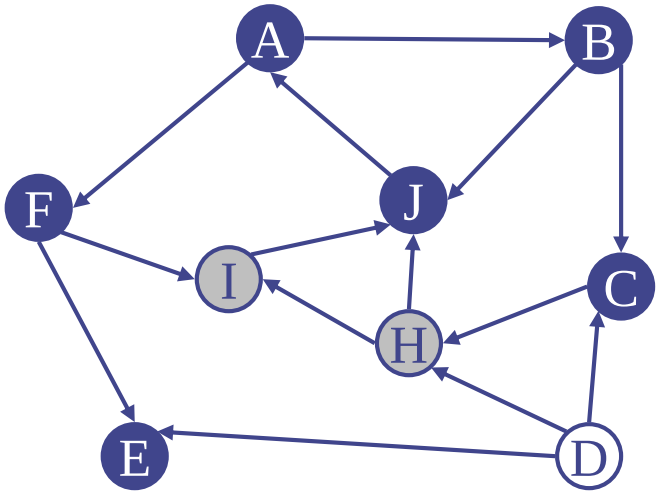
Q = {J,E,I,H}



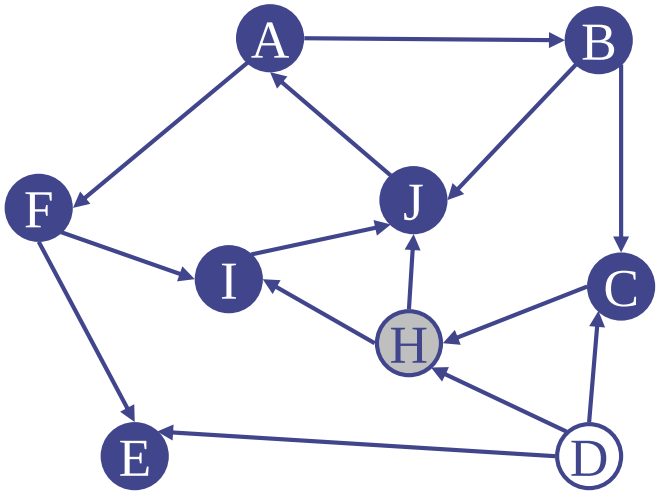
Q = {E,I,H}



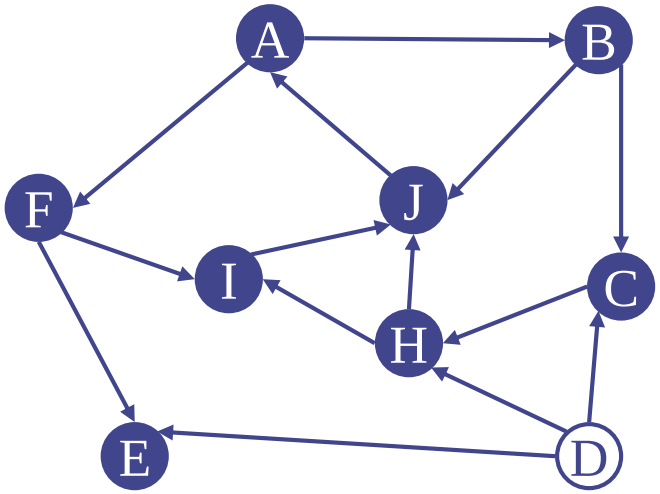
Q = {I,H}



Q = {H}



Q = {} Kết thúc BFS(A)



Tính đúng đắn của BFS

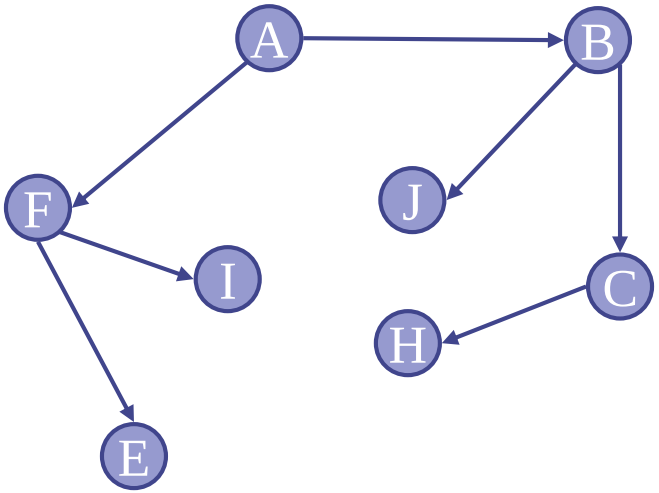
Định lý:

- BFS cho phép đến thăm tất cả các đỉnh $v \in V$ đạt đến được từ s.
- Khi thuật toán kết thúc $d[v]$ cho ta độ dài đường đi ngắn nhất (theo số cạnh) từ s đến v.
- Với mỗi đỉnh v đạt đến được từ s, $\pi[v]$ cho ta đỉnh đi trước đỉnh v trong đường đi ngắn nhất từ s đến v.

Cây tìm kiếm theo chiều rộng (Breadth-first Tree)

- Đối với đồ thị $G = (V, E)$ với đỉnh xuất phát s, ký hiệu $G_\pi = (V_\pi, E_\pi)$ là đồ thị với
  - $V_\pi = \{v \in V : \pi[v] \neq \text{NULL}\} \cup \{s\}$
  - $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$
- Đồ thị G_π được gọi là cây BFS(s):
  - V_π chứa tất cả các đỉnh đạt đến được từ s và
  - với mọi $v \in V_\pi$, đường đi từ s đến v trên G_π là đường đi ngắn nhất từ s đến v trên G.
- Các cạnh trong E_π được gọi là các ***cạnh của cây***.
  $|E_\pi| = |V_\pi| - 1$.

Ví dụ: Cây BFS(A)



Độ phức tạp của BFS

- Thuật toán loại bỏ mỗi đỉnh khỏi hàng đợi đúng 1 lần, do đó thao tác DeQueue thực hiện đúng $|V|$ lần.
- Với mỗi đỉnh, thuật toán duyệt qua tất cả các đỉnh kề của nó và thời gian xử lý mỗi đỉnh kề như vậy là hằng số. Như vậy thời gian thực hiện câu lệnh **if** trong vòng lặp **while** là bằng hằng số nhân với số cạnh kề với đỉnh đang xét.
- Do đó tổng thời gian thực hiện việc duyệt qua tất cả các đỉnh là bằng một hằng số nhân với số cạnh $|E|$.
- **Thời gian tổng cộng:** $O(|V|) + O(|E|) = O(|V|+|E|)$, hay $O(|V|^2)$

Thuật toán tìm kiếm theo chiều sâu (DFS)

Tìm kiếm theo chiều sâu

- ***Input:*** $G = (V, E)$ - đồ thị vô hướng hoặc có hướng.
- ***Output:*** **V**ới mỗi $v \in V$.
  - $d[v]$ = ***thời điểm bắt đầu thăm*** (v chuyển từ màu trắng sang xám)
  - $f[v]$ = ***thời điểm kết thúc thăm*** (v chuyển từ màu xám sang đen)
  - $\pi[v]$: đỉnh từ đó ta đến thăm đỉnh v.
- ***Rừng tìm kiếm theo chiều sâu*** **(**gọi tắt là rừng DFS - Forest of *depth-first trees*):
  - $G_\pi = (V, E_\pi)$,
  - $E_\pi = \{(\pi[v], v): v \in V \text{ và } \pi[v] \neq \textbf{null}\}$.

Thuật toán tìm kiếm theo chiều sâu bắt đầu từ đỉnh *u*

DFS-Visit(*u*)

color[*u*] ← GRAY

time ← *time* + 1

d[*u*] ← *time*

for *v* ∈ *Adj*[*u*]

do if *color*[*v*] = WHITE

then π[*v*] ← *u*

DFS-Visit(*v*)

color[*u*] ← BLACK

f[*u*] ← *time* ← *time* + 1

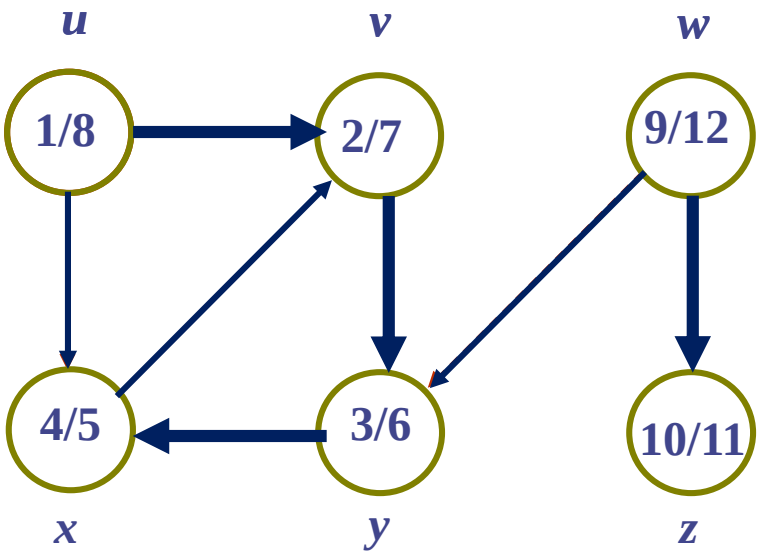
Thuật toán tìm kiếm theo chiều sâu trên đồ thị *G*

DFS(G)

1. **for** $u \in V[G]$
2. **do** $color[u] \leftarrow$ white
3. $\pi[u] \leftarrow$ NULL
4. $time \leftarrow 0$
5. **for** $u \in V[G]$
6. **do if** $color[u]$ = white
7. **then DFS-Visit(*u*)**

Ví dụ

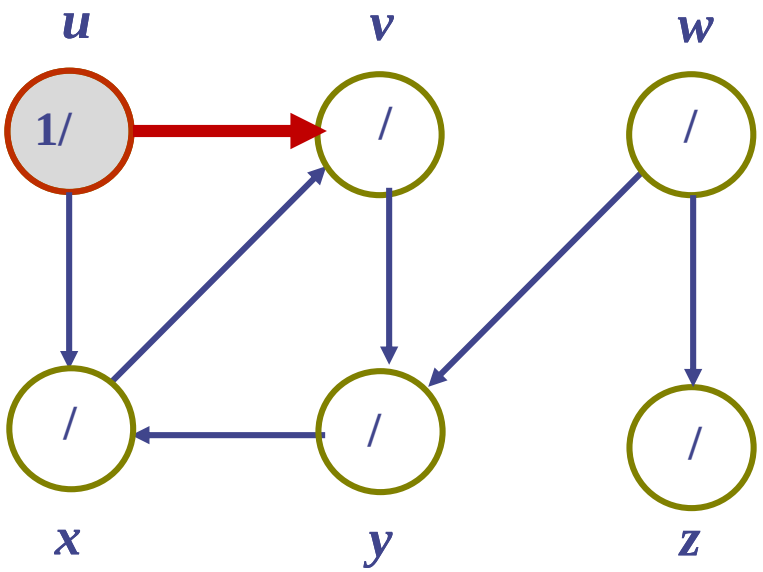
- Thực hiện DFS trên đồ thị sau:



- DFS(G) sẽ gọi thực hiện DFS(u) và DFS(w).
- Cặp số viết trong các đỉnh v là d[v]/f[v].
- Các cạnh đậm là các cạnh của rừng tìm kiếm.

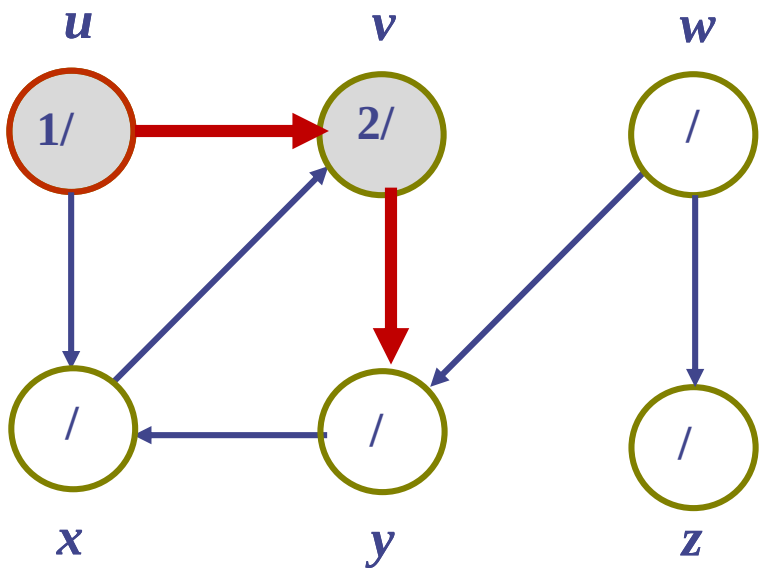
DFS(u)

- Thăm đỉnh u



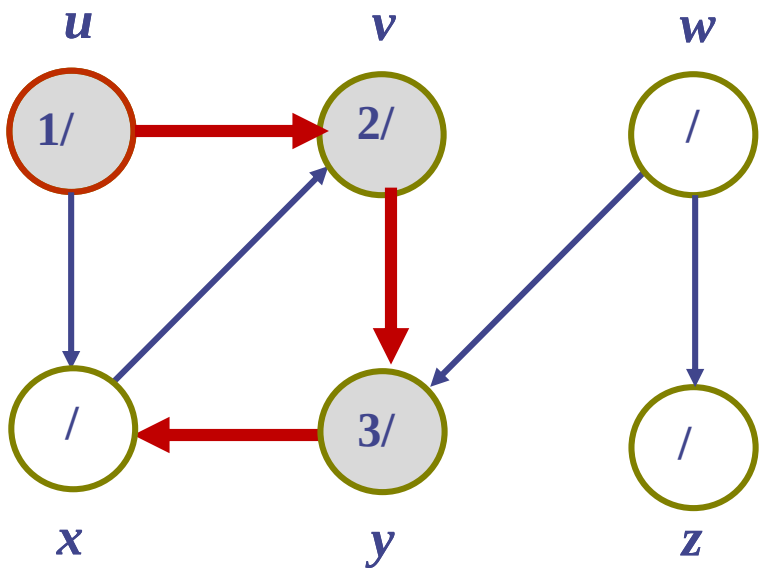
DFS(v)

- Thăm đỉnh v



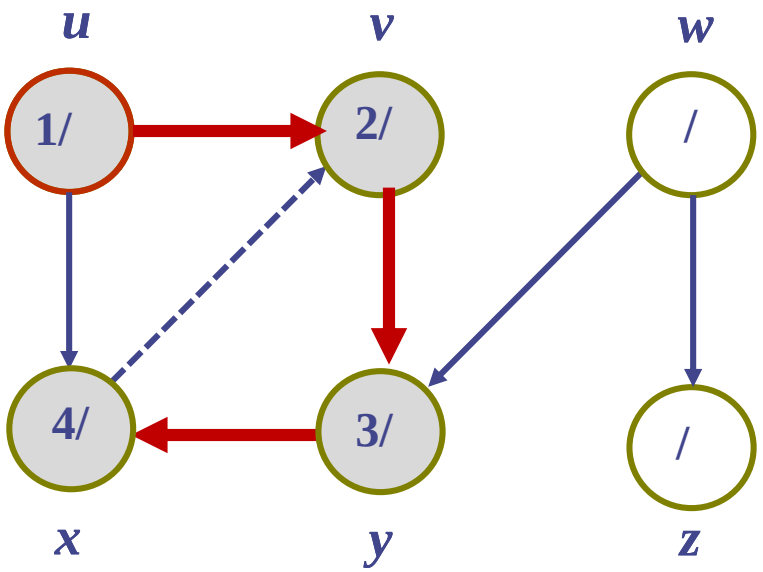
DFS(y)

- Thăm đỉnh y



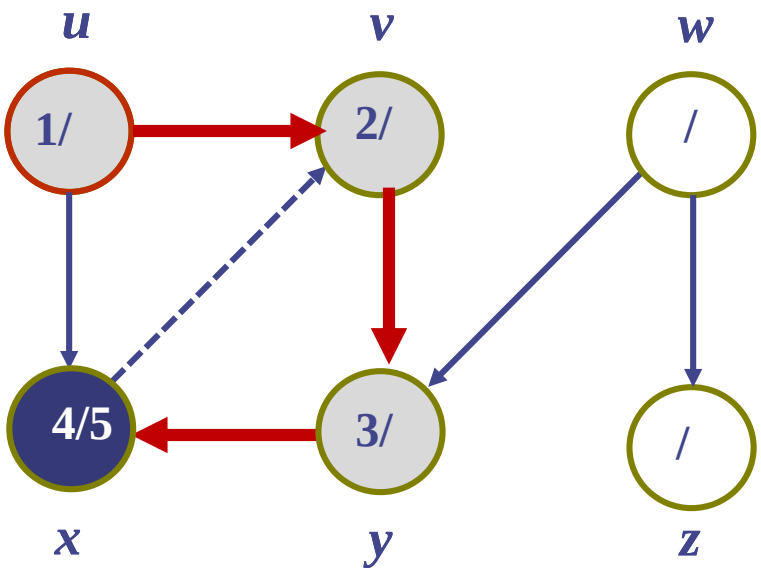
DFS(x)

- Thăm đỉnh x



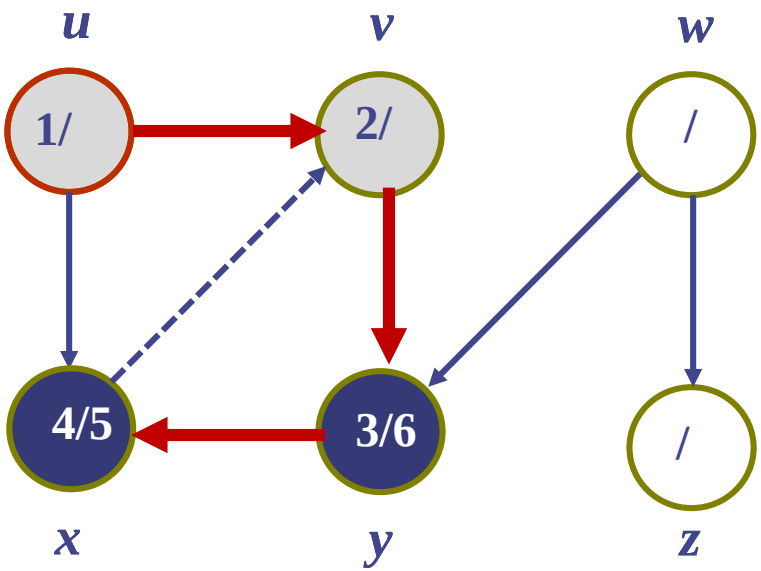
DFS(x)

- Kết thúc thăm đỉnh x



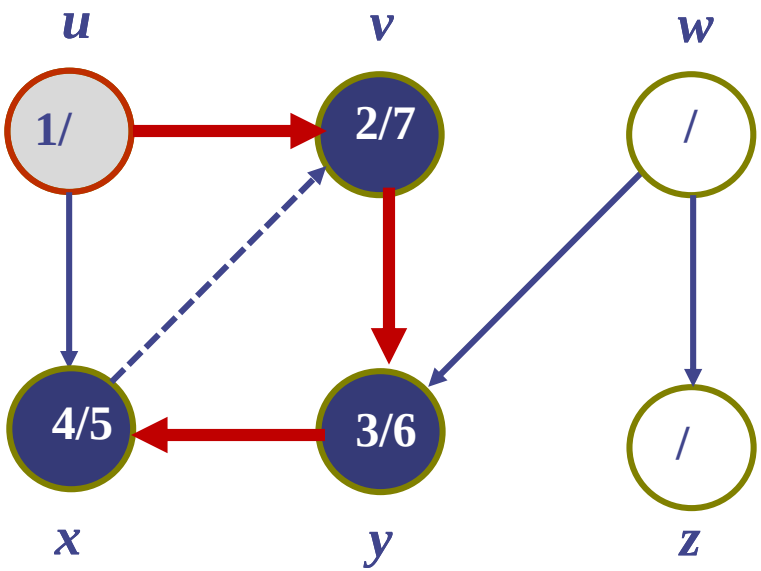
DFS(y)

- Kết thúc thăm đỉnh y



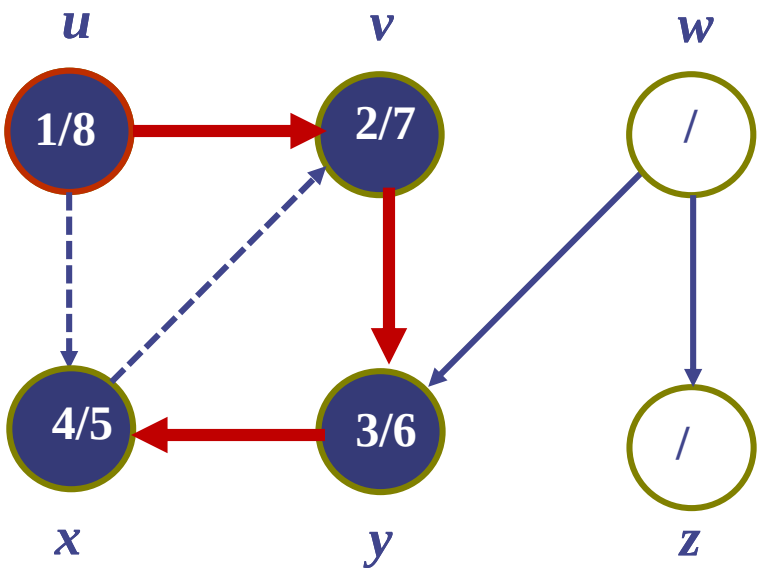
DFS(v)

- Kết thúc thăm đỉnh v



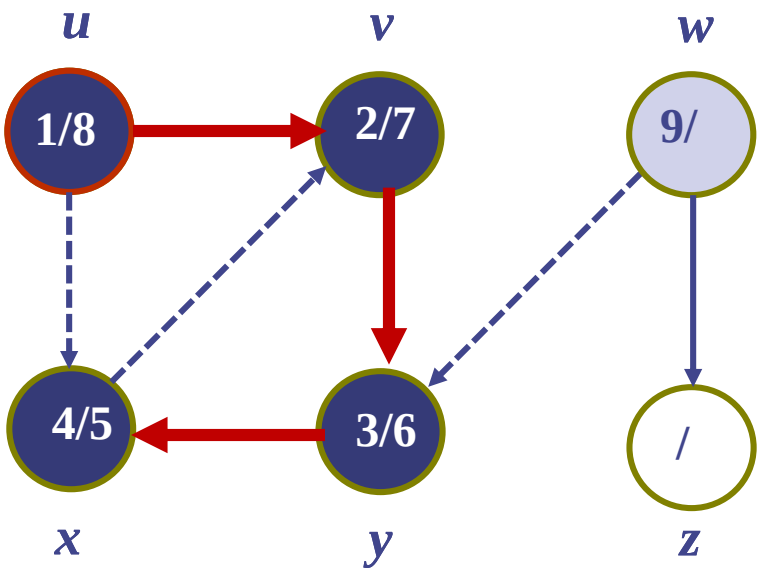
DFS(u)

- Kết thúc thăm đỉnh u



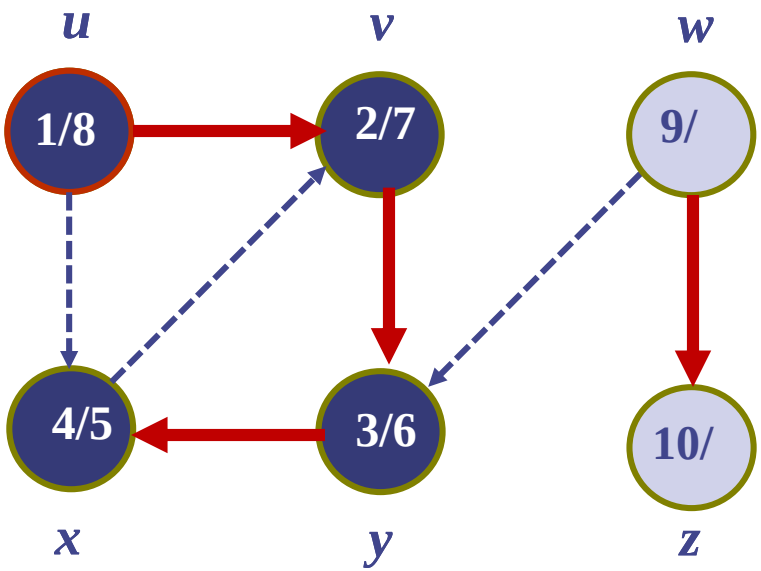
DFS(w)

- Thăm đỉnh w



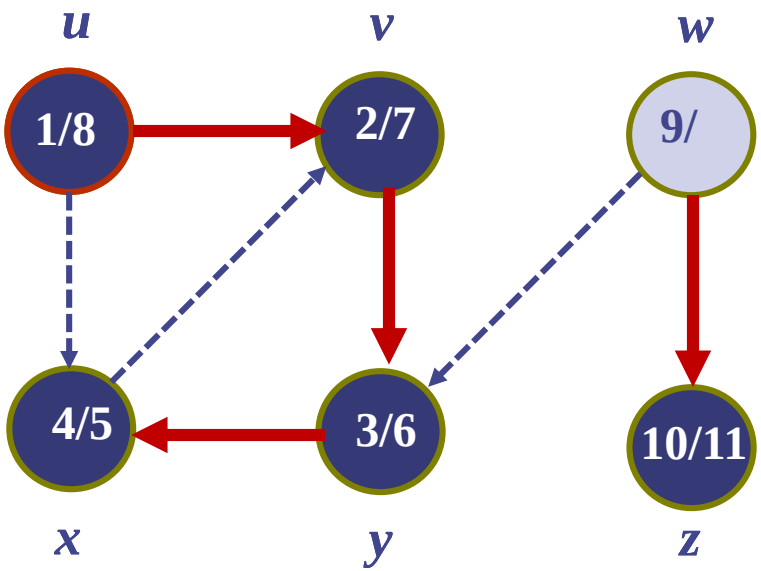
DFS(z)

- Thăm đỉnh z



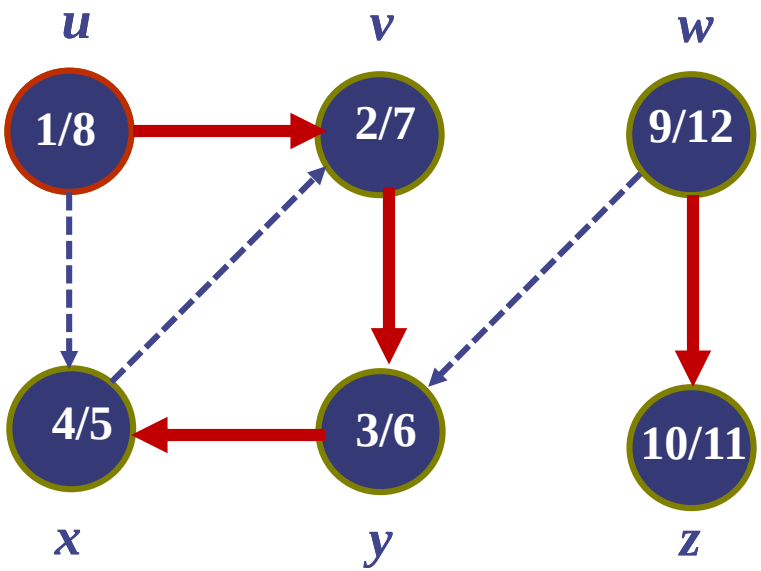
DFS(z)

- Kết thúc thăm đỉnh z



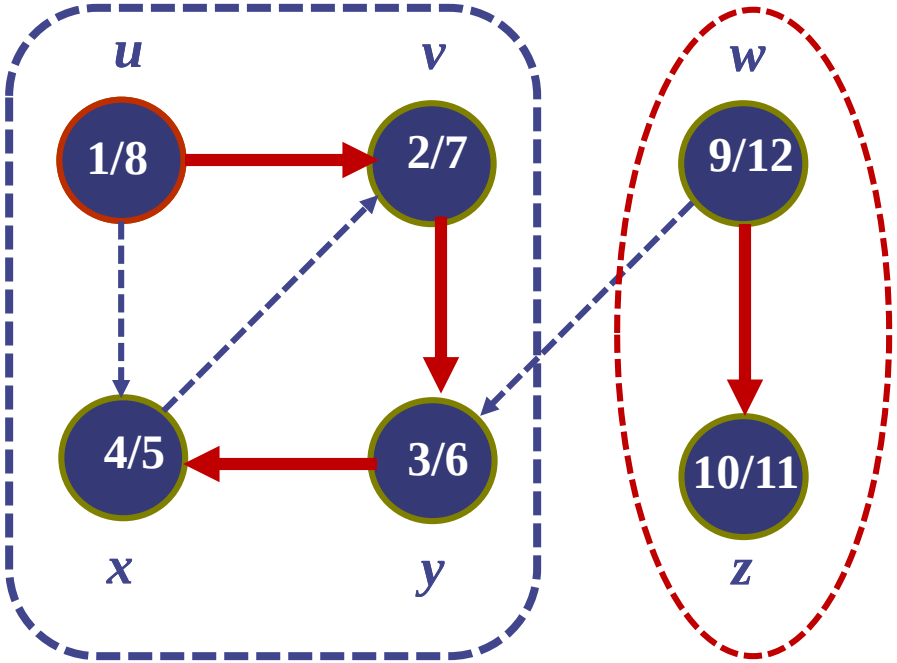
DFS(w)

- Kết thúc thăm đỉnh w

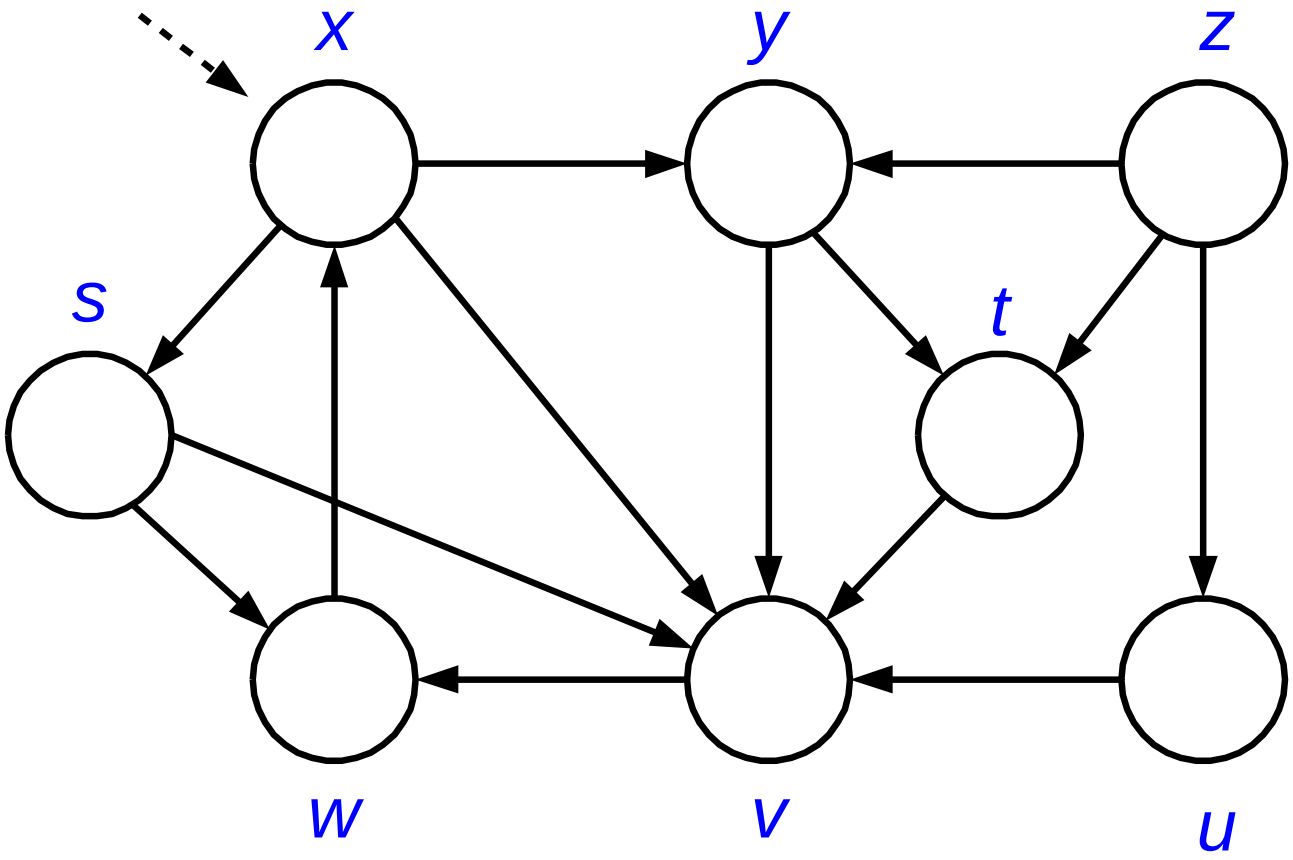


DFS(G): Kết thúc

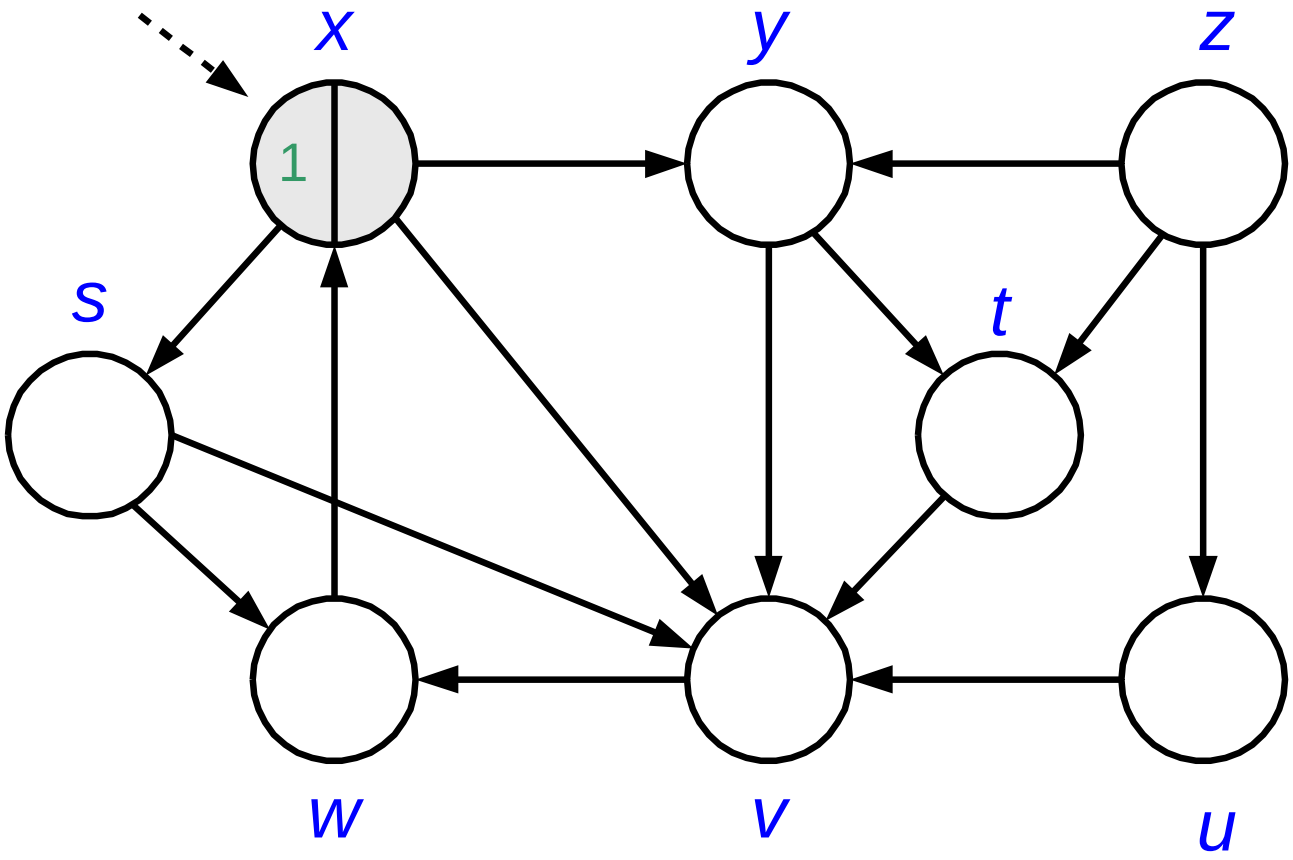
- Rừng tìm kiếm gồm 2 cây: Cây DFS(u) và cây DFS(w):



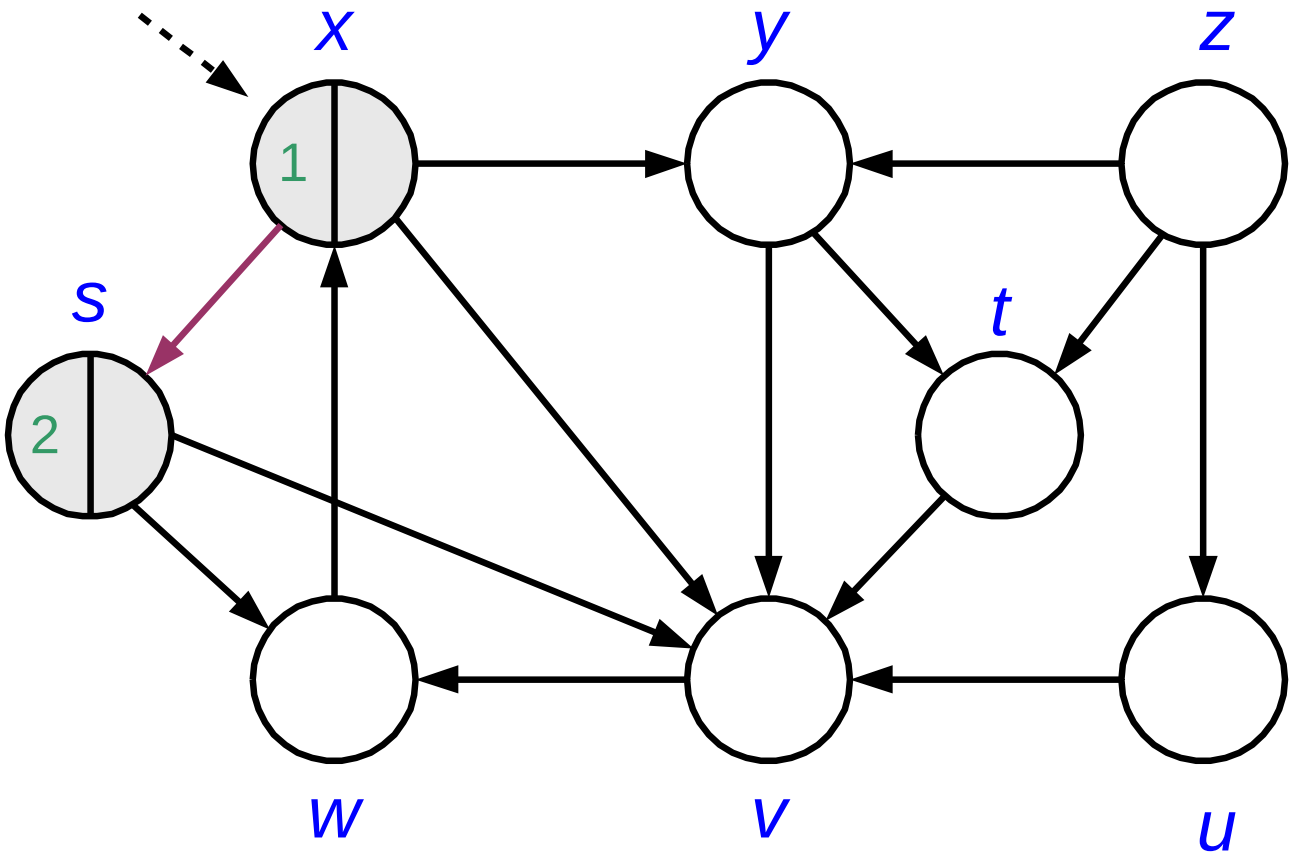
Depth-First Search: Ví dụ



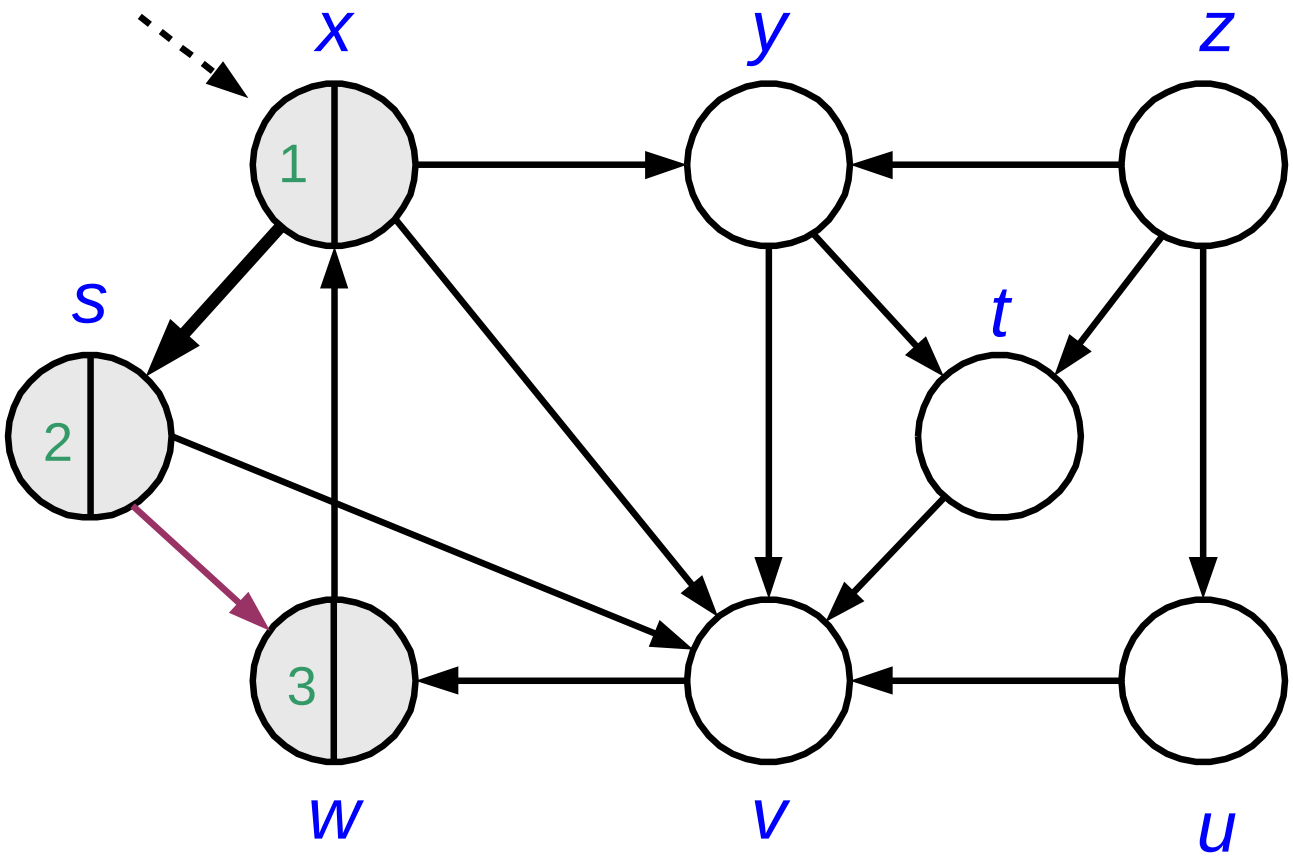
Depth-First Search: Ví dụ



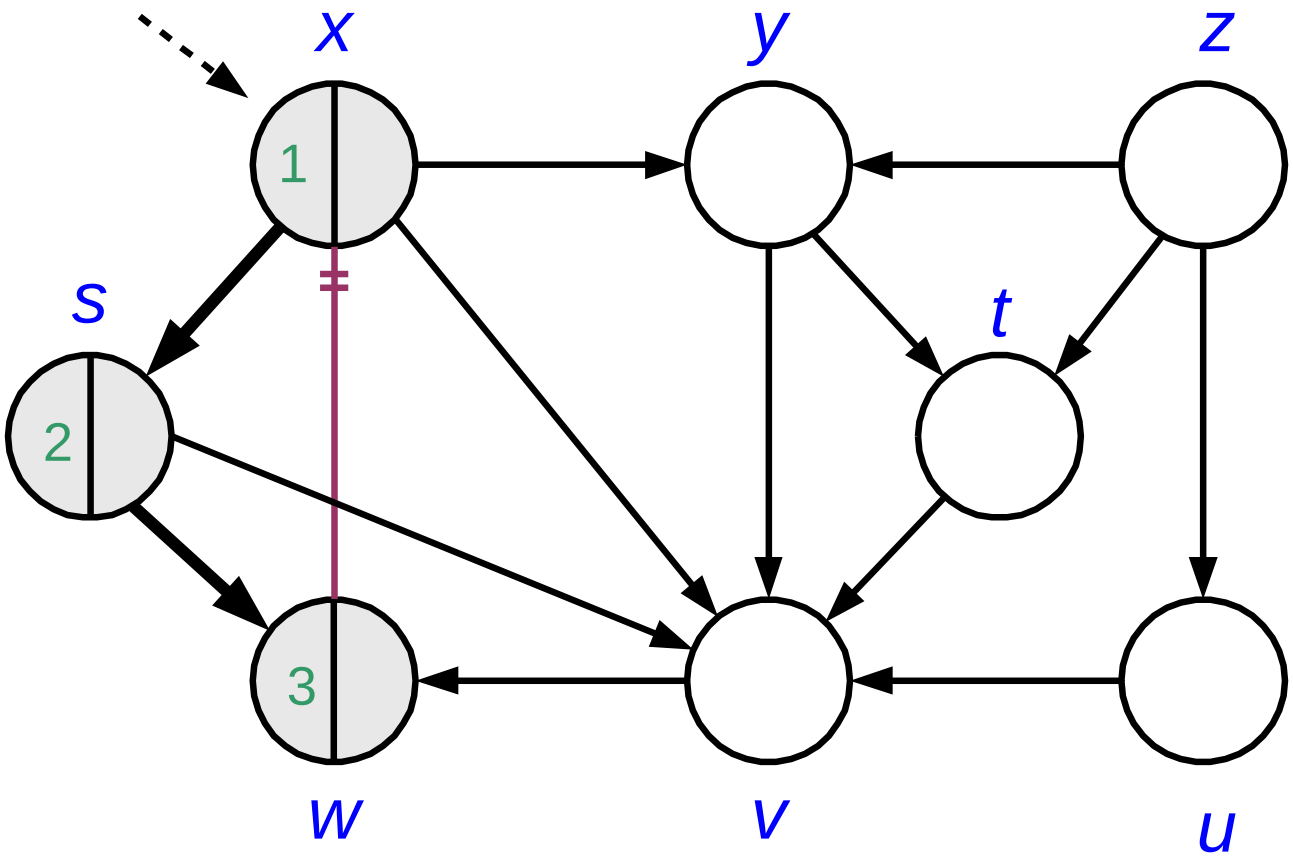
Depth-First Search: Ví dụ



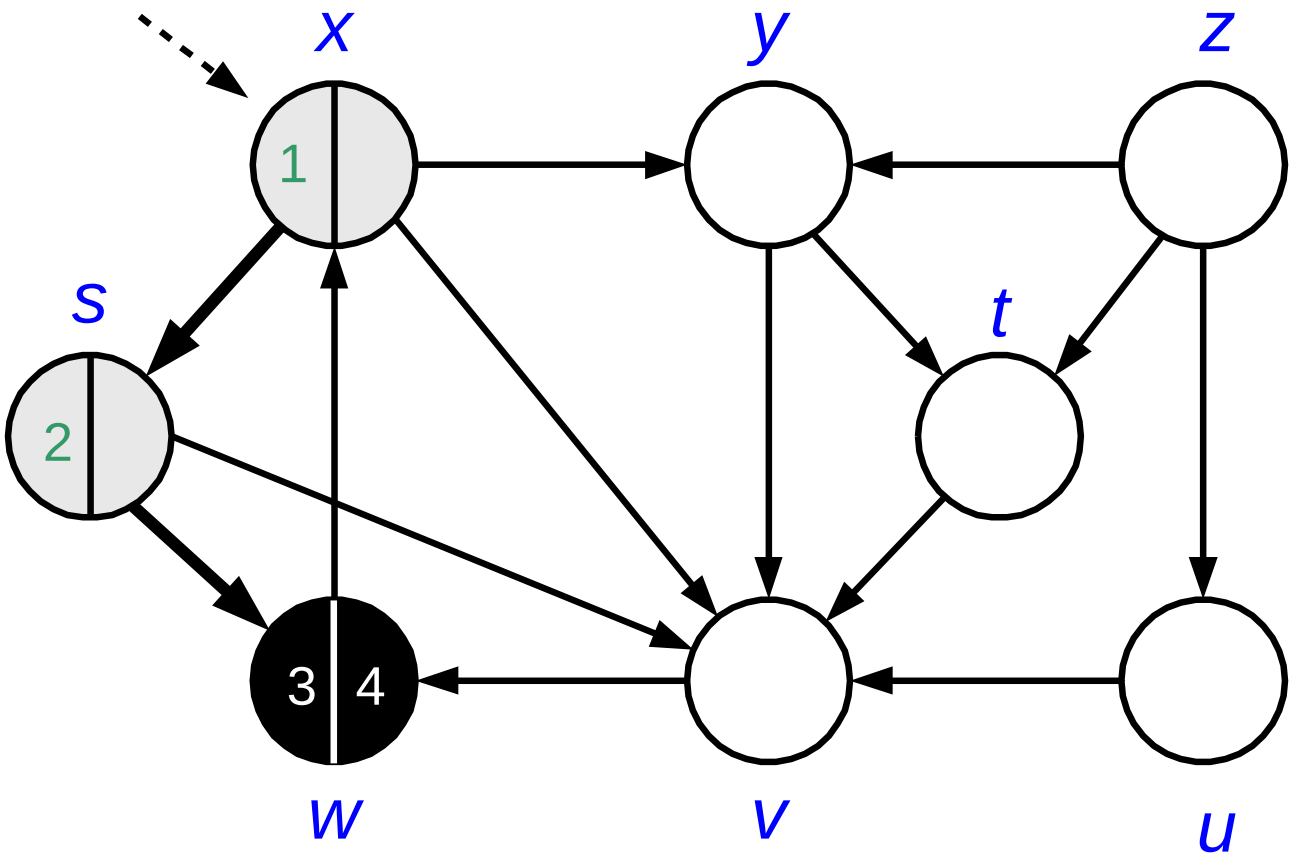
Depth-First Search: Ví dụ



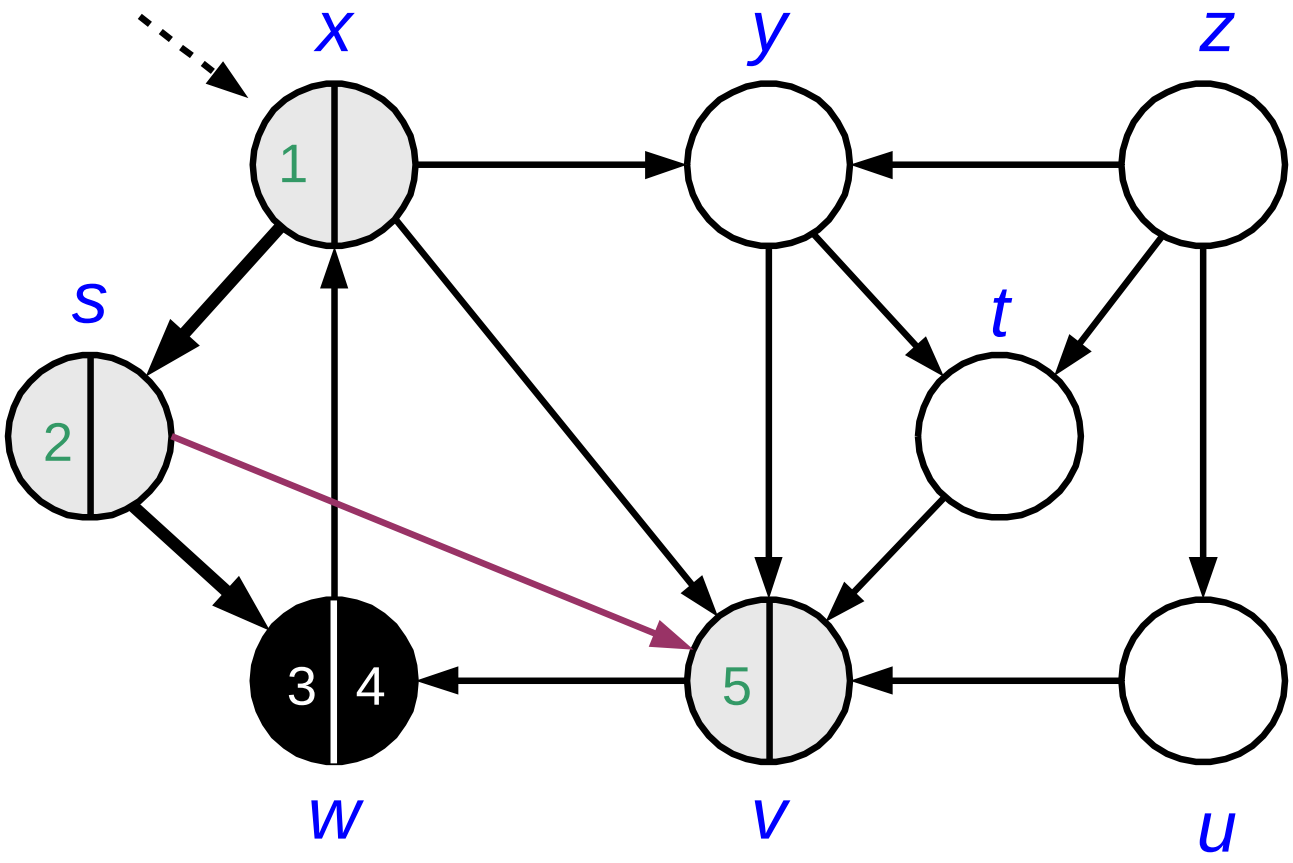
Depth-First Search: Ví dụ



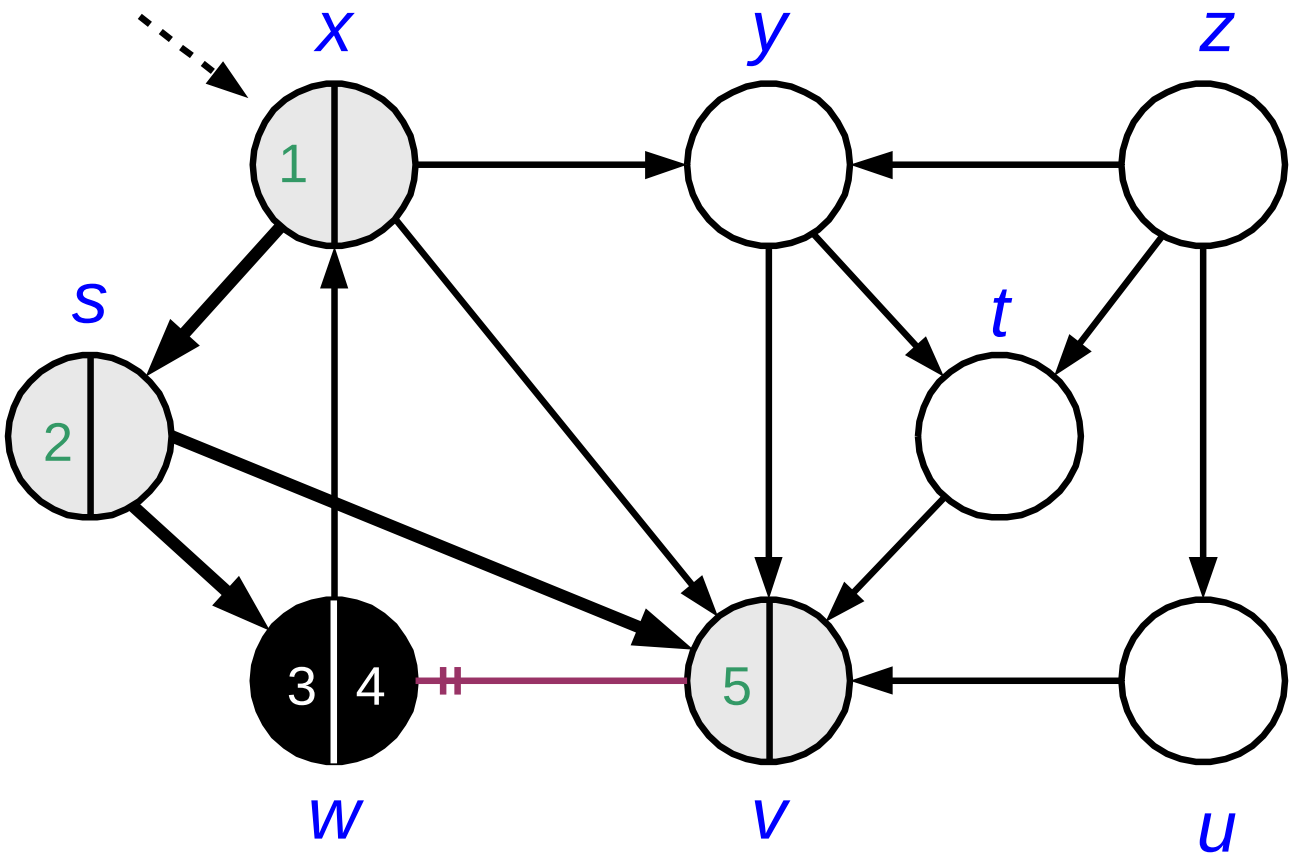
Depth-First Search: Ví dụ



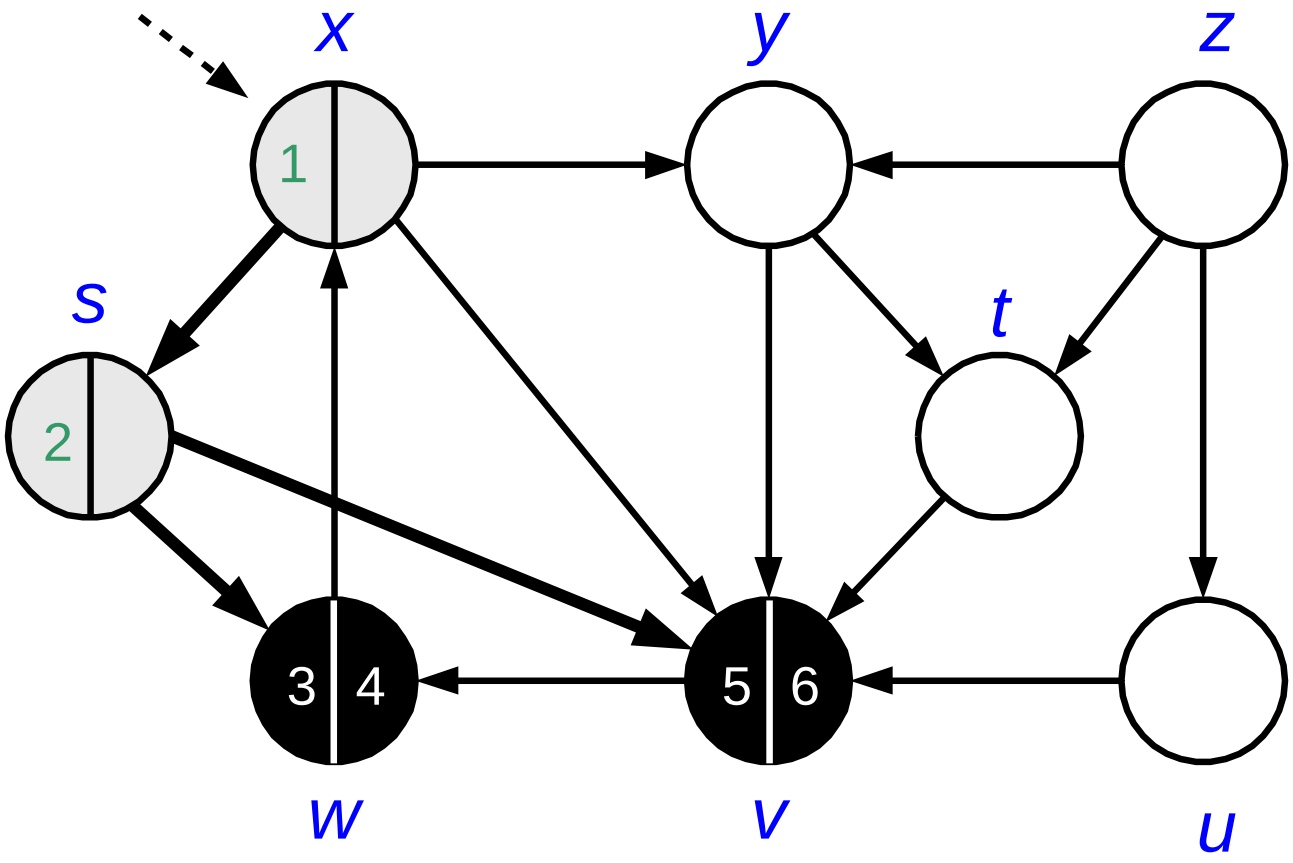
Depth-First Search: Ví dụ



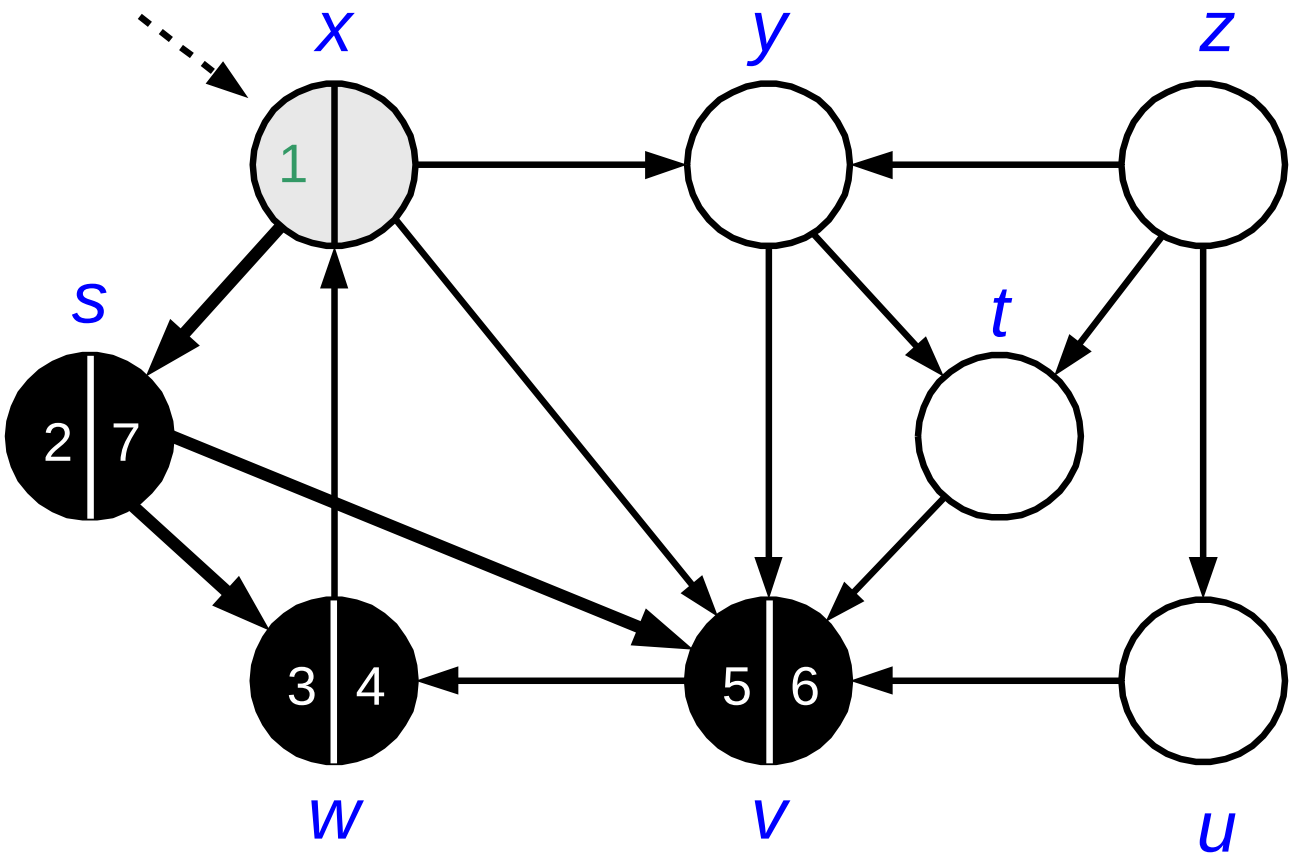
Depth-First Search: Ví dụ



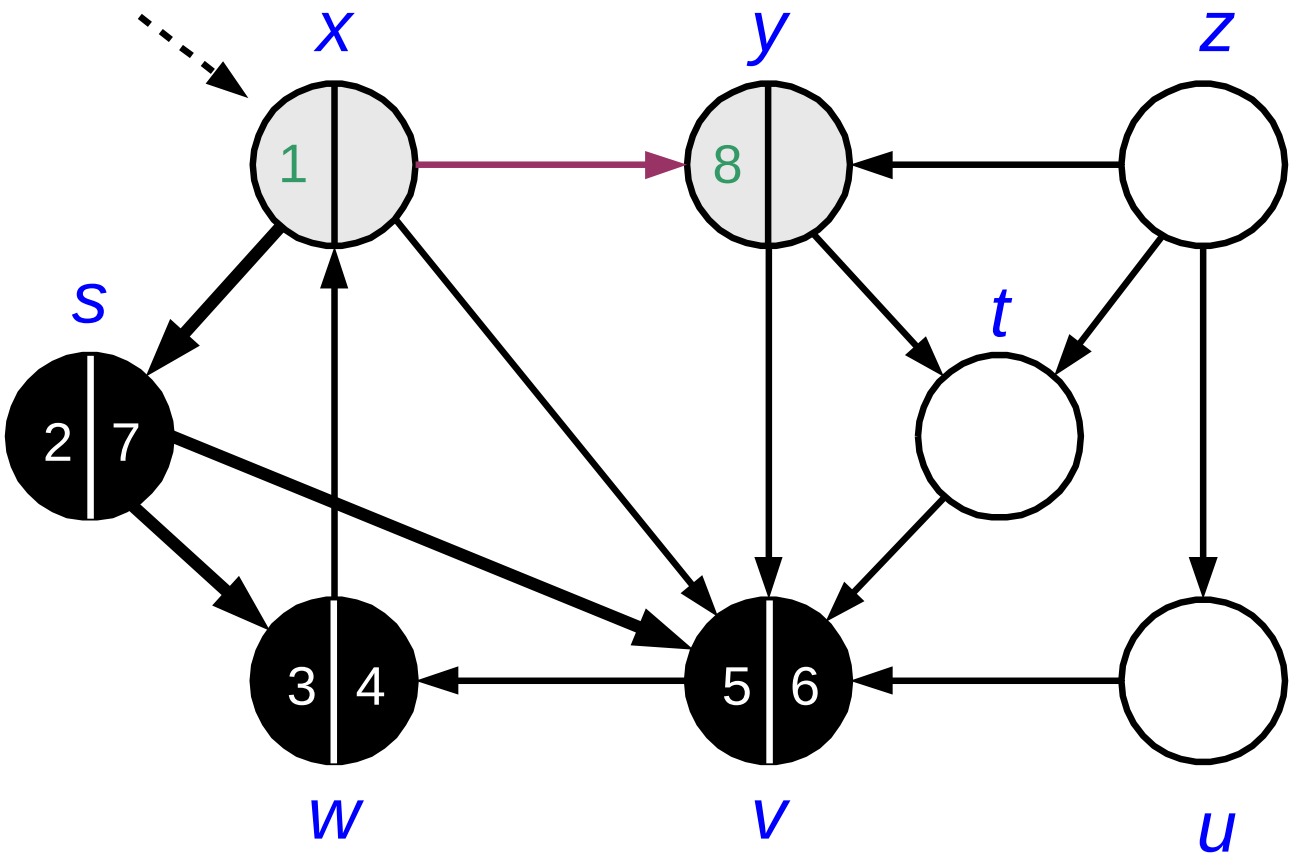
Depth-First Search: Ví dụ



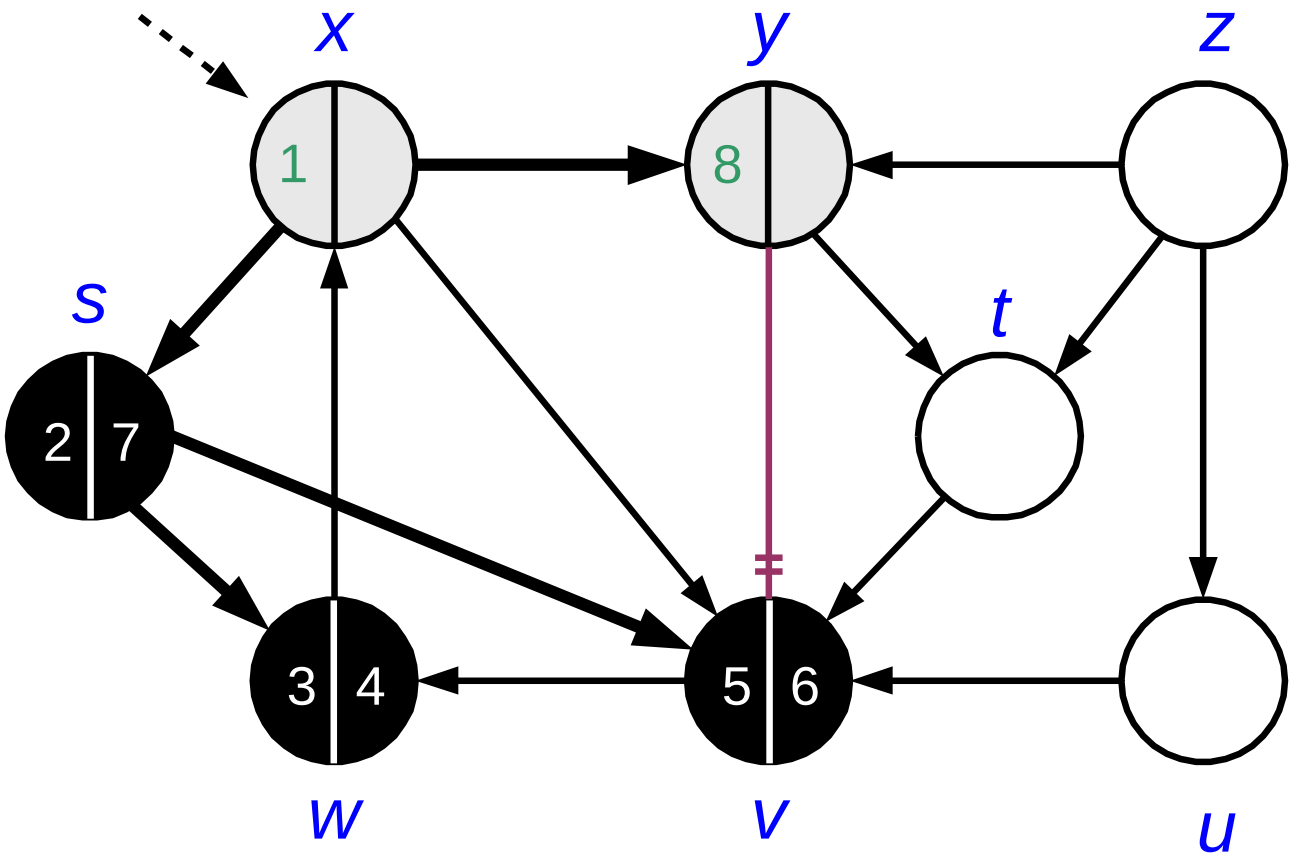
Depth-First Search: Ví dụ



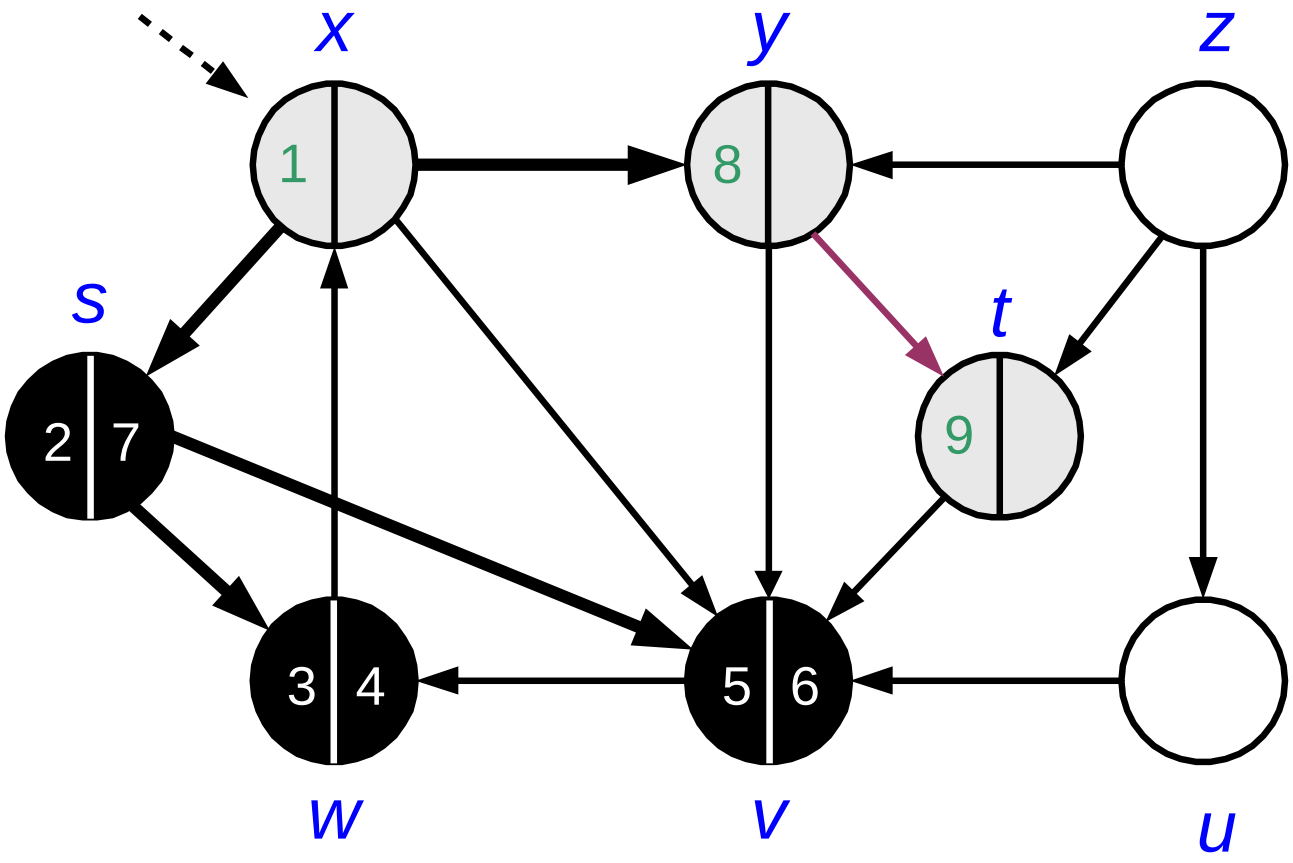
Depth-First Search: Ví dụ



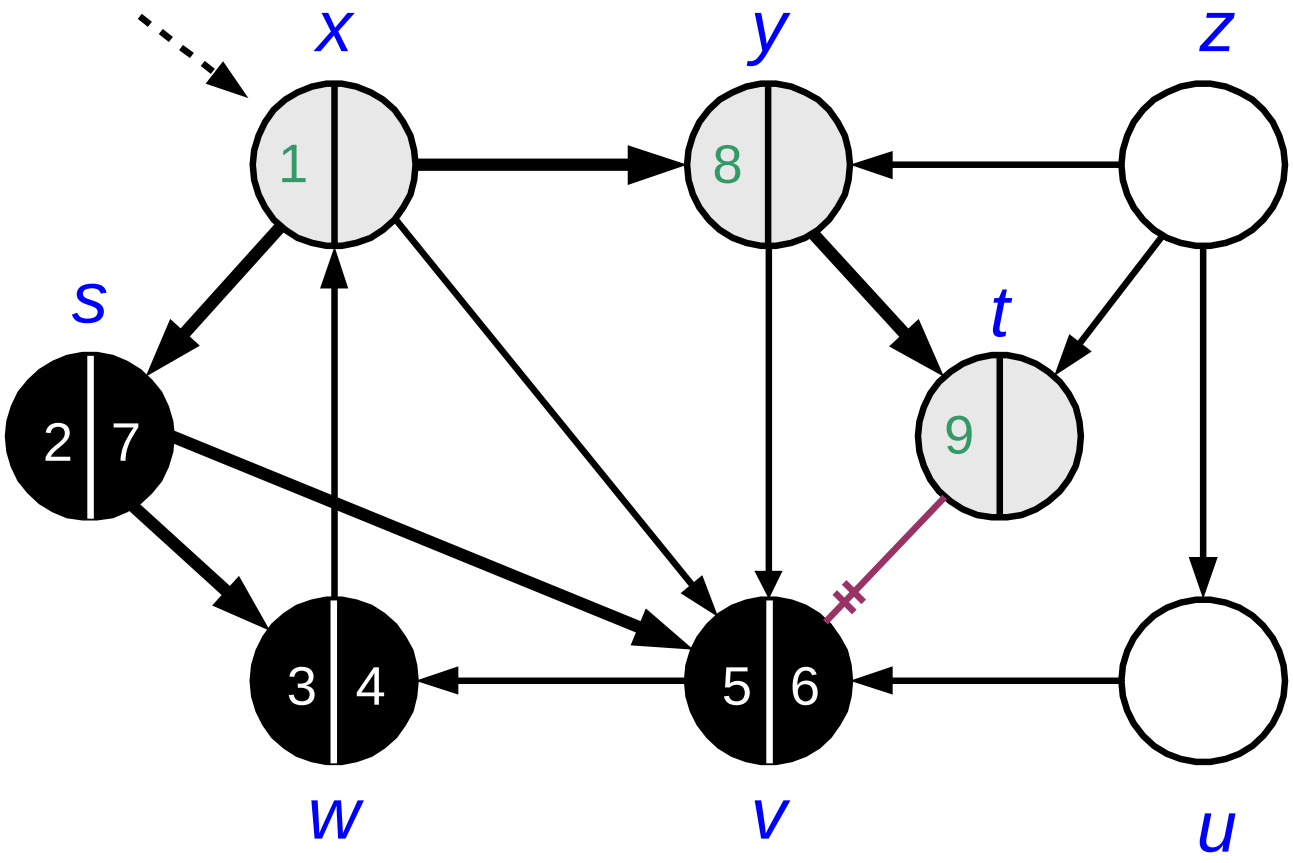
Depth-First Search: Ví dụ



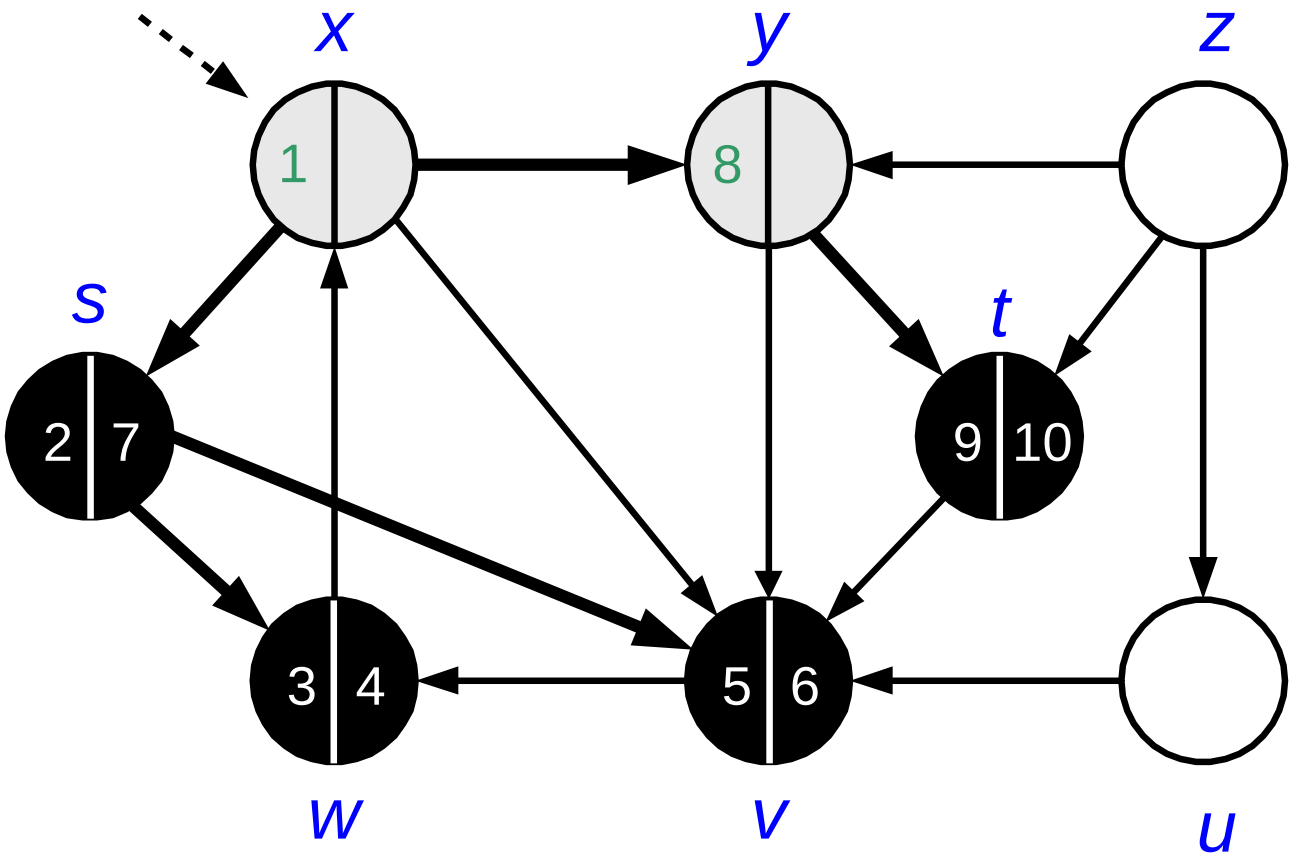
Depth-First Search: Ví dụ



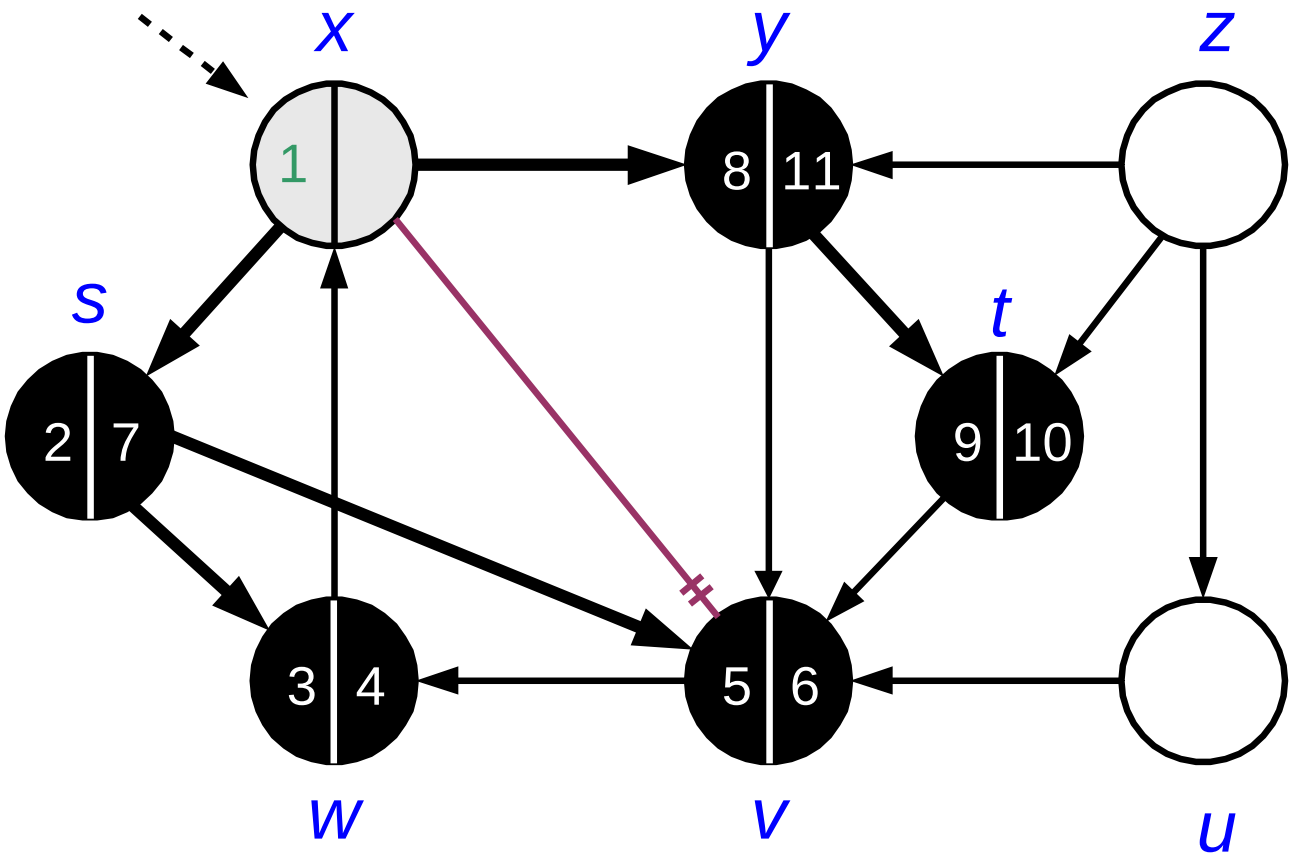
Depth-First Search: Ví dụ



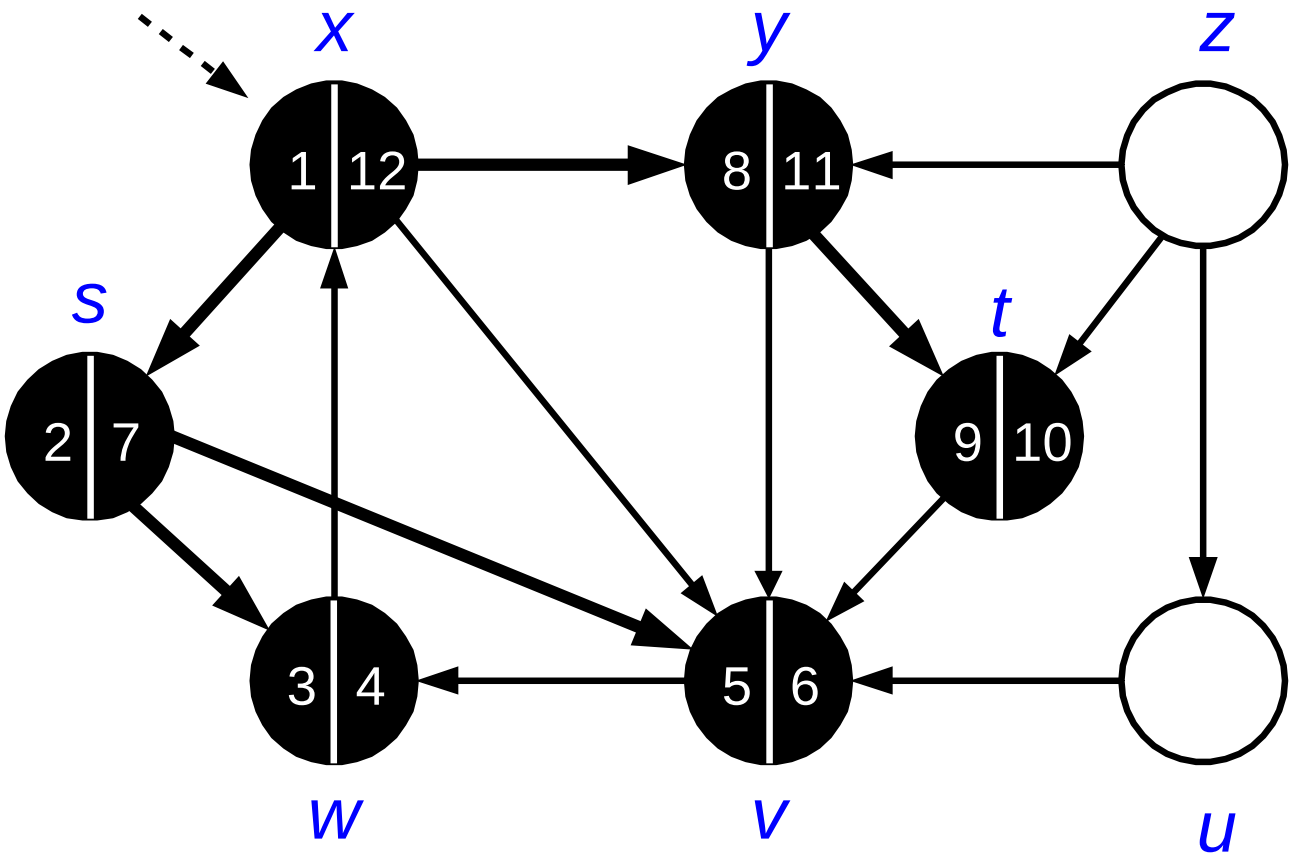
Depth-First Search: Ví dụ



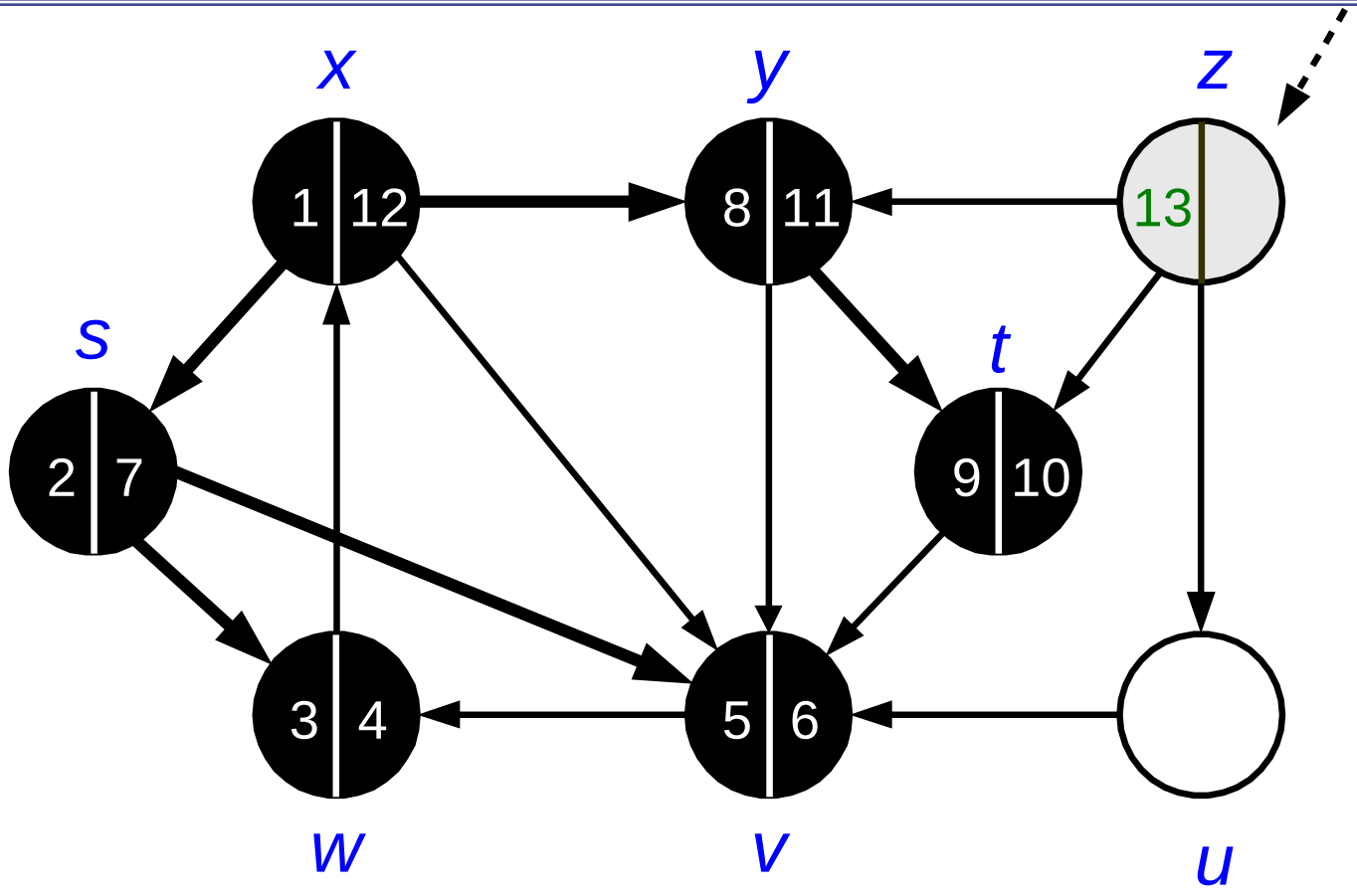
Depth-First Search: Ví dụ



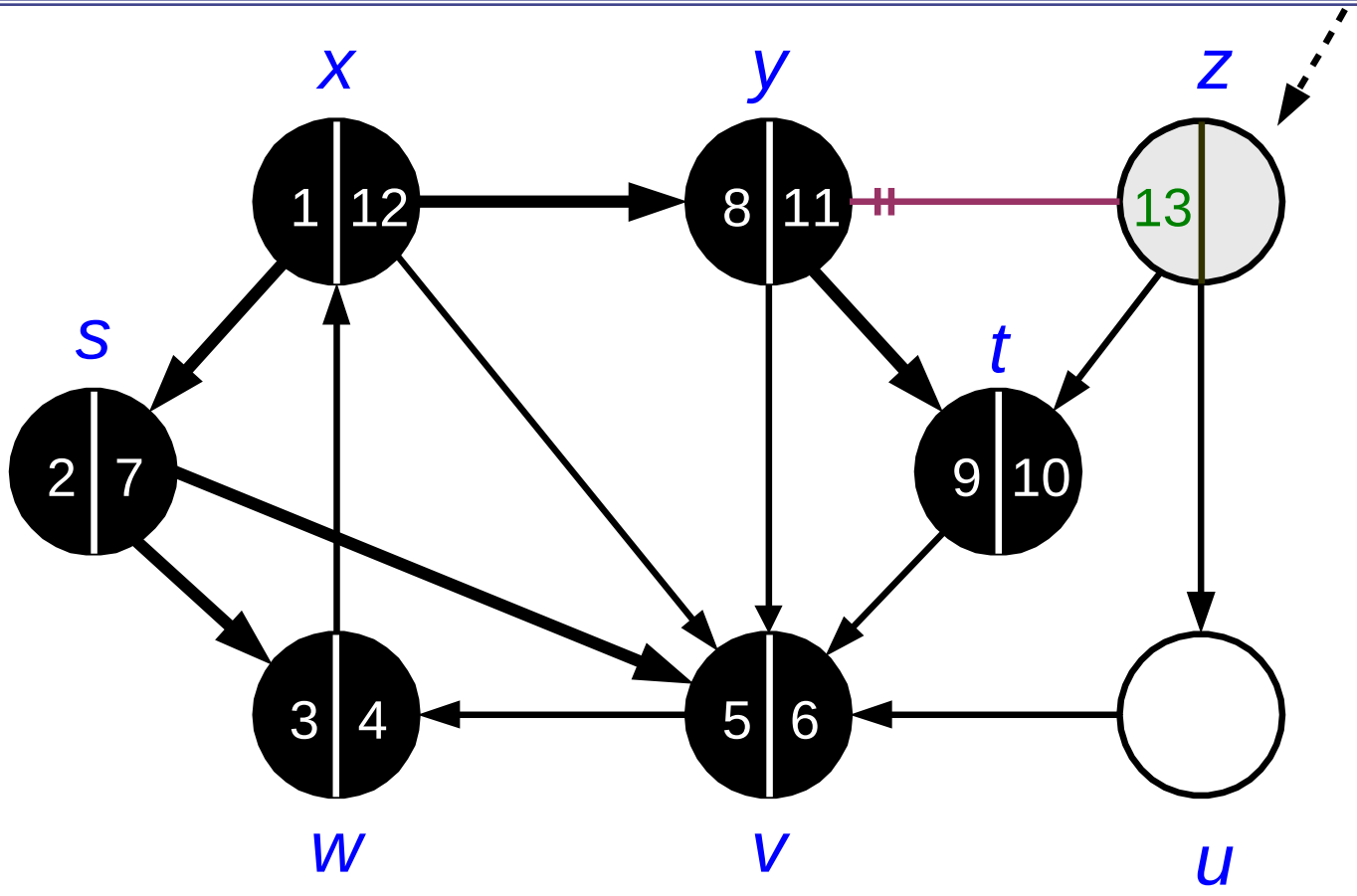
Depth-First Search: Ví dụ



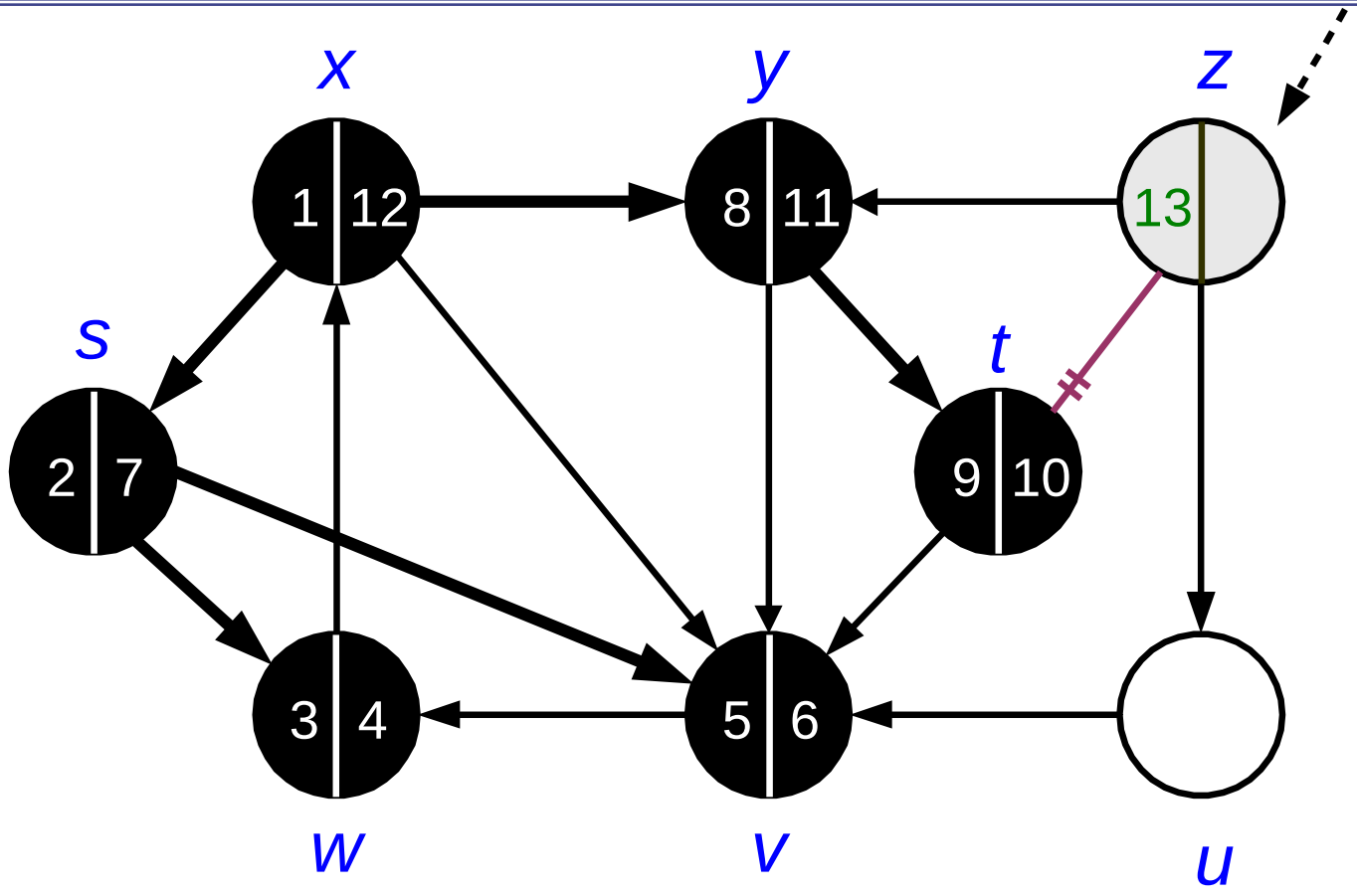
Depth-First Search: Ví dụ



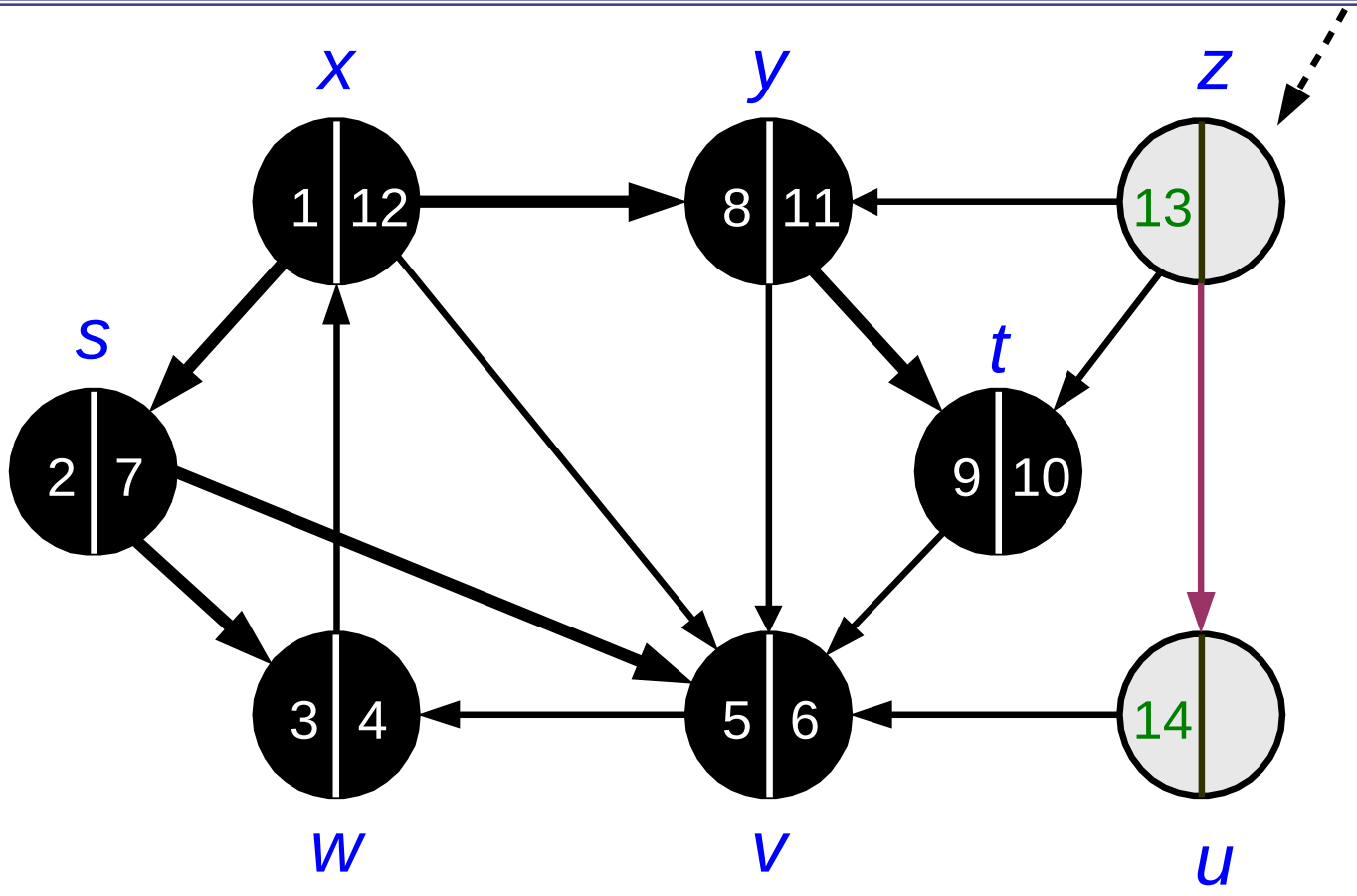
Depth-First Search: Ví dụ



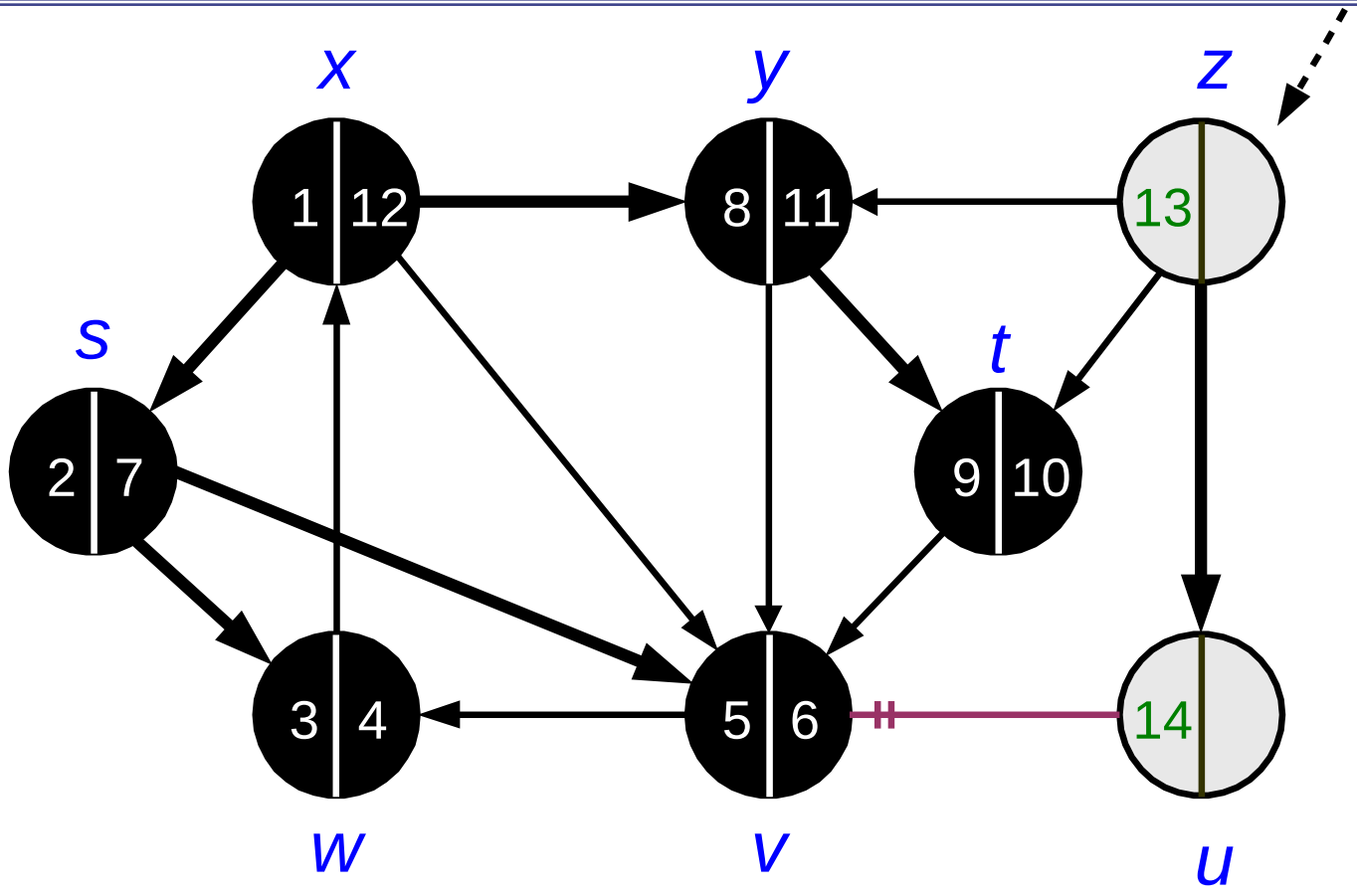
Depth-First Search: Ví dụ



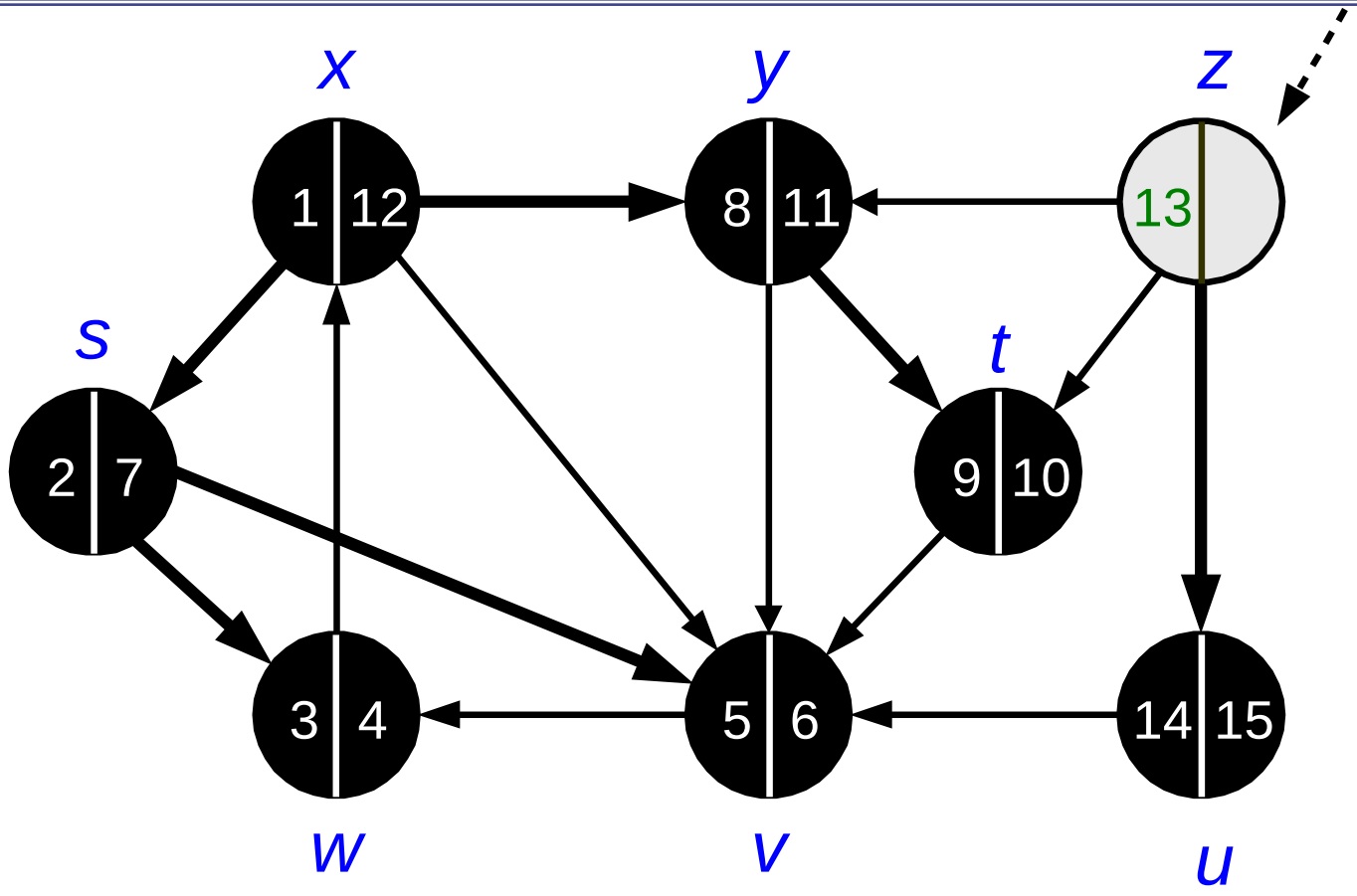
Depth-First Search: Ví dụ



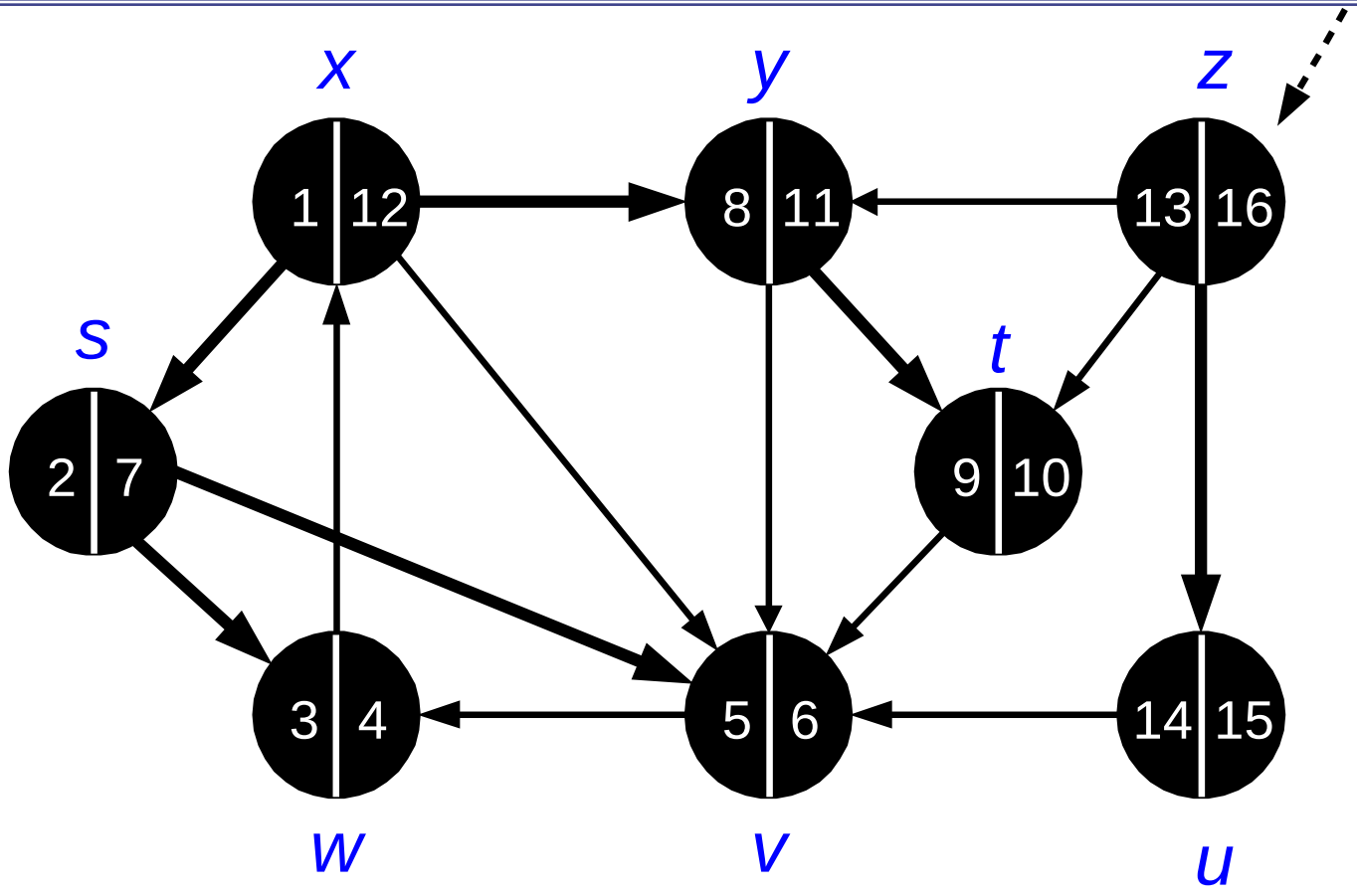
Depth-First Search: Ví dụ



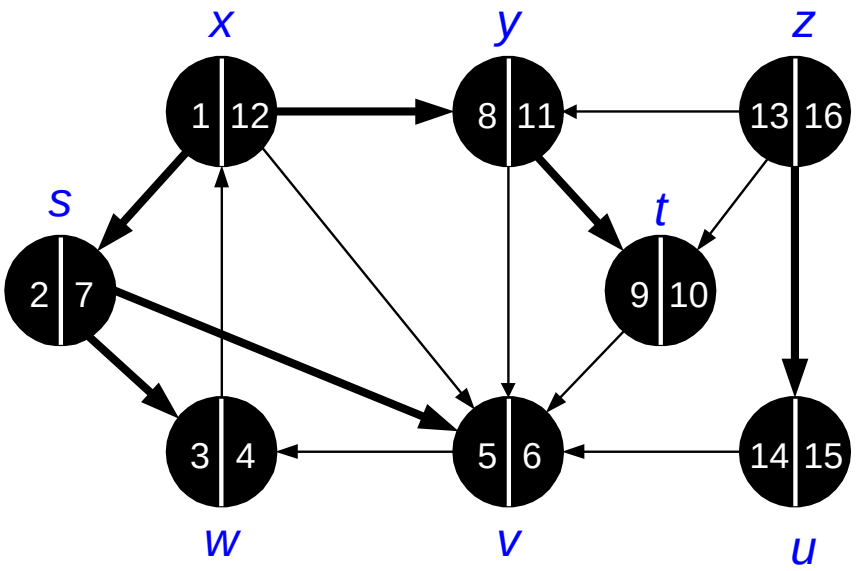
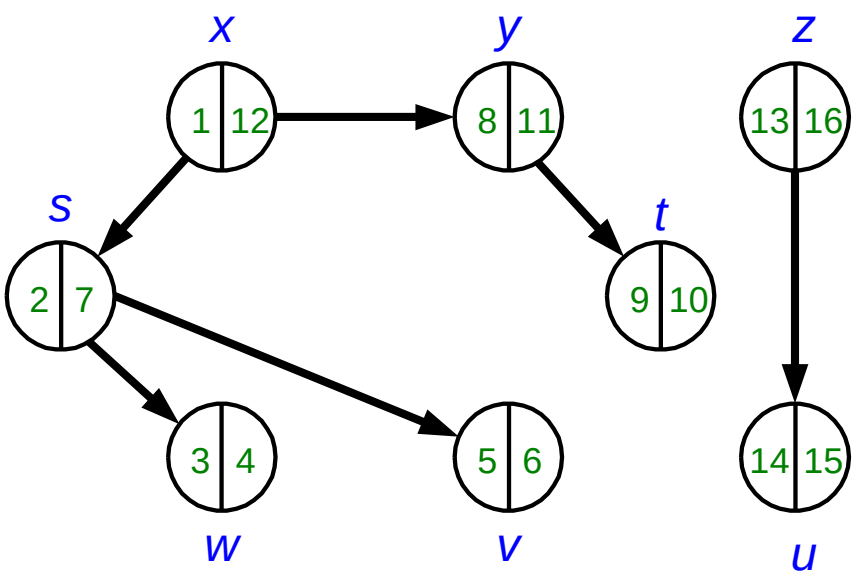
Depth-First Search: Ví dụ



Depth-First Search: Ví dụ



Depth-First Search: Ví dụ

DFS(G) dừng	Depth first search forest (DFS forest)
	

Các tính chất của DFS

- Rừng DFS là phụ thuộc vào thứ tự các đỉnh được duyệt trong các vòng lặp **for** duyệt đỉnh trong DFS(G) và DFS_Visit(u).
- Để gỡ đệ qui có thể sử dụng ngăn xếp. Có thể nói, điểm khác biệt cơ bản của DFS với BFS là các đỉnh đang được thăm trong DFS được cất giữ vào ngăn xếp thay vì hàng đợi trong BFS.
- Các khoảng thời gian thăm [$d[v]$, $f[v]$] của các đỉnh có cấu trúc lồng nhau (*parenthesis structure*).

Cấu trúc lồng nhau (parenthesis structure)

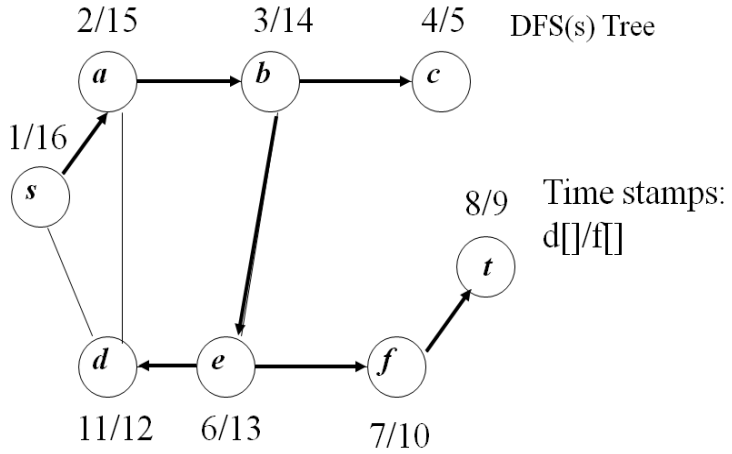
- **Định lý.** Với mọi u, v, chỉ có thể xảy ra một trong các tình huống sau:

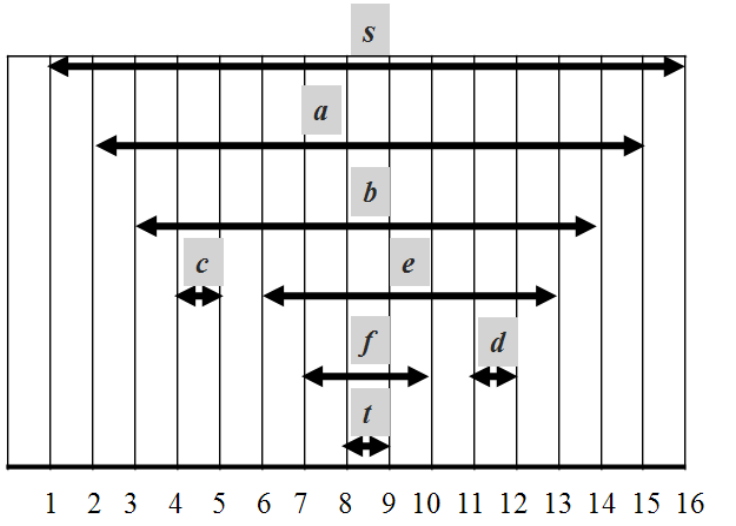
  1. $d[u] < f[u] < d[v] < f[v]$ hoặc $d[v] < f[v] < d[u] < f[u]$ (nghĩa là hai khoảng thời gian thăm của u và v là dời nhau) và khi đó u và v là không có quan hệ tổ tiên – hậu duệ.

  2. $d[u] < d[v] < f[v] < f[u]$ (nghĩa là khoảng thời gian thăm của v là lồng trong khoảng thời gian thăm của u) và khi đó v là hậu duệ của u.

  3. $d[v] < d[u] < f[u] < f[v]$ (nghĩa là khoảng thời gian thăm của u là lồng trong khoảng thời gian thăm của v) và khi đó u là hậu duệ của v.

Ví dụ





(*s* (*a* (*b* (*c* *c*) (*e* (*f* (*t* *t*) *f*) (*d* *d*) *e*) *b*) *a*) *s*)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Độ phức tạp của DFS

- Thuật toán thăm mỗi đỉnh $v \in V$ đúng một lần, việc thăm đỉnh đòi hỏi thời gian $\Theta(|V|)$
- Với mỗi đỉnh v duyệt qua tất cả các đỉnh kề, với mỗi đỉnh kề thực hiện thao tác với thời gian hằng số. Do đó việc duyệt qua tất cả các đỉnh mất thời gian:

$$\Sigma_{v \in V} |neighbors[v]| = \Theta(|E|)$$

- Tổng cộng: $\Theta(|V|) + \Theta(|E|) = \Theta(|V|+|E|)$, hay $\Theta(|V|^2)$
- Như vậy, DFS có cùng độ phức tạp như BFS.

Phân loại cạnh

- DFS tạo ra một cách phân loại các cạnh của đồ thị đã cho:
  - ***Cạnh của cây*** (*Tree edge*): là cạnh mà theo đó từ một đỉnh ta đến thăm một đỉnh mới
  - ***Cạnh ngược*** (*Back edge*): đi từ con cháu (descendent) đến tổ tiên (ancestor)
  - ***Cạnh tới*** (*Forward edge*): đi từ tổ tiên đến hậu duệ
  - ***Cạnh vòng*** (*Cross edge*): cạnh nối hai đỉnh không có quan hệ họ hàng.

Nhận biết các loại cạnh

- Để nhận biết cạnh (u, v) thuộc loại cạnh nào, ta dựa vào màu của đỉnh v khi lần đầu tiên cạnh (u, v) được khảo sát. Cụ thể, nếu màu của đỉnh v là
  - Trắng, thì (u,v) là cạnh của cây;
  - Xám, thì (u,v) là cạnh ngược;
  - Đen, thì (u,v) là cạnh tới hoặc vòng. Trong trường hợp này để phân biệt cạnh tới và cạnh vòng ta cần xét xem hai đỉnh u và v có quan hệ họ hàng hay không nhờ sử dụng kết quả của định lý về cấu trúc lồng nhau.

DFS trên đồ thị vô hướng

- **Định lý.** Nếu G là đồ thị vô hướng, thì DFS chỉ sản sinh ra cạnh của cây và cạnh ngược.
- **Chứng minh.**
  - Giả sử $(u,v) \in E$. Không giảm tổng quát giả sử $d[u] < d[v]$. Khi đó v phải trở thành đã duyệt xong trước khi u trở thành đã duyệt xong.
  - Nếu (u,v) được khảo sát lần đầu tiên theo hướng $u \rightarrow v$, thì trước thời điểm khảo sát v phải có màu trắng, và do đó (u,v) là cạnh của cây.
  - Nếu (u,v) được khảo sát lần đầu tiên theo hướng $v \rightarrow u$, u phải có màu xám tại thời điểm khảo sát cạnh này và do đó nó là cạnh ngược.

NỘI DUNG

7.1. Đồ thị

7.2. Biểu diễn đồ thị

7.3. Các thuật toán duyệt đồ thị

7.4. Một số ứng dụng của tìm kiếm trên đồ thị

7.5. Bài toán cây khung nhỏ nhất

7.6. Bài toán đường đi ngắn nhất

7.4. Một số ứng dụng của tìm kiếm trên đồ thị

- Bài toán đường đi
- Bài toán liên thông
- Bài toán liên thông mạnh
- Bài toán chu trình
- Bài toán sắp xếp tôpô
- Bài toán tìm bao đóng truyền ứng

Các ứng dụng của DFS và BFS

- Các thuật toán tìm kiếm trên đồ thị BFS và DFS được ứng dụng để giải nhiều bài toán trên đồ thị, chẳng hạn như
  - Tìm đường đi giữa hai đỉnh s và t của đồ thị;
  - Kiểm tra tính liên thông, liên thông mạnh của đồ thị;
  - Xác định các thành phần liên thông, song liên thông, liên thông mạnh;
  - Tính hai phía của đồ thị;
  - Tính phẳng của đồ thị.
  - ...

Bài toán đường đi

- **Bài toán đặt ra là:** "Cho đồ thị $G=(V,E)$ và hai đỉnh s, t của nó. Hỏi có tồn tại đường đi từ s đến t hay không? Trong trường hợp câu trả lời là khẳng định cần đưa ra một đường đi từ s đến t."
- Để giải bài toán này ta có thể thực hiện DFS_Visit(s) hoặc BFS_Visit(s). Kết thúc, nếu đỉnh t là được thăm thì câu trả lời là khẳng định và khi đó để đưa ra đường đi từ s đến t ta sử dụng biến ghi nhận $\pi[v]$:

$$t \leftarrow \pi[t] \leftarrow \pi[\pi[t]] \leftarrow \ldots \leftarrow s.$$

- Nếu t không được thăm, ta khẳng định là không có đường đi cần tìm.
- **Chú ý:** Đường đi tìm được từ s đến t theo BFS_Visit(s) là đường đi ngắn nhất (theo số cạnh).

Bài toán liên thông

- **Bài toán liên thông.** Cho đồ thị vô hướng $G=(V,E)$. Hãy kiểm tra xem đồ thị G có phải liên thông hay không. Nếu G không là liên thông, cần đưa ra số lượng thành phần liên thông và danh sách các đỉnh của từng thành phần liên thông.

- Để giải bài toán này, ta chỉ việc thực hiện DFS(G) (hoặc BFS(G)). Khi đó số lần gọi thực hiện BFS_Visit() (DFS_Visit()) sẽ chính là số lượng thành phần liên thông của đồ thị. Việc đưa ra danh sách các đỉnh của từng thành phần liên thông sẽ đòi hỏi phải đưa thêm vào biến ghi nhận xem mỗi đỉnh được thăm ở lần gọi nào trong BFS(G) (DFS(G)).

Bài toán liên thông mạnh

- **Bài toán liên thông mạnh.** Cho đồ thị có hướng $G=(V,E)$. Hãy kiểm tra xem đồ thị G có phải liên thông mạnh hay không?
- Kết quả sau đây cho phép qui dẫn bài toán cần giải về bài toán đường đi.
- **Mệnh đề.** Đồ thị có hướng $G=(V,E)$ là liên thông mạnh khi và chỉ khi luôn tìm được đường đi từ một đỉnh v đến tất cả các đỉnh còn lại và luôn tìm được đường đi từ tất cả các đỉnh thuộc $V \setminus \{v\}$ đến v.
- **Chứng minh.** Suy trực tiếp từ định nghĩa đồ thị có hướng liên thông mạnh.

Đồ thị đảo hướng (đồ thị chuyển vị)

- Cho đồ thị có hướng $G=(V,E)$. Ta gọi đồ thị đảo hướng (đồ thị chuyển vị) của đồ thị G là đồ thị có hướng $G^T = (V, E^T)$, với $E^T = \{(u, v): (v, u) \in E\}$, nghĩa là tập cung E^T thu được từ E bởi việc đảo ngược hướng của tất cả các cung.
- Dễ thấy nếu A là ma trận kề của G thì ma trận chuyển vị A^T là ma trận kề của G^T (điều này giải thích tên gọi đồ thị chuyển vị).

Thuật toán kiểm tra tính liên thông mạnh

- Chọn $v \in V$ là một đỉnh tuỳ ý.
- Thực hiện DFS(v) trên G. Nếu tồn tại đỉnh u không được thăm thì G không liên thông mạnh và thuật toán kết thúc. Trái lại thực hiện tiếp
- Thực hiện DFS(v) trên $G^T = (V, E^T)$. Nếu tồn tại đỉnh u không được thăm thì G không liên thông mạnh, nếu trái lại G là liên thông mạnh.

Đồ thị không chứa chu trình

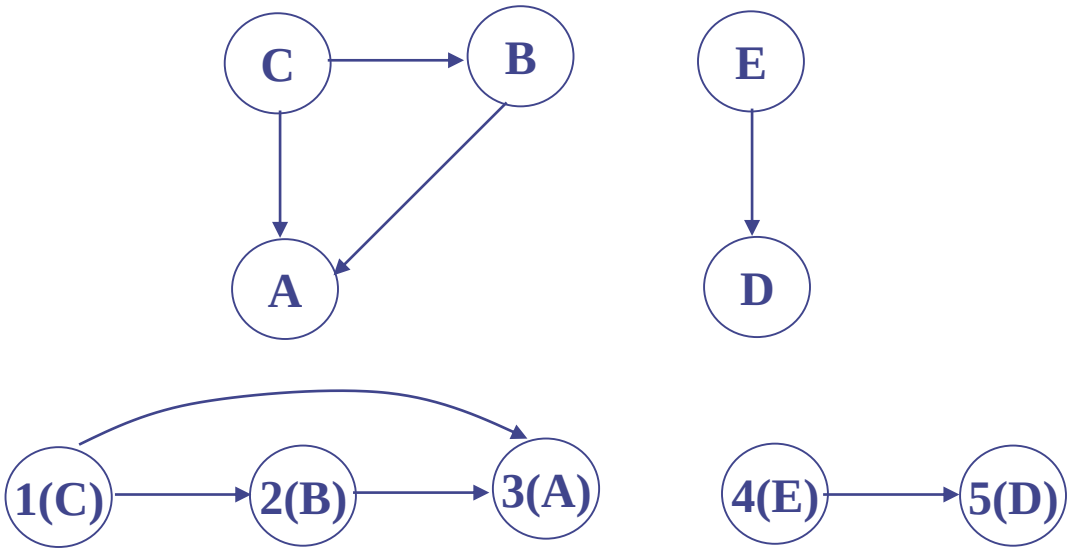
- **Bài toán:** Cho đồ thị $G=(V,E)$. Hỏi G có chứa chu trình hay không
- **Mệnh đề.** Đồ thị G là không chứa chu trình khi và chỉ khi DFS thực hiện đối với G không phát hiện ra cạnh ngược.
- **Chứng minh**
  - $\Rightarrow$) Nếu G không chứa chu trình thì không thể có cạnh ngược. Hiển nhiên: bởi vì sự tồn tại cạnh ngược kéo theo sự tồn tại chu trình.
  - $\Leftarrow$) Ta phải chứng minh: Nếu không có cạnh ngược thì G là á chu trình. Ta chứng minh bằng lập luận phản đề: G có chu trình $\Rightarrow \exists$ cạnh ngược. Gọi v là đỉnh trên chu trình được thăm đầu tiên, và u là đỉnh đi trước v trên chu trình. Khi v được thăm, các đỉnh khác trên chu trình đều là đỉnh trắng.Ta phải thăm được tất cả các đỉnh đạt được từ v trước khi quay trở lại từ DFS-Visit(). Vì thế cạnh $u \rightarrow v$ được duyệt từ đỉnh u về tổ tiên v của nó, vì thế (u, v) là cạnh ngược.

Phân tích độ phức tạp

- Độ phức tạp: Chính là độ phức tạp của DFS – $O(|V|+|E|)$.
- Tuy nhiên, đối với đồ thị vô hướng, thời gian tính là $O(|V|)$.
- **Mệnh đề.** Đơn đồ thị vô hướng với n đỉnh và n cạnh chắc chắn chứa chu trình.

Bài toán sắp xếp tôpô (Topological Sort)

- **Bài toán đặt ra là:** Cho đồ thị có hướng không có chu trình $G = (V, E)$. Hãy tìm cách sắp xếp các đỉnh sao cho nếu có cạnh (u,v) thì u phải đi trước v trong thứ tự đó (nói cách khác, cần tìm cách đánh số các đỉnh của đồ thị sao cho mỗi cung của đồ thị luôn hướng từ đỉnh có chỉ số nhỏ hơn đến đỉnh có chỉ số lớn hơn).
- Ví dụ



Cơ sở thuật toán

- **Mệnh đề.** Nếu có cung (u,v) thì $f[u] > f[v]$ trong DFS.
- **Chứng minh.**
  - Khi cung (u,v) được khảo sát, thì u có màu xám. Khi đó v có thể có một trong 3 màu: xám, trắng, đen.
  - Nếu v có màu xám $\Rightarrow$ (u,v) là cạnh ngược $\Rightarrow$ Tồn tại chu trình?
  - Nếu v có màu trắng $\Rightarrow$ v trở thành con cháu của u $\Rightarrow$ $f[v] < f[u]$.
  - Nếu v có màu đen $\Rightarrow$ v đã duyệt xong $\Rightarrow$ $f[v] < f[u]$.

Thuật toán sắp xếp tôpô

Thuật toán có thể mô tả vắn tắt như sau:

- Thực hiện DFS(*G*), khi mỗi đỉnh được duyệt xong ta đưa nó vào đầu danh sách liên kết (điều đó có nghĩa là những đỉnh kết thúc thăm càng muộn sẽ càng ở gần đầu danh sách hơn).
- Danh sách liên kết thu được khi kết thúc DFS(G) sẽ cho ta thứ tự cần tìm.

Thuật toán sắp xếp tôpô

TopoSort(G)

1. **for** $u \in V$ color[u] = white; // khởi tạo
2. L = new(linked_list); // khởi tạo danh sách liên kết rỗng L
3. **for** $u \in V$
4. if (color[u] == white) TopVisit(u);
5. **return** L; // L cho thứ tự cần tìm

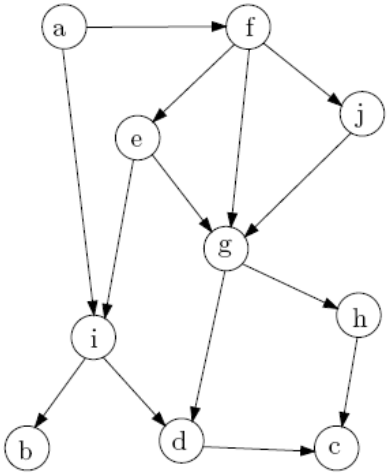
TopVisit(u) { // Bắt đầu tìm kiếm từ u

1. color[u] = gray; // Đánh dấu u là đã thăm
2. **for** $v \in$ Adj(u)
3. **if** (color[v] == white) TopVisit(v);
4. Nạp u vào đầu danh sách L // u đã duyệt xong

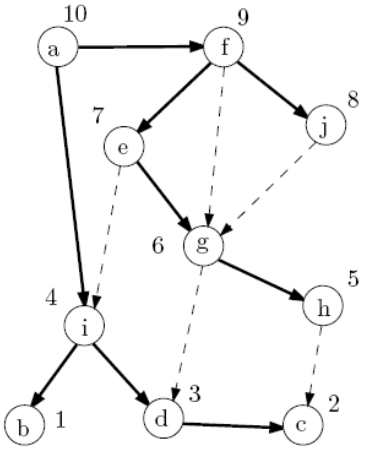
}

Thời gian tính của TopoSort(G) là $O(|V|+|E|)$.

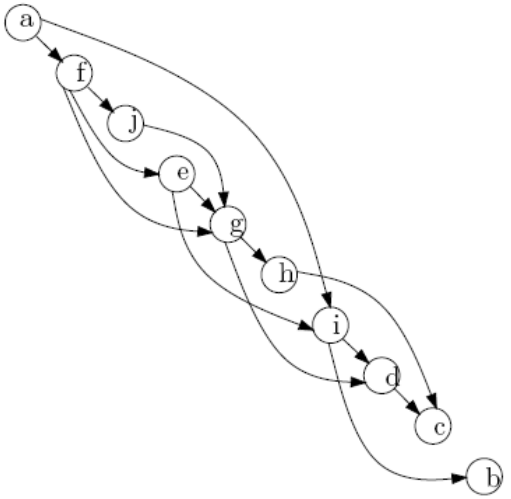
Ví dụ



Đồ thị G



DFS(G)



Thứ tự tôpô

Thuật toán xoá dần đỉnh

- Một thuật toán khác để thực hiện sắp xếp tôpô được xây dựng dựa trên mệnh đề sau
- **Mệnh đề.** Giả sử *G* là đồ thị có hướng không có chu trình. Khi đó

  1) Mọi đồ thị con *H* của *G* đều là đồ thị phi chu trình.

  2) Bao giờ cũng tìm được đỉnh có bán bậc vào bằng 0.

Thuật toán xoá dần đỉnh

- Từ mệnh đề ta suy ra thuật toán xoá dần đỉnh để thực hiện sắp xếp tôpô sau đây:
  - Thoạt tiên, tìm các đỉnh có bán bậc vào bằng 0. Rõ ràng ta có thể đánh số chúng theo một thứ tự tuỳ ý bắt đầu từ 1.
  - Tiếp theo, loại bỏ khỏi đồ thị những đỉnh đã được đánh số cùng các cung đi ra khỏi chúng, ta thu được đồ thị mới cũng không có chu trình, và thủ tục được lặp lại với đồ thị mới này.
  - Quá trình đó sẽ được tiếp tục cho đến khi tất cả các đỉnh của đồ thị được đánh số.

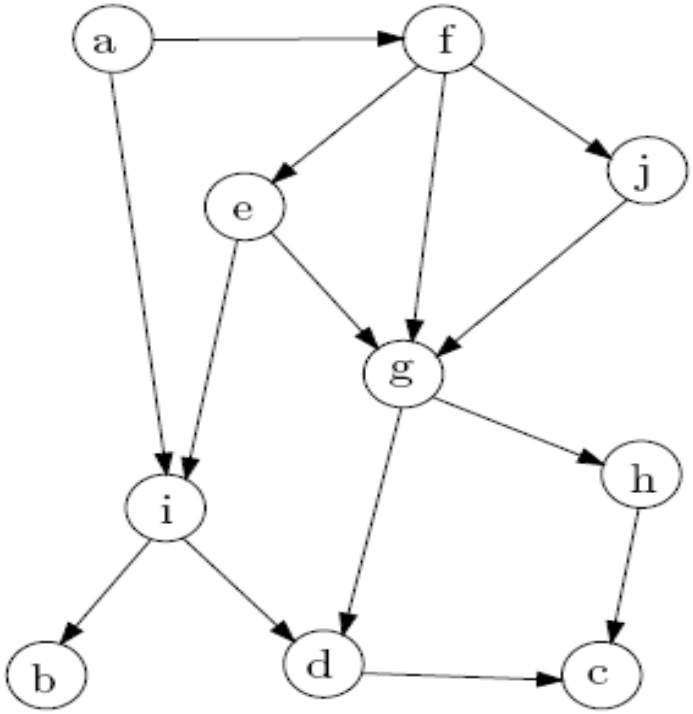
Thuật toán xoá dần đỉnh

for $v \in V$ **do**
    Tính Degree[v] – bán bậc vào của đỉnh v;
Q = hàng đợi chứa tất cả các đỉnh có bán bậc vào = 0;
num=0;
while $Q \neq \varnothing$ **do**
    v = dequeue(Q); num=num+1;
    Đánh số đỉnh v bởi num;
    for $u \in$ Adj(v) **do**
        Degree[u]=Degree[u] -1;
        if Degree[u]==0
            Enqueue(Q,u);

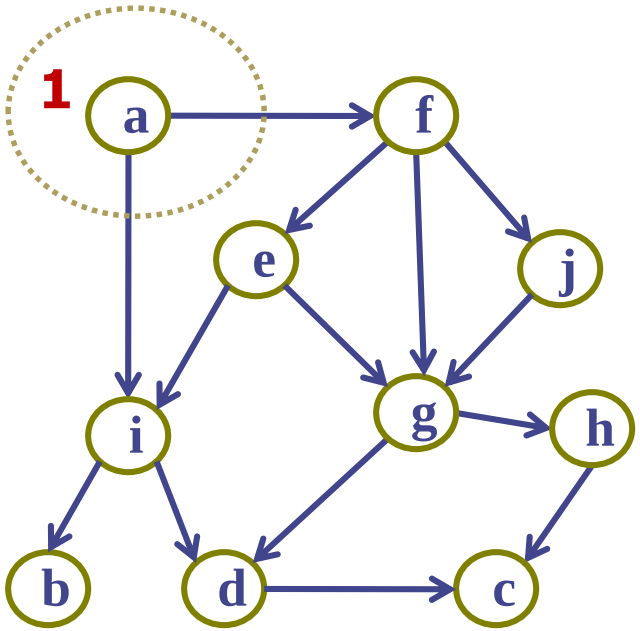
Thời gian tính: $\Theta(|V|+|E|)$

Thuật toán xoá dần đỉnh

Ví dụ. Thực hiện thuật toán xoá dần đỉnh đối với đồ thị



Thuật toán xoá dần đỉnh

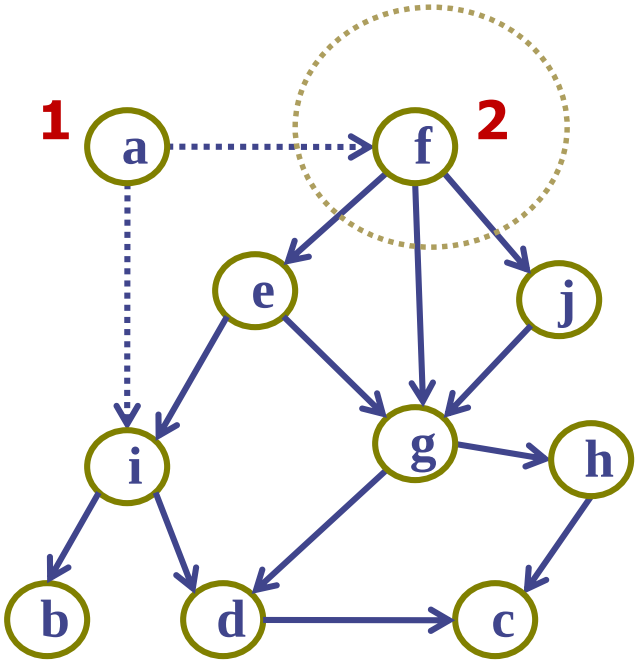


Đỉnh a có deg-(a)=0

Đánh số a bởi 1

Xoá các cung đi ra khỏi a

Thuật toán xoá dần đỉnh

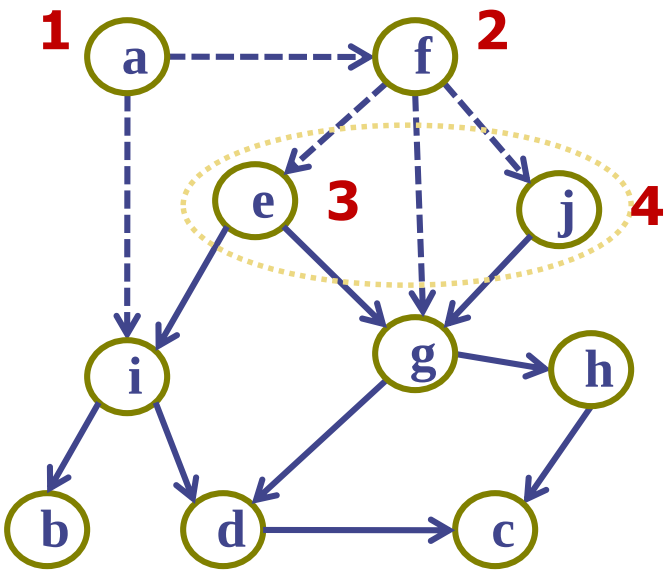


Đỉnh f có deg-(f)=0

Đánh số f bởi 2

Xoá các cung đi ra khỏi f

Thuật toán xoá dần đỉnh



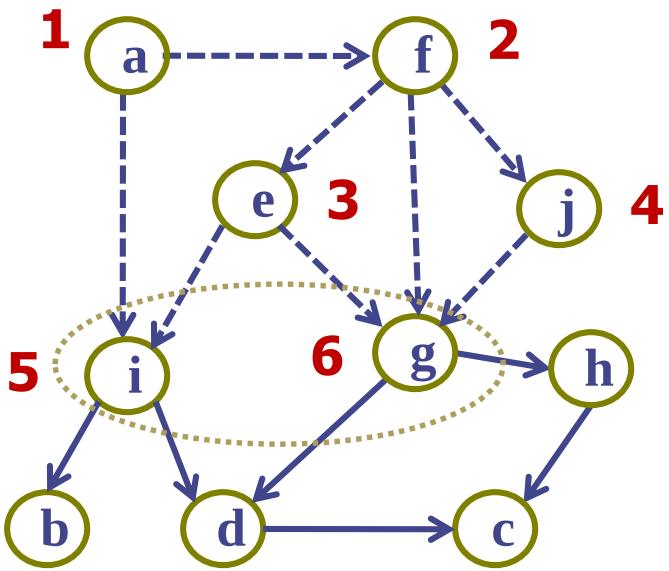
Đỉnh e và j có deg-(e)= deg-(j) = 0

Đánh số e và j theo thứ tự tuỳ ý

bởi 3 và 4

Xoá các cung đi ra khỏi e và j

Thuật toán xoá dần đỉnh



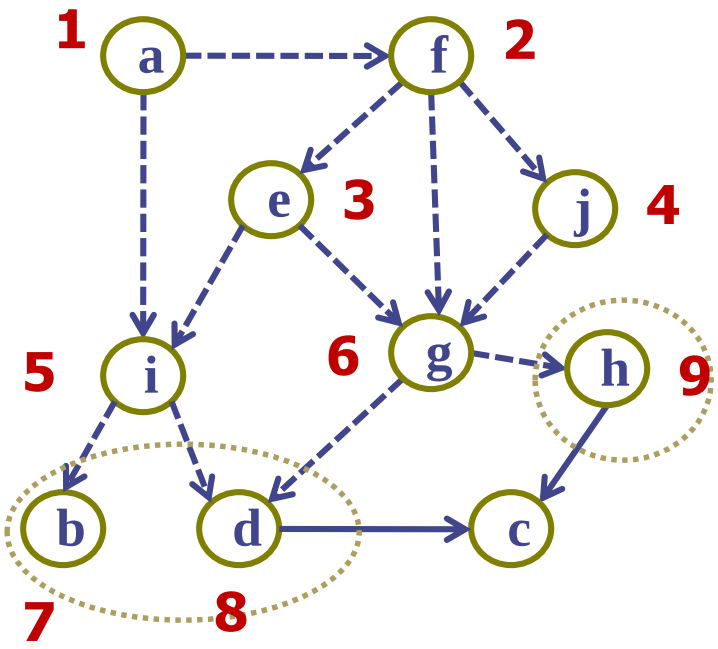
Đỉnh i và g có deg-(i)= deg-(g) = 0

Đánh số i và g theo thứ tự tuỳ ý

bởi 5 và 6

Xoá các cung đi ra khỏi i và g

Thuật toán xoá dần đỉnh

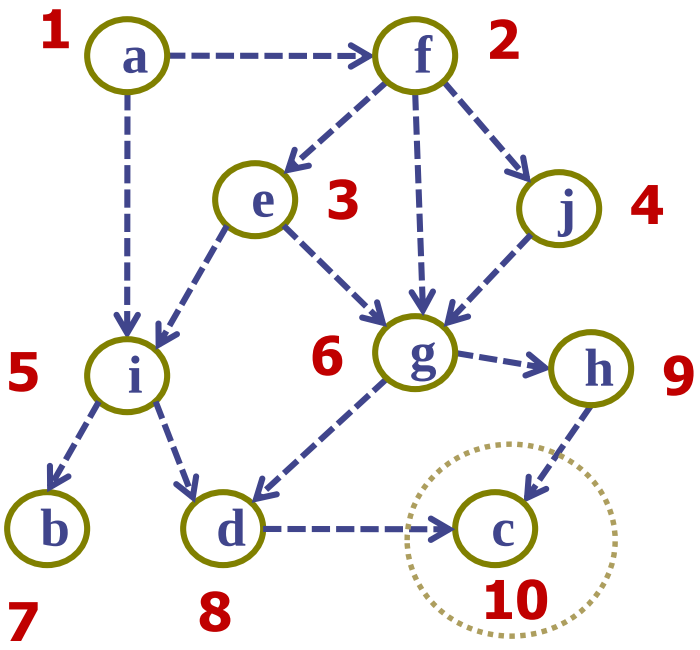


Đỉnh b, d và h có deg-(b)= deg-(d) = deg(h) = 0

Đánh số b, d và h theo thứ tự tuỳ ý bởi 7, 8 và 4

Xoá các cung đi ra khỏi b, d và h

Thuật toán xoá dần đỉnh



Đỉnh c có deg-(c)= 0

Đánh số c bởi 10

Thuật toán kết thúc

BAO ĐÓNG TRUYỀN ỨNG

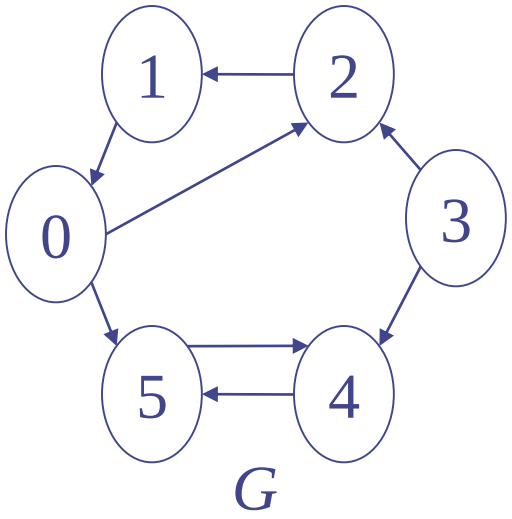
Transitive Closure

Transitive Closure

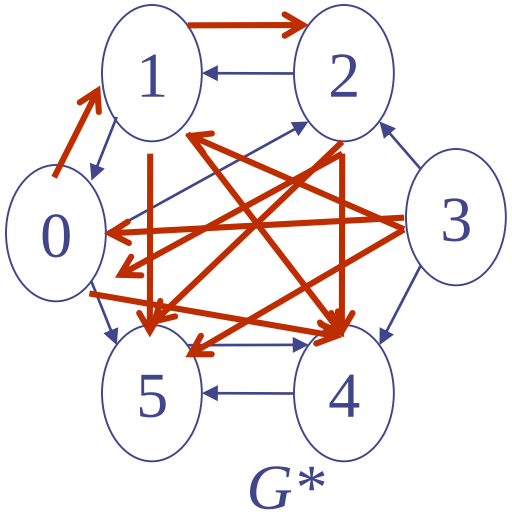
- **Định nghĩa.** Bao đóng truyền ứng của đồ thị có hướng $G=(V,E)$ là đồ thị có hướng $G^*=(V,E^*)$ với tập đỉnh là tập đỉnh của đồ thị G và tập cạnh

$$E^* = \{(u,v) | \text{ có đường đi từ } u \text{ đến } v \text{ trên } G\}$$

Bài toán: Cho đồ thị có hướng G, tìm bao đóng truyền ứng G^*



G



G^*

Nhân ma trận Bun

Boolean Matrix multiplication

- Ma trận Bun – là ma trận có tất cả phần tử là 0 hoặc 1
- Để tính tích của A và B ta thay thế
  - phép toán logic *and* vào chỗ phép toán số học *
  - phép toán logic *or* vào chỗ phép toán số học +
- Cho 2 ma trận kích thước $|V|.|V|$
  - Với mỗi *s* và *t*, tính tích vô hướng của dòng *s* của ma trận thứ nhất với cột *t* của ma trận thứ hai.

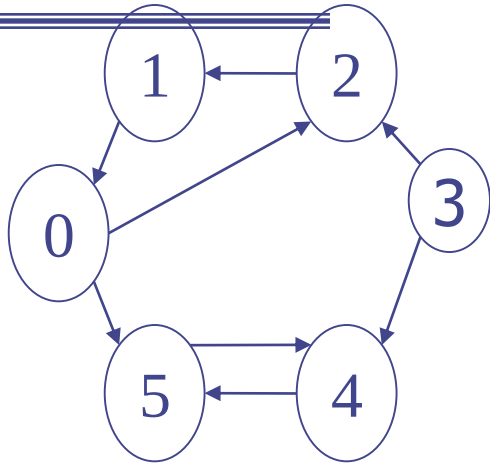
Matrix Multiplication Implementation

- C = A*B (số)

```
for (s = 0; s < V; s++)
   for (t = 0; t < V; t++)
      for (i = 0, C[s][t] = 0; i < V; i++)
         C[s][t] += A[s][i] * B[i][t];
```

- C = A*B (bun)

```
for (s = 0; s < V; s++)
   for (t = 0; t < V; t++)
      for (i = 0, C[s][t] = 0; i < V; i++)
         if (A[s][i] && B[i][t])
            C[s][t] = 1
```



to *t*

from *s*

	0	1	2	3	4	5
0	1		1			1
1	1	1				
2		1	1			
3			1	1	1	
4					1	1
5					1	1

Sử dụng tích ma trận để tìm bao đóng truyền ứng

- Ta có thể tính bao đóng truyền ứng cho đồ thị có hướng bằng cách xây dựng ma trận kề A cho nó, bổ sung các số 1 vào đường chéo rồi tính luỹ thừa $A^{|V|}$
- CM
  - A^2 – với mỗi cặp s và t, ta ghi 1 vào ma trận tích C khi và chỉ khi có đường đi từ s đến i và đường đi từ i đến t trong A
    - Xét đến các đường đi độ dài 2
  - A^3 – xét đến các đường đi độ dài 3
  - Không cần xét đến các đường đi độ dài lớn hơn $|V|$ bởi vì đường đi đơn trên đồ thị có độ dài lớn nhất là $|V|$

Thời gian tính

- Q: Thuật toán tìm bao đóng truyền ứng vừa mô tả đòi hỏi thời gian bao nhiêu ?
- Ans: $|V|^4$

- Liệu có thể tính A^2 tiếp đến A^4 … A^t với $t \geq |V|$ để cải thiện thời gian tính?
- Ans: $|V|^3 \log |V|$

Thuật toán Warshall

```
// n = |V|, Các đỉnh đánh số từ 0 đến n-1
for (i = 0; i < n; i++)
   for (s = 0; s < n; s++)
      for (t=0; t < n; t++)
         if (A[s][i] && A[i][t])
            A[s][t] = 1;
```

- **Mệnh đề.** Thuật toán tìm được bao đóng truyền ứng với thời gian $O(|V|^3)$.
- **CM:** Ta chứng minh thuật toán tìm được bao đóng truyền ứng bằng qui nạp.
  - Lần lặp 1: Ma trận có 1 ở vị trí (*s*,*t*) ***iff*** có đường đi *s*-*t* hoặc *s*-0-*t*
  - Lần lặp thứ *i*: Gán phần tử ở vị trí (*s*,*t*) giá trị 1 ***iff*** có đường đi từ *s* đến *t* trong đồ thị không chứa đỉnh với chỉ số lớn hơn *i* (ngoại trừ hai mút)
  - Lần lặp thứ *i*+1
    - Nếu có đường đi từ s đến t không chứa đỉnh có chỉ số lớn hơn i – A[s,t] đã có giá trị 1
    - Nếu có đường đi từ s đến i+1 và đường đi từ i+1 đến t, và cả hai đều không chứa đỉnh với chỉ số lớn hơn i (ngoại trừ hai mút) thì A[s,t] được gán giá trị 1

Thuật toán Warshall cải tiến

Ta có thể cải tiến thuật toán bằng cách bổ sung thêm câu lệnh if trước vòng lặp for trong cùng

```
// n = |V|, Các đỉnh đánh số từ 0 đến n-1
for (i = 0; i < n; i++)
   for (s = 0; s < n; s++)
      if A[s][i]
         for (t=0; t < n; t++)
            if A[i][t]
               A[s][t] = 1;
```

Cải tiến này chỉ có tác dụng tăng hiệu quả thực tế của thuật toán, mà không thay đổi được đánh giá thời gian tính trong tình huống tồi nhất của thuật toán

Áp dụng DFS tìm bao đóng truyền ứng

- **Mệnh đề.** Sử dụng DFS ta có thể xác định bao đóng truyền ứng sau thời gian $O(|V|*(|E|+|V|))$
- **Chứng minh**
  - DFS cho phép xác định tất cả các đỉnh đạt đến được từ một đỉnh cho trước v sau thời gian $O(|E|+|V|)$ nếu ta sử dụng biểu diễn đồ thị bởi danh sách kề
  - Do đó để xác định bao đóng truyền ứng ta thực hiện DFS với mỗi $v \in V$ ($|V|$ lần).
  - Thời gian tính: $O(|V|*(|E|+|V|))$.

Kinh nghiệm tính toán

đồ thị thưa (|E|=10|V|)					đồ thị dày (250 đỉnh)				
V	W	W*	A	L	E	W	W*	A	L
25	0	0	1	0	5000	289	203	177	23
50	3	1	2	1	10000	300	214	184	38
125	35	24	23	4	25000	309	226	200	97
250	275	181	178	13	50000	315	232	218	337
500	2222	1438	1481	54	100000	326	246	235	784

W	Thuật toán Warshall
W*	Thuật toán Warshall cải tiến
A	DFS sử dụng ma trận kề
L	DFS sử dụng danh sách kề

Minh hoạ cài đặt trên C

Cấu trúc dữ liệu:

```
struct ListIt {
   int vertex;              // Ghi đỉnh kề
   int type;                // Ghi nhận phân loại cạnh
   ListIt *p_next;          // con trỏ
};
typedef ListIt* pListIt;

pListIt* adjacencyList;  // Mảng danh sách kề
```

Ứng dụng DFS

- Kiểm tra đồ thị có chứa chu trình?
- Phân loại cạnh
- Sắp xếp tô pô (đối với đồ thị có hướng không có chu trình)

- Cài đặt: Xem file "DFS_ApplPart1.CPP"

Các thủ tục

1) Thêm vào cạnh (dest, val)

```
bool add(pListIt &dest, int val);
```

2) Đọc dữ liệu vào từ file văn bản

```
void read();
```

3) Tìm kiếm theo chiều sâu bắt đầu từ đỉnh vertex

```
void DFS(int vertex);
```

4) In ra biểu diễn danh sách kề của đồ thị

```
void print();
```

5) Xây dựng danh sách kề ngược

```
void inverse(pListIt*& from, pListIt*&to);
```

Xác định các hằng

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <conio.h>

#define TREE_EDGE  1
#define BACK_EDGE 2
#define CROSS_EDGE 3
#define FORWARD_EDGE 4
#define INT_MAX 999999999

#define WHITE 0
#define GRAY 1
#define BLACK 2
```

Khai báo biến

```
int n, m, callCount, at, atPos = 0;
int* parent;
int* color;
int* distance;
int* topological_order;
bool isAcyclic;
//  A structure for the element point
struct ListIt {
   int vertex;
   int type;
   ListIt *p_next; };
// typedef for the pointer type
typedef ListIt* pListIt;
pListIt* adjacencyList; // Graph representation by Adj List
pListIt* inverseList;   // Graph representation by reverse Adj List
```

Các chương trình con

- `bool add(pListIt &dest, int val);`
- `void read();`
- `void DFS(int vertex);`
- `void DFS_inver(int vertex);`
- `void print();`
- `void inverse(pListIt*& from, pListIt*&to);`

Thêm vào một cạnh

```
// Ta muốn các đỉnh trong danh sách kề
// được sắp xếp theo thứ tự tăng dần của chỉ số
bool add(pListIt &dest, int val){
   //create the item
   pListIt p;
   p = (pListIt) malloc(sizeof(ListIt));
   p -> vertex = val;
   p->type = 0;
   if(!dest)  // first item addition
   {    p -> p_next = NULL;
        dest = p;
   } else
   {   pListIt find = dest;
        pListIt at = NULL;
```

Thêm vào một cạnh

```
        // Find the first greater number, insert before
        while(find && find->vertex <= val){
           if( find->vertex == val) // do not add a duplicate
           {  free(p);
              return false; }
            at = find;
            find = find->p_next; }
        // insert at
        if (at) // insert at a valid point
        {   p->p_next = at->p_next;
            at->p_next = p;
        } else // insert at the start
        {  p->p_next = dest;
           dest = p; }
   } return true;
}
```

Đọc dữ liệu, khởi tạo biến

```
void read() {
   // make the file opens
   freopen("TOPOORDER.in","r",stdin);
   scanf("%d",&n);   // read in the number of vertices
   // according to this allocate the required memory
   adjacencyList = (pListIt*)
   calloc(n+1,sizeof(pListIt));
   color = (int*)calloc(n+1, sizeof(int));
   parent  = (int*)calloc(n+1, sizeof(int));
   distance  = (int*)calloc(n+1, sizeof(int));
   topological_order = (int*)calloc(n+1, sizeof(int));
```

Đọc dữ liệu, khởi tạo biến

```
    // make the read in
    int i, from, to;
    m = 0;
    for (i = 1; ; i++) {
        scanf("%d %d",&from,&to); // from -> to
        if(feof(stdin)) break;
        add(adjacencyList[from], to);
        ++m;
        //add(adjacencyList[to], from);
        // this statment is need for the undirected graph
    }
}
```

DFS từ một đỉnh

```
void DFS(int vertex) {
   //printf(" %d ", vertex);
   pListIt p;
   color[vertex] = GRAY; // visited
   for (p = adjacencyList[vertex]; p != NULL; p = p ->
   p_next)
       if (!color[p -> vertex]) {
              parent[p->vertex] = vertex;
              distance[p->vertex] = distance[vertex] +1;
              p->type = TREE_EDGE;
              DFS(p -> vertex);
       }
```

DFS từ một đỉnh

```
      else {
          if( color[p->vertex] == GRAY)
          {  p->type = BACK_EDGE;
             if (isAcyclic) {
                 isAcyclic = false;
                 // print the cycle
                 printf("\n \n Graph contains the
following cycle: ");
                  int l = vertex;
                  while(l != p->vertex) {
                       printf( " %d ", l);
                       l = parent[l];
                  } printf( " %d ", l);
             }
          }
```

DFS từ một đỉnh

```
        else
            if (distance[p->vertex] > distance[vertex])
            {
                p->type = FORWARD_EDGE ;
            }
            else
                p->type = CROSS_EDGE ;
        }
        color[vertex] = BLACK;
         // add to the topological order
        ++atPos;
        topological_order[n-atPos] = vertex;
}
```

PrintGraph

```
void printGraph() {
    int vertex;
   pListIt p;
    printf("\n\n-----------------------------------");
    printf("                ADJ LIST: \n");
    for (vertex=1; vertex<=n; vertex++){
        printf("Adj[%d] ->  ", vertex);
           for (p = adjacencyList[vertex]; p != NULL; p =
   p -> p_next)
      {    printf(" %d ", p->vertex); }
      printf("\n");
   }
    printf("\n    Press any key to continue...");
   getch();
    printf("\n-----------------------------------\n");
}
```

Main program

```
int main() {
   int i;
   isAcyclic = true;
   read();
   printGraph();
   at = 0;
   memset(color, WHITE, (n+1)*sizeof(int));
   memset(parent, 0, (n+1)*sizeof(int));
   memset(topological_order, 0, (n+1)*sizeof(int));
```

Main program (cont)

```
// DFS(G)
    for (i = 1; i <= n; i++)  // until fully traveled repeat the DFS
      if (!color[i]) {
        callCount++;      // for counting the number of connected component
                          // in undirected graph
        DFS(i);
      }
    if ( isAcyclic)    {
       printf(" \n\nGraph is acyclic.");
       printf(" \n\nTopological Order: \n");
       for ( i = 0; i != n; ++i)
              printf(" %d" , topological_order[i]);
    }
```

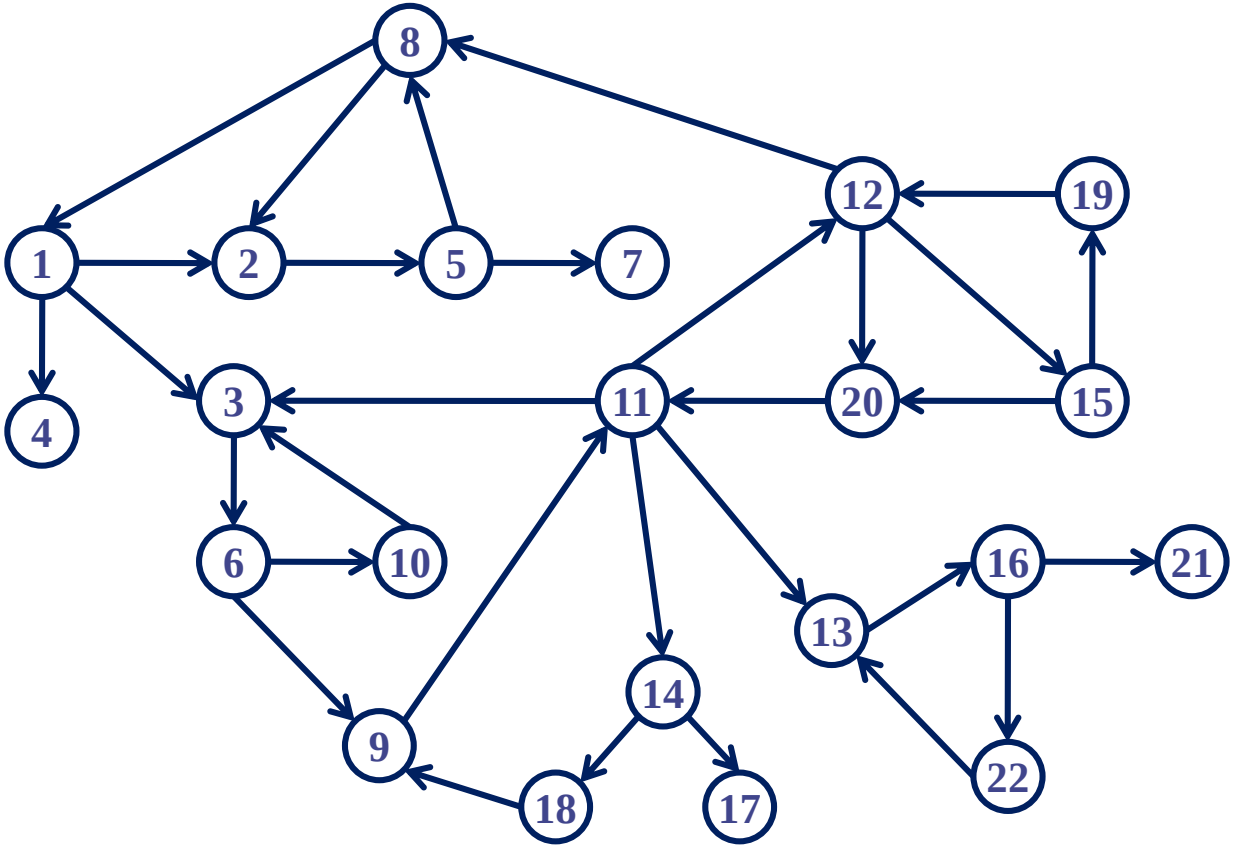
Main program (cont)

```
// Classifying the edges
    printf("\n\n Tree edges: \n");
    pListIt p;
    for(i = 1; i <= n; ++i)
            for (p = adjacencyList[i]; p; p = p->p_next) {
               if (p->type == TREE_EDGE) {
                    printf(" %d - %d ; ", i, p->vertex);   }
            }
    printf(" \n\n Back edges: \n");
    for(i = 1; i <= n; ++i)
            for (p = adjacencyList[i]; p; p = p->p_next){
                        if (p->type == BACK_EDGE){
                           printf(" %d - %d ; ", i, p->vertex); }
            }
```

Main program (cont)

```
    printf("\n\n Forward edges: \n");
    for(i = 1; i <= n; ++i)
        for (p = adjacencyList[i]; p; p = p->p_next) {
            if (p->type == FORWARD_EDGE){
                printf(" %d - %d ;", i, p->vertex);}
        }
    printf("\n\n Cross edges: \n");
    for(i = 1; i <= n; ++i)
        for (p = adjacencyList[i]; p; p = p->p_next) {
            if (p->type == CROSS_EDGE){
                printf(" %d - %d ; ", i, p->vertex);}
        }
  printf("\n     Press any key to stop program...");
  getch();  return 0;
}
```

Đồ thị có hướng minh hoạ cho ứng dụng của DFS



Chuẩn bị file dữ liệu vào

- Dòng 1: n - số lượng đỉnh
- Các dòng tiếp theo, mỗi dòng ghi đỉnh đầu và đỉnh cuối của một cạnh: d_i c_i , $i = 1, 2, ..., m$

```
22
1 2
1 3
1 4
2 5
5 7
5 8
3 6
6 9
6 10
8 2
8 1
9 11
10 3
11 12
11 13
11 14
11 3
12 8
12 15
13 16
12 20
14 17
14 18
15 19
15 20
16 21
16 22
18 9
19 12
20 11
22 13
```

Danh sách kề và kết quả

1 -> 2 3 4
2 -> 5
3 -> 6
4 ->
5 -> 7 8
6 -> 9 10
7 ->
8 -> 1 2
9 -> 11
10 -> 3
11 -> 3 12 13 14

12 -> 8 15 20
13 -> 16
14 -> 17 18
15 -> 19 20
16 -> 21 22
17 ->
18 -> 9
19 -> 12
20 -> 11
21 ->
22 -> 13

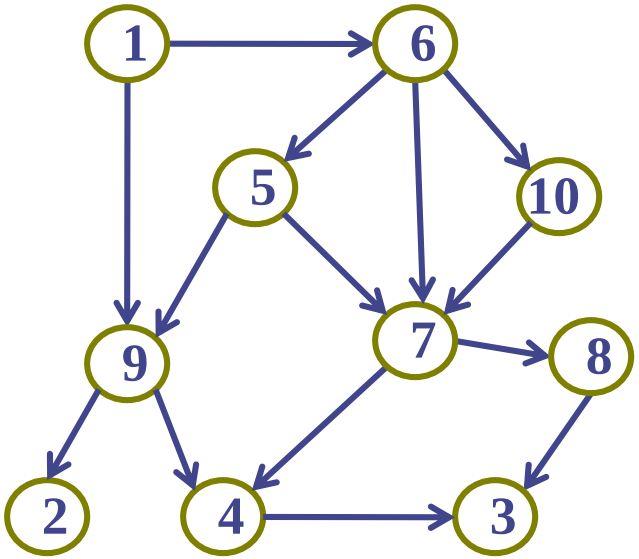
Kết quả:

+ Đồ thị có chứa chu trình

+ Đưa ra phân loại các cạnh

Ví dụ: Sắp xếp tôpô

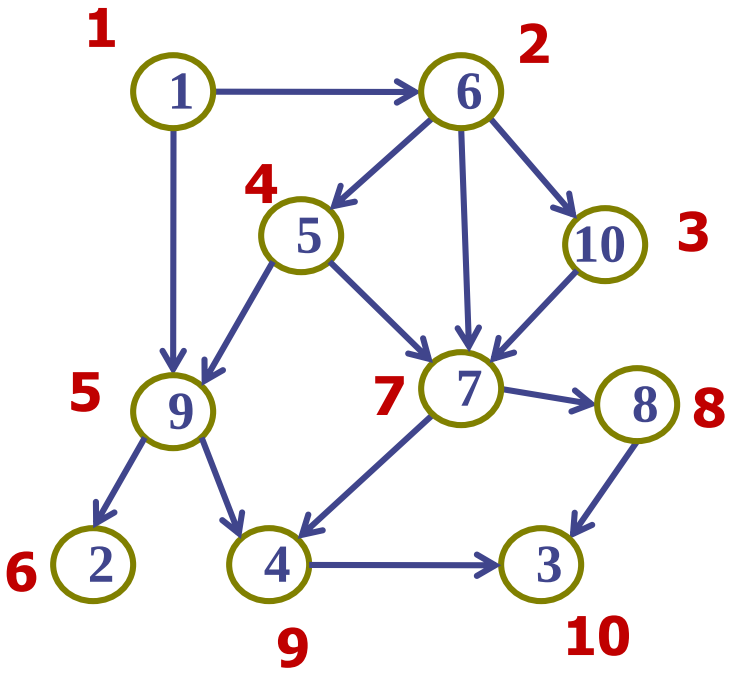
- Thứ tự tô pô:



Ví dụ: Sắp xếp tôpô

Kết quả:

Thứ tự:	**1**	**6**	**10**	**5**	**9**	**2**	**7**	**8**	**4**	**3**
Cách đánh số mới:	1	2	3	4	5	6	7	8	9	10



NỘI DUNG

7.1. Đồ thị

7.2. Biểu diễn đồ thị

7.3. Các thuật toán duyệt đồ thị

7.4. Một số ứng dụng của tìm kiếm trên đồ thị

7.5. Bài toán cây khung nhỏ nhất

7.6. Bài toán đường đi ngắn nhất

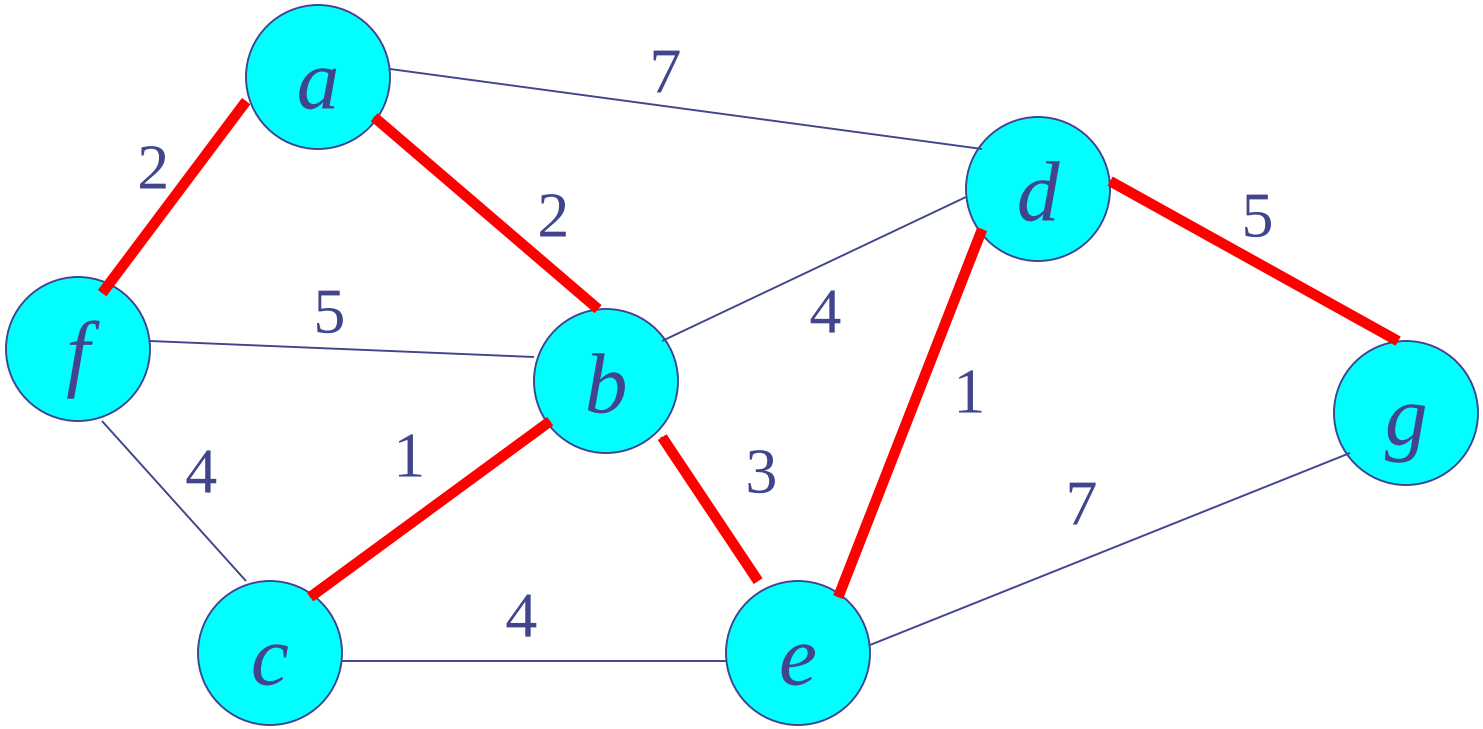
7.5. Bài toán cây khung nhỏ nhất

- Phát biểu bài toán
- Thuật toán Kruskal
- Cài đặt thuật toán Kruskal với cấu trúc dữ liệu các tập không giao nhau
- Thuật toán Prim
- Cài đặt thuật toán Prim với cấu trúc dữ liệu đống

Phát biểu bài toán

- Giả sử $G = (V, E)$ là đồ thị vô hướng liên thông có trọng số trên cạnh $c(e)$, $e \in E$.
- **Định nghĩa.** Cây $T = (V, E_T)$ với $E_T \subseteq E$, được gọi là cây khung của G. Độ dài của cây khung T là tổng trọng số trên các cạnh của nó:
- Bài toán đặt ra là tìm cây khung có độ dài nhỏ nhất.

Ví dụ



- Cây khung nhỏ nhất có độ dài 14

Thuật toán Kruskal

- Thuật toán sẽ xây dựng tập cạnh E_T của cây khung nhỏ nhất $T = (V, E_T)$ theo từng bước.
- Trước hết sắp xếp các cạnh của đồ thị G theo thứ tự không giảm của độ dài.
- Bắt đầu từ tập $E_T = \varnothing$, ở mỗi bước ta sẽ lần lượt duyệt trong danh sách cạnh đã sắp xếp, từ cạnh có độ dài nhỏ đến cạnh có độ dài lớn hơn, để tìm ra cạnh mà việc bổ sung nó vào tập E_T không tạo thành chu trình trong tập này.
- Thuật toán sẽ kết thúc khi ta thu được tập E_T gồm n-1 cạnh.

Thuật toán Kruskal

Kruskal_Algorithm

$E_T := \varnothing$;

while $|E_T| < (n-1)$ and $(E \neq \varnothing)$ **do**

{

Chọn e là cạnh có độ dài nhỏ nhất trong E;

$E := E \setminus \{e\}$;

if ($E_T \cup \{e\}$ *không chứa chu trình*) **then** $E_T := E_T \cup \{e\}$;

}

if ($|E_T| < n-1$) **then** *Đồ thị không liên thông;*

Cài đặt thuật toán Kruskal

- Có 2 thao tác đòi hỏi nhiều tính toán nhất trong 1 bước lặp của thuật toán Kruskal:
  - Chọn e là cạnh có độ dài nhỏ nhất trong E;
  - Kiểm tra xem tập cạnh $E_T \cup \{e\}$ có chứa chu trình hay không?

Chọn *e* là cạnh có độ dài nhỏ nhất trong *E*

- Ta sẽ thực hiện trước việc sắp xếp các cạnh của đồ thị theo thứ tự không giảm của độ dài. Đối với đồ thị có m cạnh, bước này đòi hỏi thời gian $O(m \log m)$. Khi đó trong bước lặp việc chọn cạnh có độ dài nhỏ nhất đòi hỏi thời gian $O(1)$.
- Tuy nhiên, để xây dựng cây khung nhỏ nhất với n-1 cạnh, nói chung, thường ta chỉ phải xét $p < m$ cạnh. Do đó thay vì sắp xếp toàn bộ dãy cạnh ta sẽ sử dụng heap-min:
  - Để tạo đống đầu tiên ta mất thời gian $O(m)$,
  - Việc vun lại đống sau khi lấy ra phần tử nhỏ nhất ở gốc đòi hỏi thời gian $O(\log m)$.
  - Suy ra thuật toán sẽ đòi hỏi thời gian $O(m+p \log m)$ cho việc sắp xếp các cạnh. Trong việc giải các bài toán thực tế, số p thường nhỏ hơn rất nhiều so với m.

Kiểm tra: Tập $E_T \cup \{e\}$ có chứa chu trình hay không?

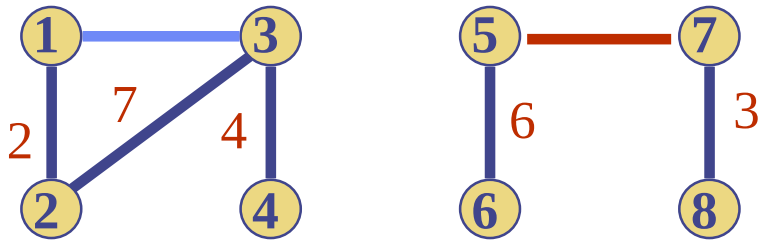
- Ký hiệu $E^* = E_T \cup \{e\}$
- Việc này có thể thực hiện nhờ sử dụng thuật toán kiểm tra xem đồ thị $G^*=(V,E^*)$ có chứa chu trình hay không đã trình bày trong mục trước. Thời gian cần thiết là $O(n)$, trong đó $n = |V|$.
- Với những đề xuất vừa nêu ta thu được cài đặt thuật toán Kruskal với thời gian

$$O(m+m \log m) + O(n.m) = O(nm + m \log m)$$

Chú ý: Có cách thực hiện khác dựa trên ***cấu trúc dữ liệu các tập không giao nhau*** để thực hiện thao tác kiểm tra tập $E_T \cup \{e\}$ có chứa chu trình hay không?

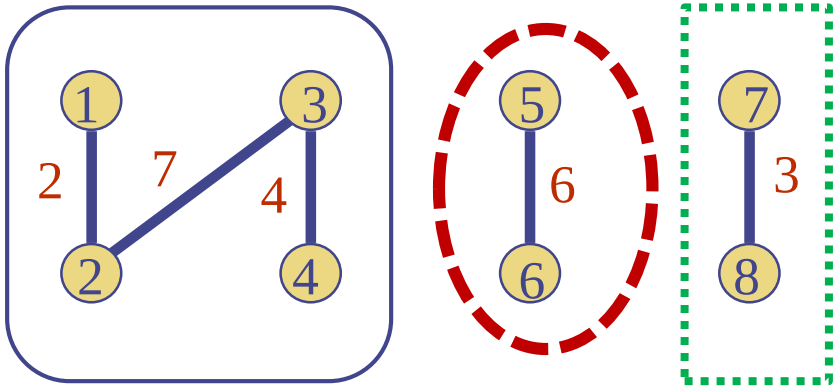
Cấu trúc dữ liệu cho thuật toán Kruskal

- Bổ sung cạnh (u, v) vào E_T có tạo thành chu trình?



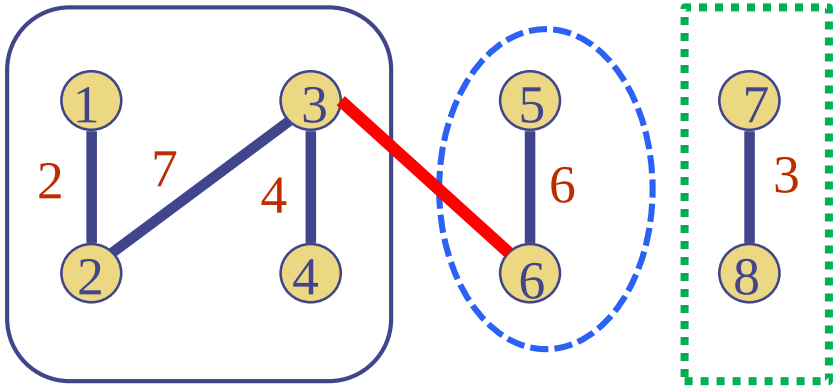
- Mỗi thành phần của T (đang xây dựng) là một cây.
- Khi u và v thuộc cùng một thành phần liên thông thì việc bổ sung (u,v) sẽ tạo thành chu trình.
- Khi u và v thuộc các thành phần liên thông khác nhau thì việc bổ sung (u,v) sẽ không tạo thành chu trình.

Cấu trúc dữ liệu cho thuật toán Kruskal



- Mỗi tplt của *T* được xác định bởi các đỉnh trong nó.
- Biểu diễn mỗi tplt bởi tập các đỉnh thuộc nó.
  - {1, 2, 3, 4}, {5, 6}, {7, 8}
- Hai đỉnh thuộc cùng một tplt khi và chỉ khi chúng thuộc cùng một tập đỉnh.

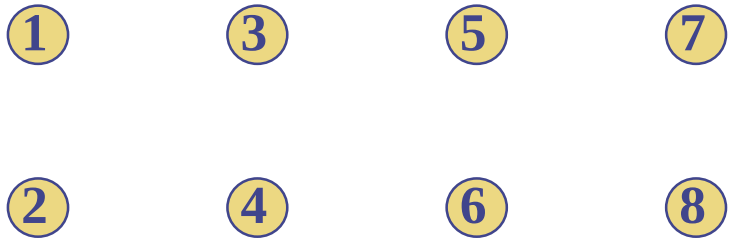
Cấu trúc dữ liệu cho thuật toán Kruskal



- Khi cạnh (u, v) được bổ sung vào T, hai thành phần chứa u và v sẽ được nối lại thành một tplt.
  - Trong cách biểu diễn các tplt dưới dạng tập hợp, tập con chứa u và tập con chứa v sẽ được hợp lại thành một tập.
    - $\{1, 2, 3, 4\} \cup \{5, 6\} \rightarrow \{1, 2, 3, 4, 5, 6\}$

Cấu trúc dữ liệu cho thuật toán Kruskal

- Thoạt tiên, E_T là rỗng. Có $|V|$ tplt, mỗi thành phần gồm 1đỉnh.



- Các tập khởi tạo là:
  - {1} {2} {3} {4} {5} {6} {7} {8}
- Nếu việc bổ sung cạnh (u, v) vào E_T không tạo thành chu trình thì cạnh này được bổ sung và E_T.

r_1 = find(u); r_2 = find(v); // r_1 (r_2) – tên của tập con chứa u (v)
if ($r_1 \neq r_2$) **then**
$E_T = E_T \cup (u,v)$;
union(r_1, r_2); // Nối hai tập con r_1, r_2

Cấu trúc dữ liệu các tập không giao nhau
(Disjoint-set Data Structures)

Vấn đề đặt ra là: Cho tập V gồm n phần tử, ta cần xây dựng cấu trúc dữ liệu biểu diễn phân hoạch tập V ra thành các tập con $V_1, V_2, \ldots, V_k$ hỗ trợ thực hiện hiệu quả các thao tác sau:

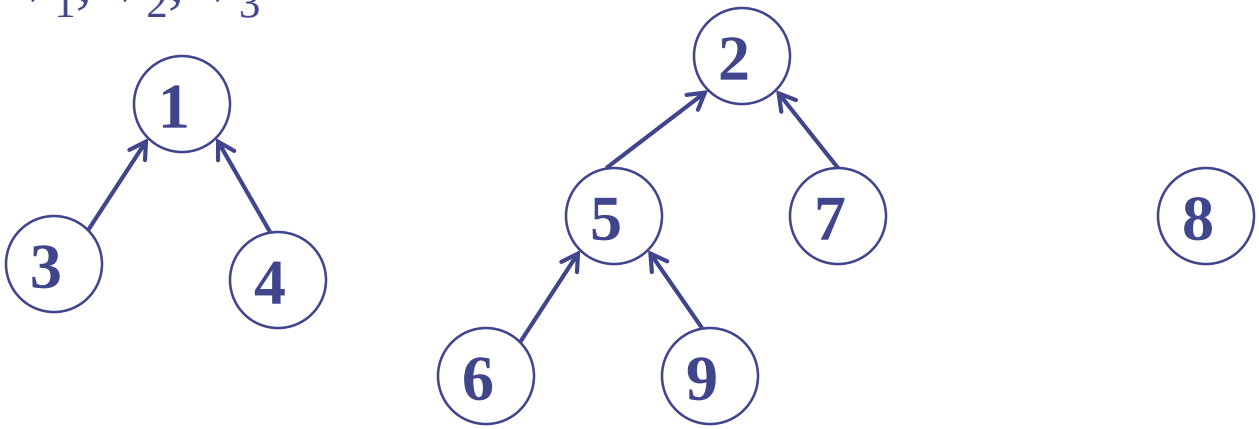
- **Makeset(x):** Tạo một tập con chứa duy nhất phần tử x.
- **Union(x, y):** Thay thế các tập V_i và V_j (trong đó $x \in V_i$ và $y \in V_j$) bởi tập $V_i \cup V_j$ trong phân hoạch đang xét.
- **Find(x):** Tìm tên $r(V_i)$ của tập V_i chứa phần tử x.

Như vậy, Find(x) và Find(y) trả lại cùng một giá trị khi và chỉ khi x và y thuộc cùng một tập con trong phân hoạch.

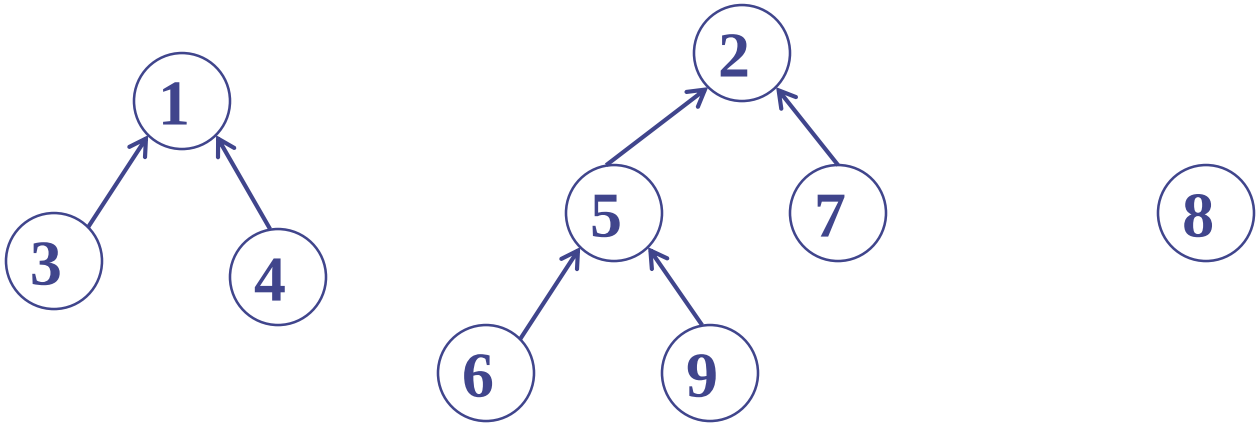
Cấu trúc dữ liệu đáp ứng yêu cầu này có tên là cấu trúc dữ liệu Union-Find (hoặc Disjoint-set data structure).

Cấu trúc dữ liệu các tập không giao nhau
(Disjoint-set Data Structures)

- Trước hết để biểu diễn mỗi tập con $X \subset V$, chúng ta sẽ sử dụng cấu trúc cây có gốc: Chọn một phần tử nào đó của X làm gốc (tên của tập con X chính là phần tử tương ứng với gốc), mỗi phần tử $x \in X$ sẽ có một biến trỏ parent[x] trỏ đến cha của nó, nếu x là gốc thì parent[x] = x.
- **Ví dụ:** Giả sử có $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. $V_1 = \{1, 3, 4\}$, $V_2 = \{2, 5, 6, 7, 9\}$, $V_3 = \{8\}$. Ta có ba cây mô tả ba tập V_1, V_2, V_3



Cấu trúc dữ liệu các tập không giao nhau
(Disjoint-set Data Structures)



- Mảng parent để biểu diễn rừng gồm 3 cây tương ứng với V_1, V_2, V_3:

v	1	2	3	4	5	6	7	8	9
parent[v]	1	2	1	1	2	5	2	8	5

Makeset(x) và FindSet

```
MakeSet(x) {
    parent[x] := x;
}
```

Thời gian: $O(1)$.

```
Find(x);
{
    while x ≠ parent[x] do x = parent[x];
    return  x;
}
```

Thời gian: $O(h)$, trong đó h là độ cao của cây chứa x.

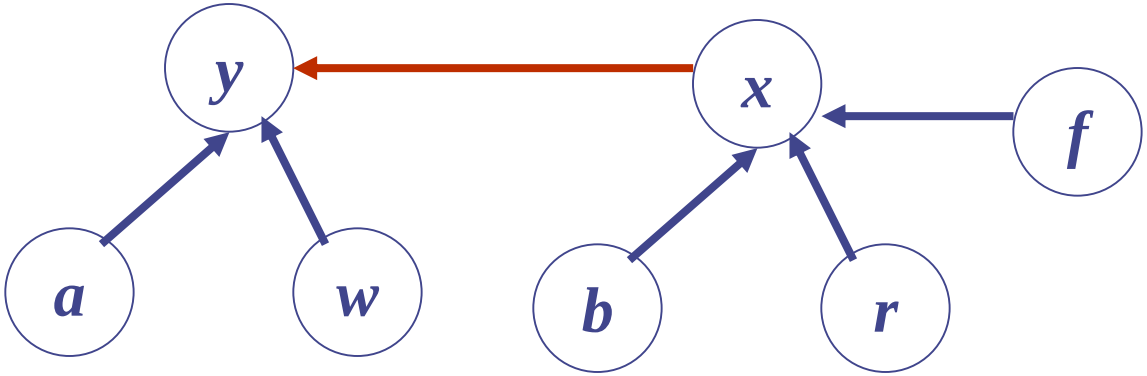
Cấu trúc dữ liệu các tập không giao nhau
(Disjoint-set Data Structures)

Để nối tập con chứa *x* và tập con chứa *y* chúng ta có thể chữa lại biến trỏ của gốc của cây chứa *x* để cho nó trỏ đến gốc của cây con chứa *y*. Điều đó được thực hiện nhờ thủ tục sau

```
Union(x, y) {
        u:= Find(x);  (* Tìm u là gốc của cây con chứa x *)
        v:= Find(y);  (* Tìm v gốc của cây con chứa y *)
        parent[u] := v;
}
```

Thời gian: *O*(*h*)

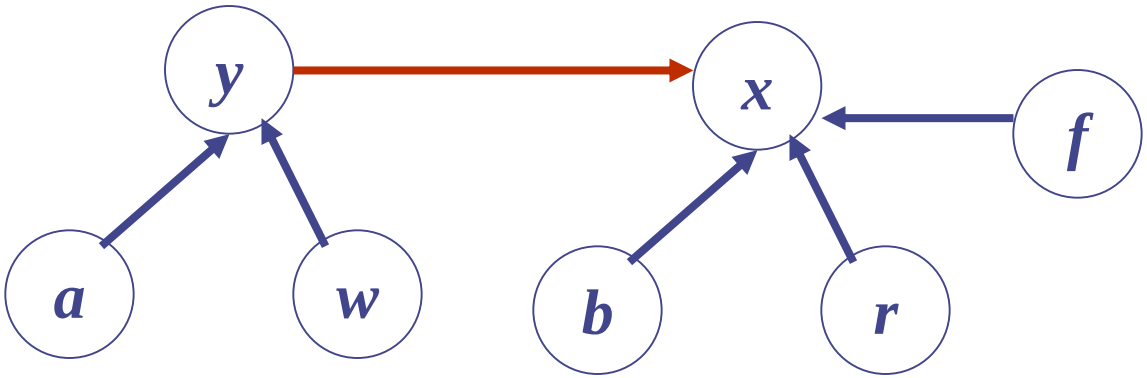
Ví dụ Union(x,y)



x trỏ đến *y*

b, *r* và *f*

chìm xuống sâu hơn



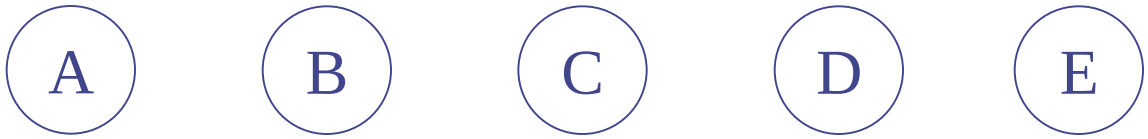
y trỏ đến *x*

a và *w*

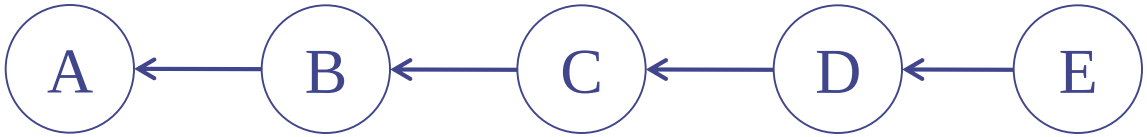
chìm xuống sâu hơn

Phân tích độ phức tạp

- Có thể thấy thời gian tính của hàm Find(x) phụ thuộc vào độ cao của cây chứa *x*. Trong trường hợp cây có *k* đỉnh và có dạng như một đường đi thì độ cao của cây sẽ là *k*- 1.
- ***Ví dụ:***



- Sau khi thực hiện Union(A,B); Union(B,C); Union(C,D); Union(D,E) có thể thu được cây



- **Do đó hàm Find(x) có đánh giá thời gian tính là *O*(*n*).**

Cấu trúc dữ liệu các tập không giao nhau

- Liệu có cách nào để giảm độ cao của các cây con?
- Có một cách thực hiện rất đơn giản: Khi nối hai cây chúng ta sẽ điều chỉnh con trỏ của gốc của cây con có ít đỉnh hơn, chứ không thực hiện việc nối một cách tuỳ tiện.
- Để ghi nhận số phần tử của một cây chúng ta sẽ sử dụng thêm biến Num[v] chứa số phần tử của cây con với gốc tại *v*.

MAKESET và Union cải tiến

```
MAKESET(x) {
    parent[x] := x;
    Num[x]:=1;
}
Union(x, y){
    u:= Find(x); // Tìm u là gốc của cây con chứa x
    v:= Find(y); // Tìm u là gốc của cây con chứa y
    if  Num[u] <= Num[v] {
            parent[u] := v;    Num[v]:= Num[u]+Num[v];
    }  else {
            parent[v] := u;  Num[u]:= Num[u]+Num[v];
           }
}
```

Cấu trúc dữ liệu các tập không giao nhau

- **Bổ đề.** Giả sử quá trình thực hiện nối cây bắt đầu từ các cây chỉ có 1 đỉnh. Khi đó độ cao của các cây xuất hiện khi thực hiện thủ tục nối không vượt quá log n.
- **CM.** Qui nạp theo số đỉnh của cây.

- Từ bổ đề suy ra các thao tác Find và Union được thực hiện với thời gian $O(\log n)$ nhờ sử dụng cách nối cây cải tiến.

Thuật toán Kruskal sử dụng cấu trúc dữ liệu Union-Find

Kruskal(G,w)

1. $E_T \leftarrow \varnothing$
2. **for** $v \in V$ **do**
3. Make-Set(v)
4. Sắp xếp các cạnh trong E theo thứ tự không giảm của trọng số
5. **for** $(u,v) \in E$ **do**
6. **if** Find(u) $\neq$ Find(v)
7. **then** $E_T \leftarrow E_T \cup \{(u,v)\}$
8. Union(u,v)
9. **return** E_T

Phân tích thời gian tính
Thuật toán Kruskal sử dụng cấu trúc dữ liệu Union-Find

- Dòng 1-3 (khởi tạo): $O(|V|)$
- Dòng 4 (sắp xếp): $O(|E| \log |E|)$
- Dòng 6-8 (các thao tác với phân hoạch): $O(|E| \log |E|)$
- Tổng cộng: $\boldsymbol{O(|E| \log |E|)}$

NỘI DUNG

7.1. Đồ thị

7.2. Biểu diễn đồ thị

7.3. Các thuật toán duyệt đồ thị

7.4. Một số ứng dụng của tìm kiếm trên đồ thị

7.5. Bài toán cây khung nhỏ nhất

7.6. Bài toán đường đi ngắn nhất

7.6. Bài toán đường đi ngắn nhất

- Phát biểu bài toán
- Thuật toán Dijkstra
- Cài đặt thuật toán với hàng đợi có ưu tiên
- Ứng dụng

Phát biểu bài toán

- **Định nghĩa.** Cho đồ thị có hướng $G = (V, E)$ với trọng số trên cạnh $c(e)$, $e \in E$. Giả sử $s, t \in V$ và $P(s, t)$ là đường đi từ s đến t trên đồ thị

  $P(s,t)$: $s = v_0, v_1, \ldots, v_{k-1}, v_k = t.$

- Ta gọi độ dài của đường đi $P(s,t)$ là tổng trọng số trên các cung của nó, tức là nếu ký hiệu độ dài này là $\rho(P(s,t))$, thì theo định nghĩa

$$\rho(P(s,t)) = \sum_{e \in P(s,t)} c(e) = \sum_{i=0}^{k-1} c(v_i, v_{i+1})$$

- Ta gọi đường đi ngắn nhất từ s đến t là đường đi có độ dài nhỏ nhất trong số tất cả các đường đi từ s đến t trên đồ thị.

- Người ta thường sử dụng ký hiệu $\delta(s,t)$ để chỉ độ dài của đường đi ngắn nhất từ s đến t,và gọi nó là khoảng cách từ s đến t.

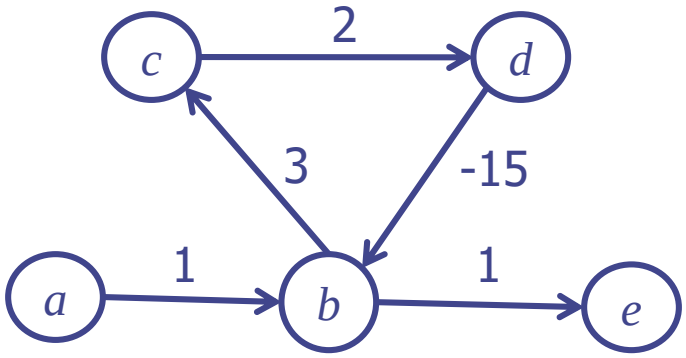
Các dạng bài toán ĐĐNN

- Có 3 dạng bài toán đường đi ngắn nhất cơ bản
  - Bài toán 1) Tìm đường đi ngắn nhất giữa 2 đỉnh cho trước.
  - Bài toán 2) Tìm đường đi ngắn nhất từ một đỉnh nguồn s đến tất cả các đỉnh còn lại.
  - Bài toán 3) Tìm đường đi ngắn nhất giữa hai đỉnh bất kì.
- Các bài toán được dẫn ra theo thứ tự từ đơn giản đến phức tạp hơn.
- Nếu ta có thuật toán để giải một trong ba bài toán thì thuật toán đó cũng có thể sử dụng để giải hai bài toán còn lại.

Chu trình âm

- Đường đi ngắn nhất giữa hai đỉnh nào đó có thể không tồn tại.
  - Chẳng hạn, nếu không có đường đi từ *s* đến *t*, thì rõ ràng cũng không có đường đi ngắn nhất từ *s* đến *t*.
  - Ngoài ra, nếu đồ thị chứa cạnh có trọng số âm thì có thể xảy ra tình huống: độ dài đường đi giữa hai đỉnh nào đó có thể làm bé tuỳ ý.

Chu trình âm



- Xét đường đi từ a đến e:

  $a, k(b, c, d, b), e.$

  nghĩa là ta đi k lần vòng theo chu trình

  $C: b, c, d, b$

  trước khi đến e. Độ dài của đường đi này là bằng:

  $c(a,b) + k\,\rho(C) + c(b,e) = 2 - 10k \longrightarrow -\infty$, khi $k \longrightarrow +\infty$.

Thuật toán Dijkstra

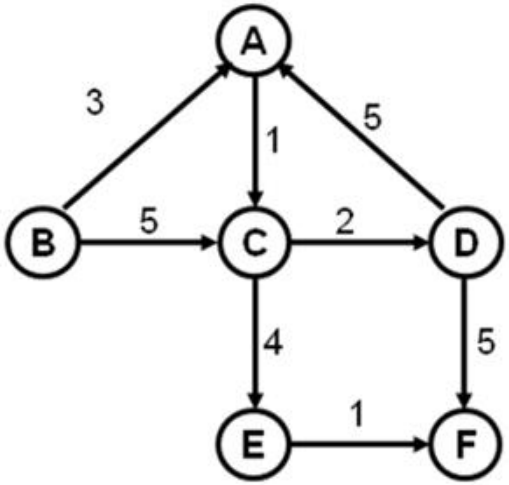
- Thuật toán Dijkstra được đề xuất để giải bài toán: Tìm đường đi ngắn nhất từ một đỉnh nguồn s đến tất cả các đỉnh còn lại trên đồ thị với trọng số không âm trên cạnh.
- Trong quá trình thực hiện thuật toán, với mỗi đỉnh v ta sẽ lưu trữ nhãn của đỉnh gồm các thông tin sau:
  - k[v]: biến bun có giá trị đúng nếu ta đã tìm được đường đi ngắn nhất từ s đến v, ban đầu biến này được khởi tạo giá trị false.
  - d[v]: khoảng cách ngắn nhất hiện biết từ s đến v. Ban đầu biến này được khởi tạo giá trị $+\infty$ đối với mọi đỉnh, ngoại trừ d[s] được đặt bằng 0.
  - p[v]: là đỉnh đi trước đỉnh v trong đường đi có độ dài d[v]. Ban đầu, các biến này được khởi tạo rỗng (chưa biết).

Thuật toán Dijkstra

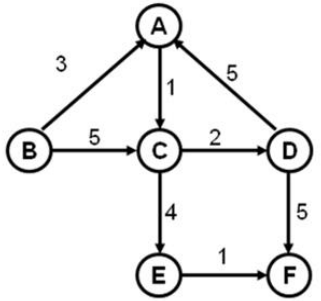
- Thuật toán lặp lại các thao tác sau đây cho đến khi tất cả các đỉnh được khảo sát xong (nghĩa là k[v] = true với mọi v):
  - Trong tập đỉnh với k[v] = false, chọn đỉnh v có d[v] là nhỏ nhất.
  - Đặt k[v] = true.
  - Với mỗi đỉnh w kề với v và có k[w]= false, ta kiểm tra d[w] > d[v] + c(v, w). Nếu đúng thì đặt lại d[w] = d[v] + c(v, w) và đặt p[w] = v.

Ví dụ

- Tìm đường đi ngắn nhất từ đỉnh B đến các đỉnh còn lại trên đồ thị sau đây



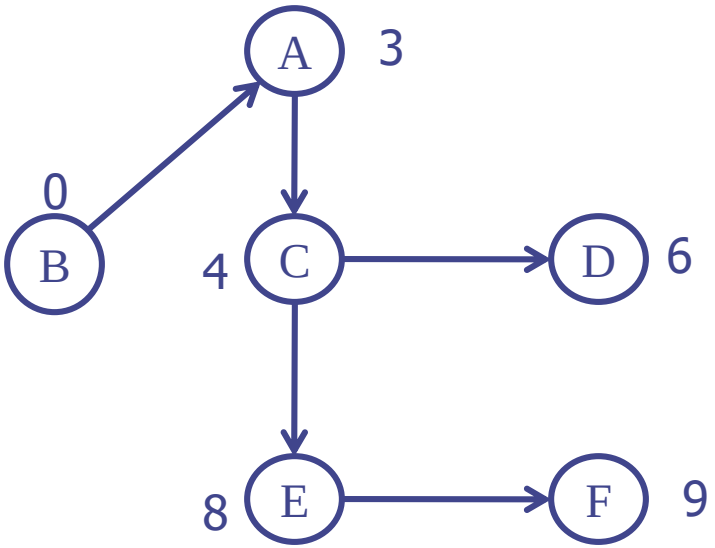
Bảng tính toán theo thuật toán Dijkstra



Bước lặp	A			B			C			D			E			F		
	d	*p*	*k*	*d*	*p*	*k*	*d*	*p*	*k*	*d*	*p*	*k*	*d*	*p*	*k*	*d*	*p*	*k*
Khởi tạo	∞	-	F	0	-	F	∞	-	F	∞	-	F	∞	-	F	∞	-	F
1	3	B	F	**0**	-	**T**	5	B	F	∞	-	F	∞	-	F	∞	-	F
2	**3**	**B**	**T**				4	A	F	∞	-	F	∞	-	F	∞	-	F
3							**4**	**A**	**T**	6	C	F	8	C	F	∞	-	F
4										**6**	**C**	**T**	8	C	F	11	D	F
5													**8**	**C**	**T**	9	E	F
6																**9**	**E**	**T**

Kết quả thực hiện

- Tập các cạnh {(p[v], v): v ∈ V-{B}} tạo thành một cây được gọi là ***cây đường đi ngắn nhất*** từ đỉnh B đến tất cả các đỉnh còn lại. Cây này được cho trong hình vẽ sau đây:



Cài đặt thuật toán với các cấu trúc dữ liệu

- Để cài đặt thuật toán Dijkstra chúng ta sử dụng bộ nhãn của các đỉnh:
- Nhãn của mỗi đỉnh v gồm 3 thành phần cho biết các thông tin:
  - k[v] - đã tìm được đường đi ngắn nhất từ đỉnh nguồn đến v hay chưa,
  - d[v] - khoảng cách (độ dài đường đi) từ s đến v hiện biết
  - p[v] - đỉnh đi trước đỉnh v trong đường đi tốt nhất hiện biết.
- Các thành phần này sẽ được cất giữ tương ứng trong các biến k[v], d[v] và p[v].

Cài đặt trực tiếp

```
Dijkstra_Table(G, s)
 1. for u ∈ V do {
2.      d[u] ← infinity;
3.      p[u] ← NIL;
4.      k[u] ← FALSE;
5.  }
6.  d[s] ← 0;
   // s là đỉnh nguồn
7.  T = V;
8.  while T ≠ ∅ do {
9.      u ← đỉnh có d[u] là nhỏ nhất trong T;
10.     k[u]=TRUE;
11.     T = T–{u};
12.     for (v ∈ Adj(u)) && !k[v] do
13.         if  d[v] > d[u] + c[u, v] {
14.             d[v] = d[u] + c[u, v];
15.             p[v] = u;
16.         }
17.  }
```

Dễ dàng nhận thấy rằng **Dijkstra_Table(G, s)** đòi hỏi thời gian $O(|V|^2+|E|)$.

Cài đặt thuật toán Dijkstra sử dụng hàng đợi có ưu tiên

- Do tại mỗi bước ta cần tìm ra đỉnh với nhãn khoảng cách nhỏ nhất, nên để thực hiện thao tác này một cách hiệu quả ta sẽ sử dụng hàng đợi có ưu tiên (Priority Queue – PQ).
- Dưới đây ta mô tả thuật toán Dijkstra với hàng đợi có ưu tiên:

Cài đặt thuật toán Dijkstra sử dụng PQ: Khởi tạo

Dijkstra_Heap(G, s)

1. **for** u ∈ V **do** {
2. d[u] ← infinity;
3. p[u] ← NIL;
4. k[u] ← FALSE;
5. }
6. d[s] ← 0; // s là đỉnh nguồn
7. Q ← Build_Min_Heap(d[V]);

// Khởi tạo hàng đợi có ưu tiên Q từ d[V] = (d[v], v∈V)

Cài đặt thuật toán Dijkstra sử dụng PQ: Lặp

```
8.  while Not Empty(Q) do {
9.      u ← Extract-Min(Q);  // loại bỏ gốc của Q và đưa vào u
10.     k[u]=TRUE;
11      for (v ∈ Adj(u)) && !k[v] do
12          if  d[v] > d[u] + c[u, v] {
13.                 d[v] = d[u] + c[u, v];
14.                 p[v] = u;
15.                 Decrease_Key(Q,v,d[v]);
16.         }
17.  }
```

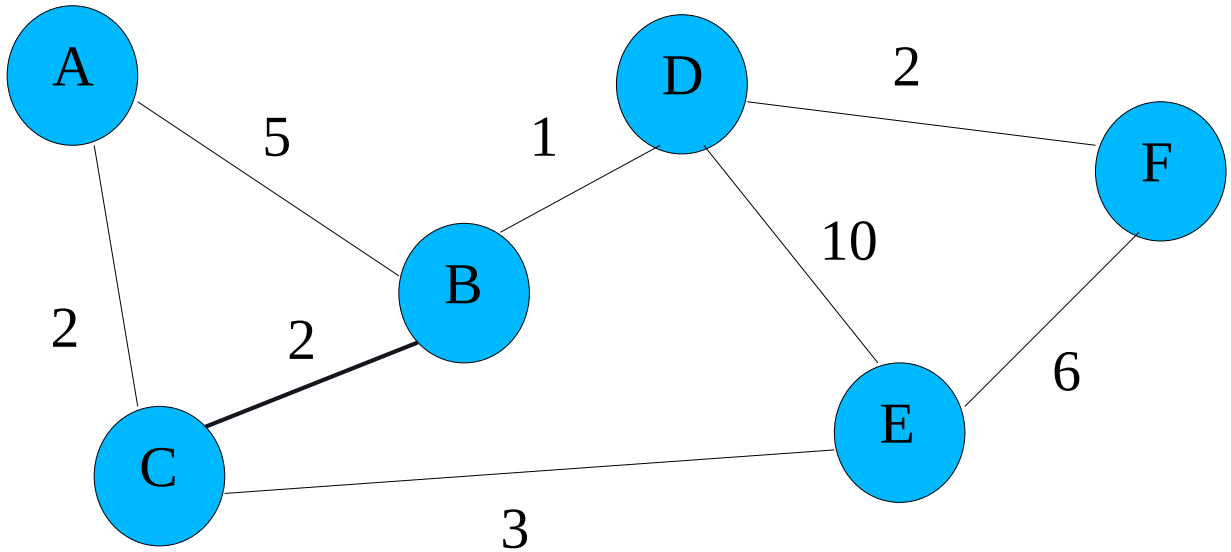
Phân tích thời gian tính của thuật toán

- Vòng lặp for ở dòng 1 đòi hỏi thời gian $O(|V|)$.
- Việc khởi tạo đống min đòi hỏi thời gian $O(|V|)$.
- Vòng lặp while ở dòng 8 lặp $|V|$ lần do đó thao tác Extract-Min thực hiện $|V|$ lần và đòi hỏi thời gian $O(|V| \log|V|)$.
- Thao tác Decrease_Key ở dòng 15 phải thực hiện không quá $O(|E|)$ lần. Do đó thời gian thực hiện thao tác này trong thuật toán là $O(|E| \log|V|)$.
- Vậy tổng cộng thời gian tính của thuật toán là

$$\boldsymbol{O((|E| + |V|) \log|V|).}$$

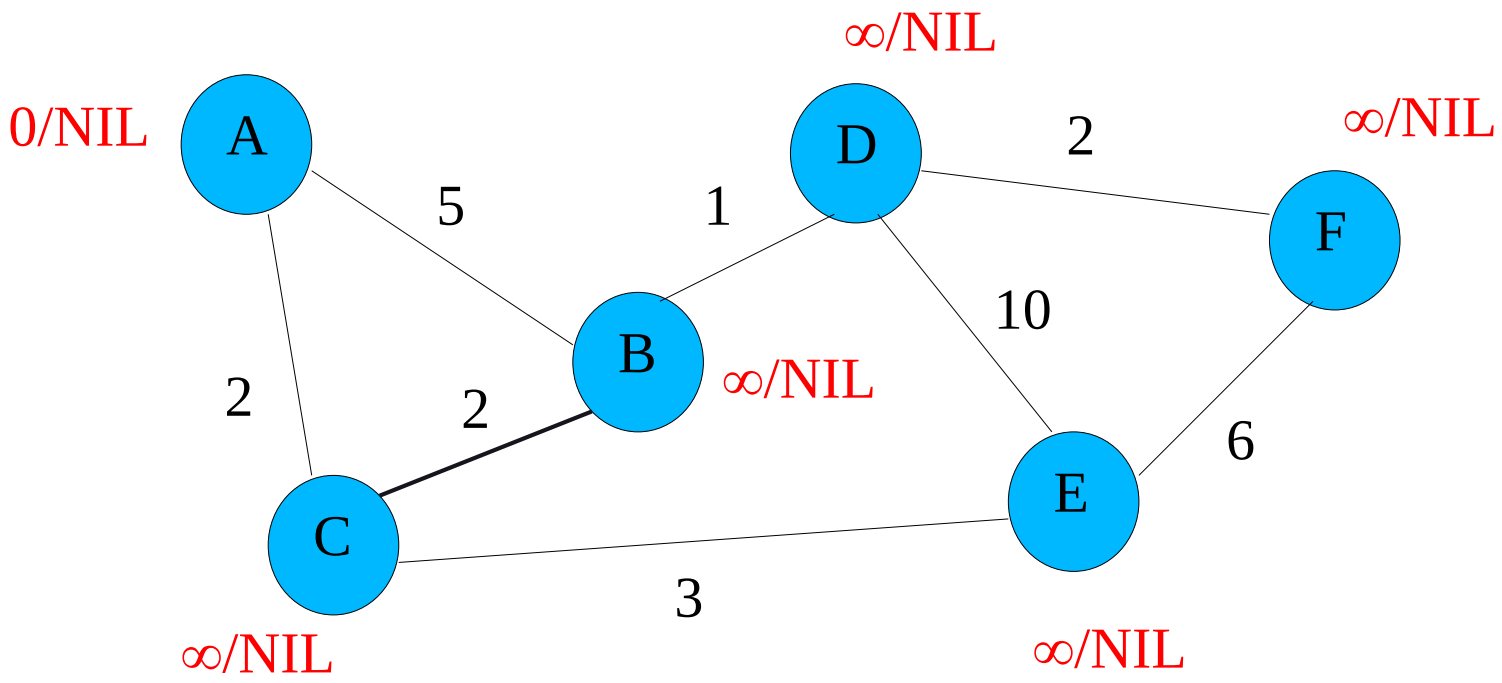
Ví dụ: Dijkstra algorithm

Tìm đường đi ngắn nhất từ đỉnh A đến tất cả các đỉnh còn lại



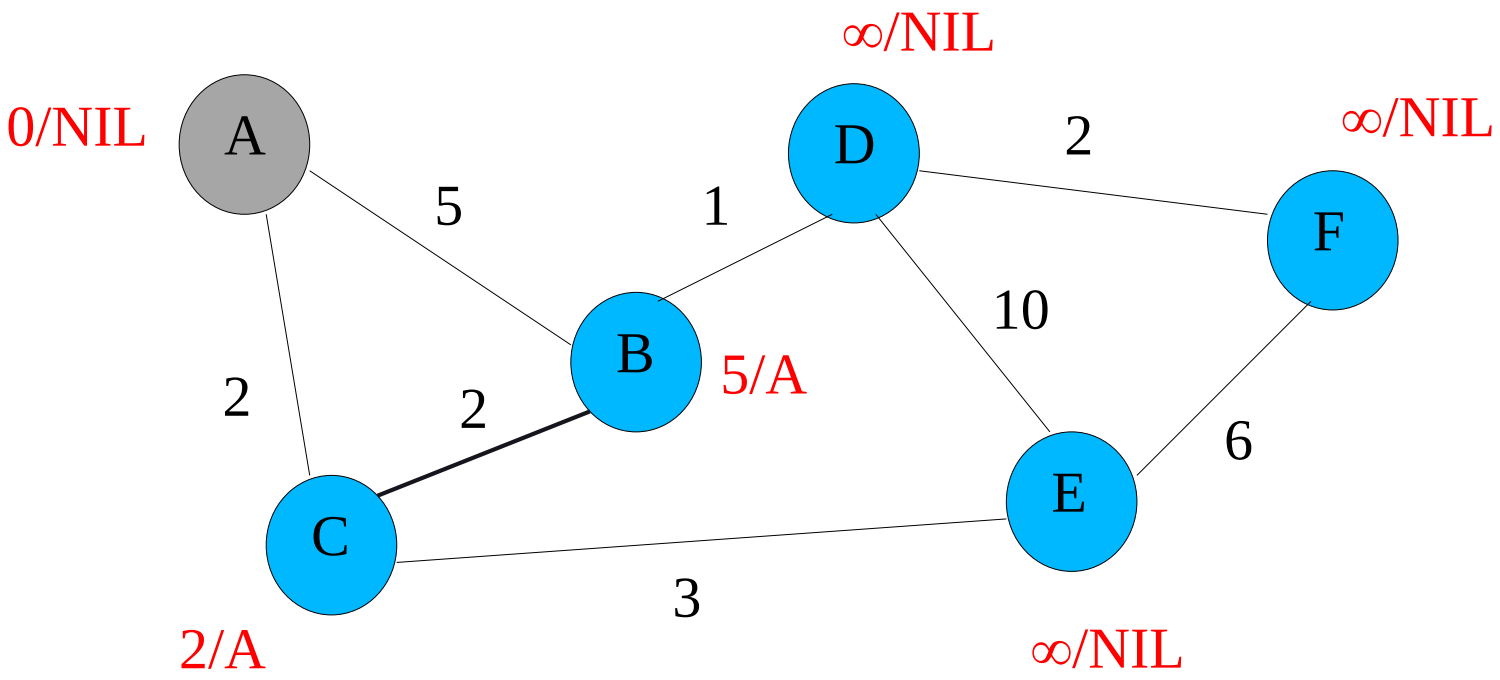
Thuật toán Dijkstra: Khởi tạo

Priority queue: A, B, C, D, E, F



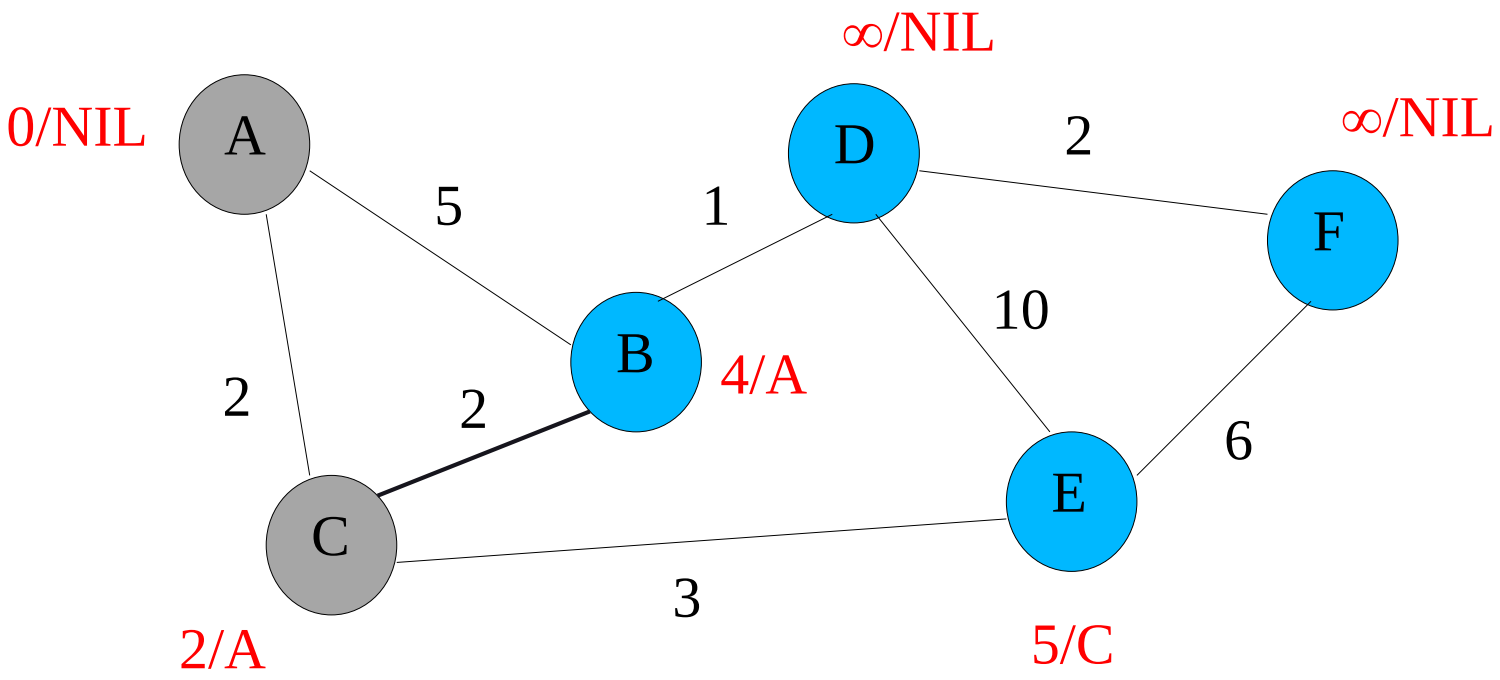
Thuật toán Dijkstra: Cố định nhãn đỉnh A

Priority queue: C, B, D, E, F



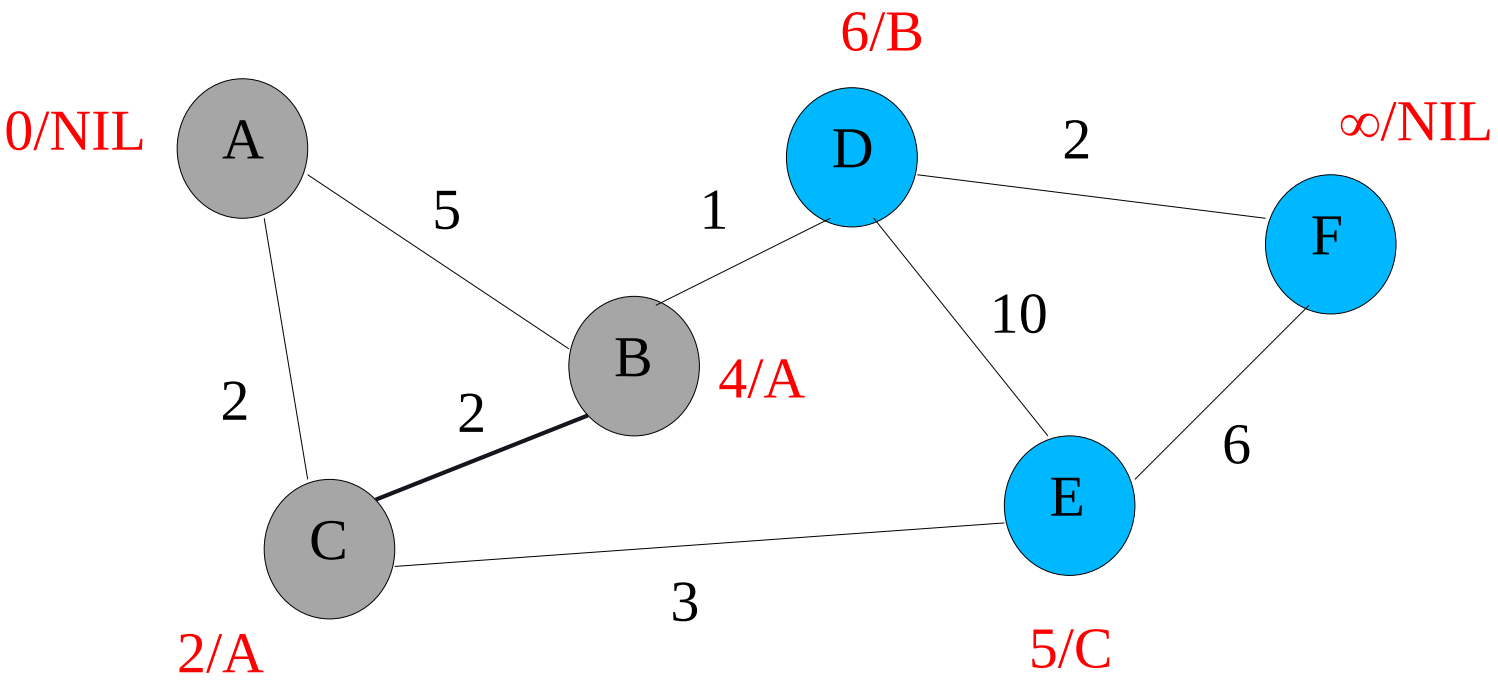
Thuật toán Dijkstra: Cố định nhãn đỉnh C

Priority queue: B, E, D, F



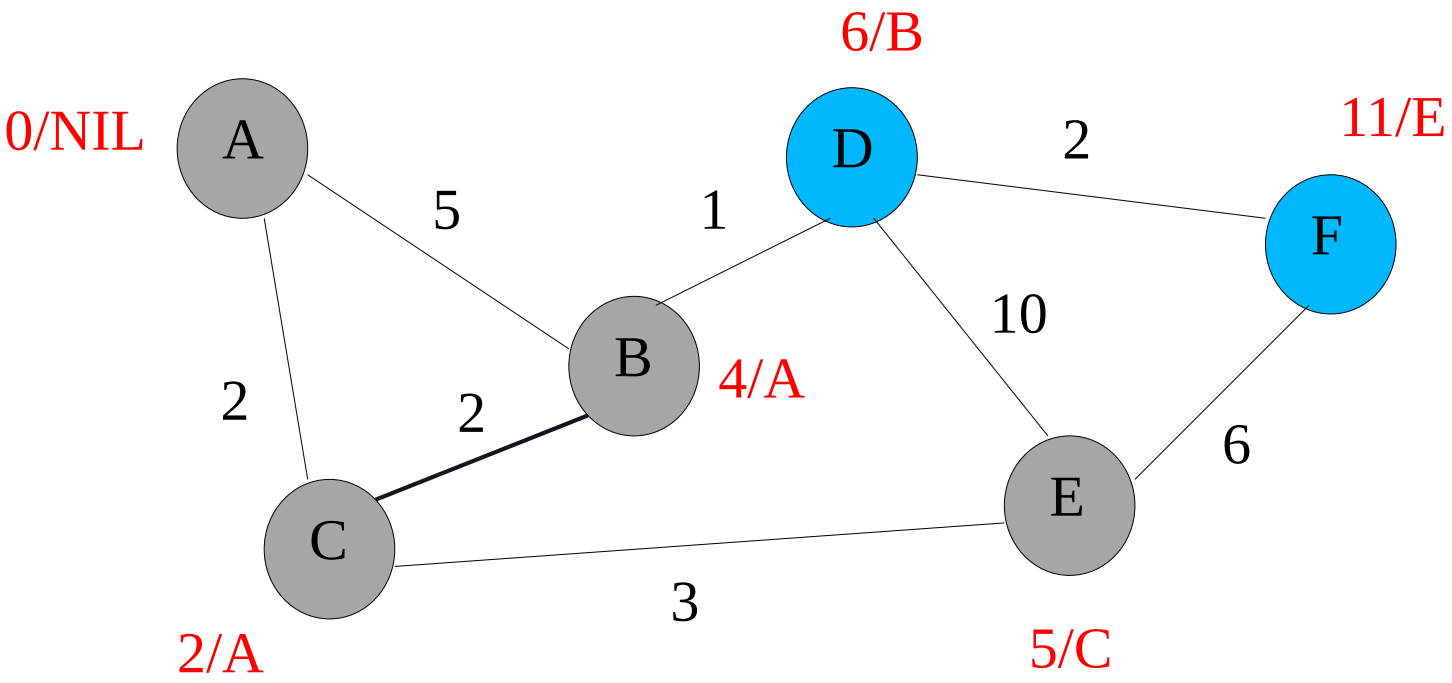
Thuật toán Dijkstra: Cố định nhãn đỉnh B

Priority queue: E, D, F



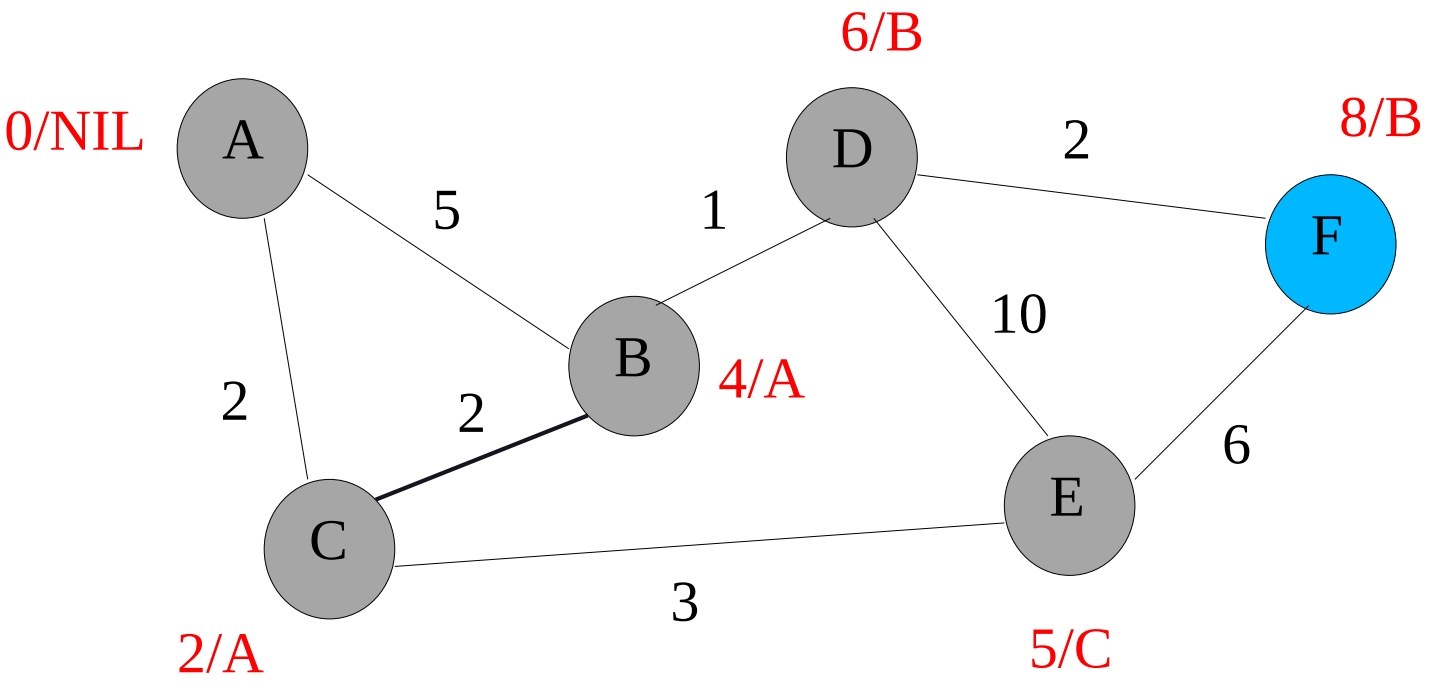
Thuật toán Dijkstra: Cố định nhãn đỉnh E

Priority queue: D, F



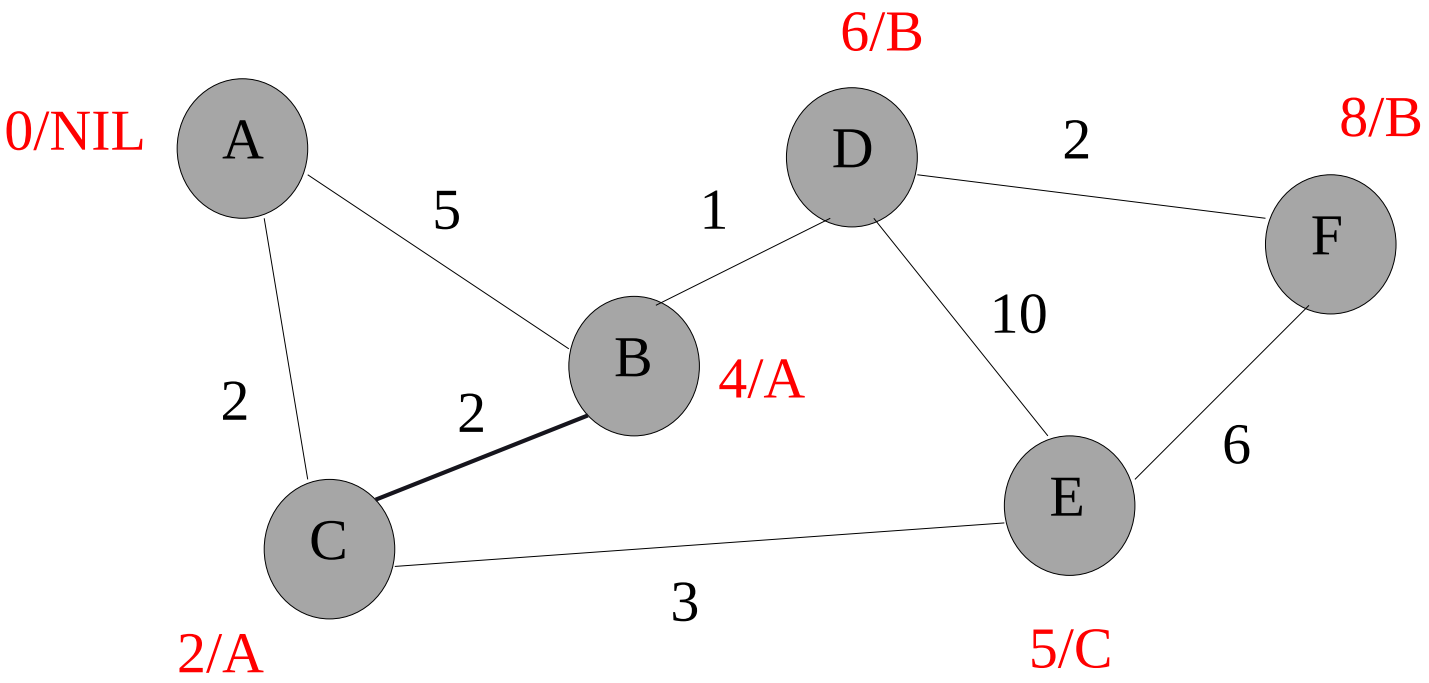
Thuật toán Dijkstra: Cố định nhãn đỉnh D

Priority queue: F



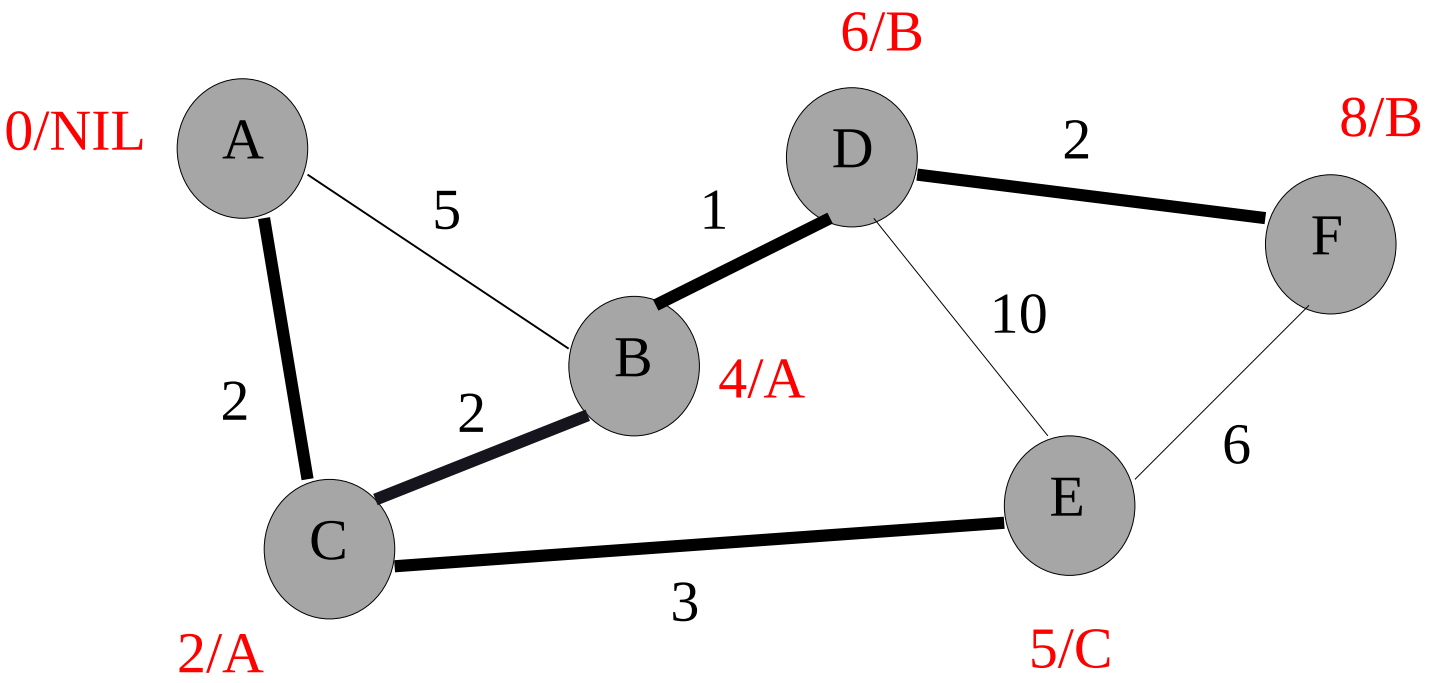
Thuật toán Dijkstra: Cố định nhãn đỉnh F

Priority queue:



Kết quả

Cây đường đi ngắn nhất xuất phát từ A:



Ứng dụng: Canh gác bảo tàng

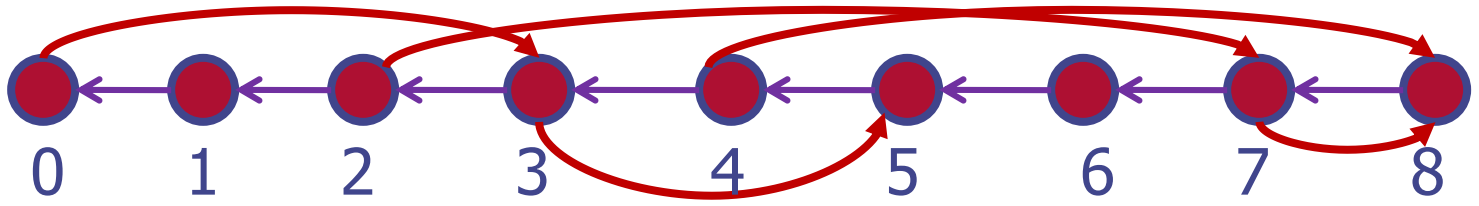
- **Bài toán.** Viện bảo tàng X tổ chức cuộc trưng bày giới thiệu một số bức tranh gốc của danh hoạ Leonard de Vinci. Những bức tranh của các danh hoạ thường là mục tiêu của nhiều tổ chức trộm cắp chuyên nghiệp. Vì thế, Ban giám đốc bảo tàng cần giải quyết bài toán bảo vệ an toàn cho các bức tranh độc nhất vô nhị này. Theo kế hoạch, cuộc triển lãm sẽ diễn ra trong khoảng thời gian n giờ. Thời điểm bắt đầu triển lãm được tính là 0. Có m vệ sỹ có nghiệp vụ cao có thể thuê để canh gác các bức tranh. Để đơn giản, các vệ sỹ sẽ được gán số hiệu từ 1 đến m. Vệ sỹ i chấp nhận đứng canh trong khoảng thời gian từ thời điểm s_i đến thời điểm t_i ($0 \le s_i < t_i \le n$) với tiền công là c_i, $i = 1, 2, \ldots, m$.

Ứng dụng

- **Yêu cầu:** Hãy giúp Ban giám đốc lựa chọn thuê các vệ sỹ nào trong số *m* vệ sỹ để tại bất cứ thời điểm nào trong thời gian diễn ra triển lãm luôn có ít nhất một vệ sỹ đứng canh, đồng thời tổng chi phí phải trả cho các vệ sỹ được thuê là nhỏ nhất.
- **Dữ liệu:** Vào từ file văn bản GALERY.INP:
  - Dòng đầu tiên chứa hai số nguyên dương *n* và *m* **($n, m \leq 10^5$);**
  - Dòng thứ *i* trong số *m* dòng tiếp theo chứa ba số nguyên không âm s_i, t_i, c_i ($0 < c_i \leq 10^5$), $i = 1, 2, \ldots, m$.
- **Kết quả:** Ghi ra file văn bản GALERY.OUT chi phí nhỏ nhất phải trả để thuê vệ sĩ

Qui về bài toán đường đi ngắn nhất

- Xây dựng đồ thị $G = (V, E)$ với trọng số trên cạnh:
  - $V = \{0, 1, ..., n\}$: mỗi đỉnh của đồ thị tương ứng với thời điểm;
  - $E = \{e_i=(s_i,t_i): i = 1, 2,..., m\} \cup \{f_k=(k,k\text{-}1): k=1, 2, ..., n\}$;
  - $c(e_i) = c_i$, $i = 1, 2,..., m$; $c(f_k) = 0$, $k=1, 2, ..., n$.
- Như vậy đồ thị G có n đỉnh, và $n+m$ cạnh.
- Bài toán đặt ra dẫn về tìm đường đi ngắn nhất từ đỉnh 0 đến đỉnh n trên đồ thị G.



Phân tích

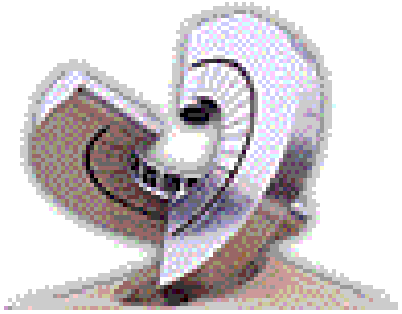
- Theo đề bài: Đồ thị có 10^5 đỉnh, và $2*10^5$ cạnh
- Thời gian tính:
  - Dùng Dijkstra Table: $O(|V|^2)$

    tức là quãng $O(10^{10})$
  - Dùng Dijkstra với PQ: $O((|E| + |V|) \log|V|)$

    tức là quãng $O((3*10^5) \log 10^5) \sim O(60*10^5)$
- **Chú ý:**

$10^{10}/(60*10^5) = 100000/60 = 10000/6 >$ **1666**

Questions?



7. Bài toán ghép cặp

Graphs Matching

Nội dung

Bài toán ghép cặp trên đồ thị

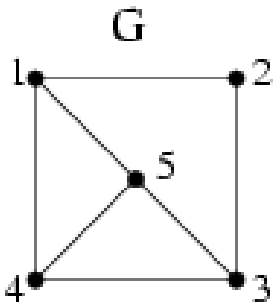
- Giả sử *G=(V,E) là đồ thị vô hướng, mỗi cạnh (v,w) của nó được gán với một số thực c(v,w) gọi là trọng số của nó.*
- **Định nghĩa.** ***Cặp ghép M trên đồ thị G là tập các cạnh của đồ thị trong đó không có hai cạnh nào có đỉnh chung.***
- *Số cạnh trong M* - ***kích thước,***
- *Tổng trọng số của các cạnh trong M* - ***trọng lượng của cặp ghép.***
- *Cặp ghép với kích thước lớn nhất được gọi là* ***cặp ghép cực đại.***
- *Cặp ghép với trọng lượng lớn nhất được gọi là* ***cặp ghép lớn nhất.***
- *Cặp ghép được gọi là* ***đầy đủ (hoàn hảo) nếu mỗi đỉnh của đồ thị là đầu mút của ít nhất một cạnh trong cặp ghép.***

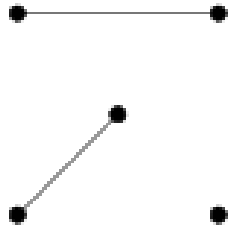
Hai bài toán

- **Bài toán cặp ghép cực đại:** ***Tìm cặp ghép với kích thước lớn nhất trong đồ thị G.***
- **Bài toán cặp ghép lớn nhất:** ***Tìm cặp ghép với trọng lượng lớn nhất trong đồ thị G.***

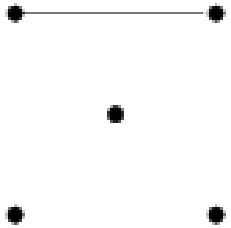
- *Ta hạn chế xét các bài toán đặt ra trên đồ thị hai phía* $G = (X \cup Y, E)$.

Ví dụ

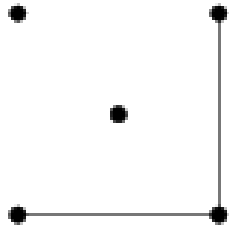




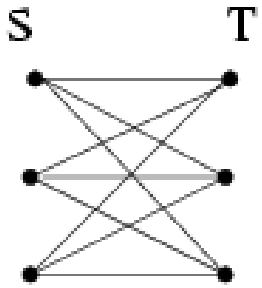
Cặp ghép cực đại

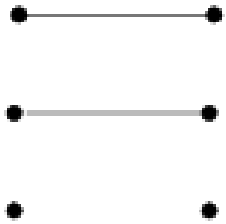


Cặp ghép

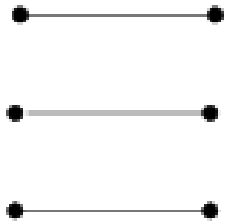


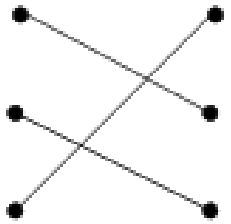
không là cặp ghép





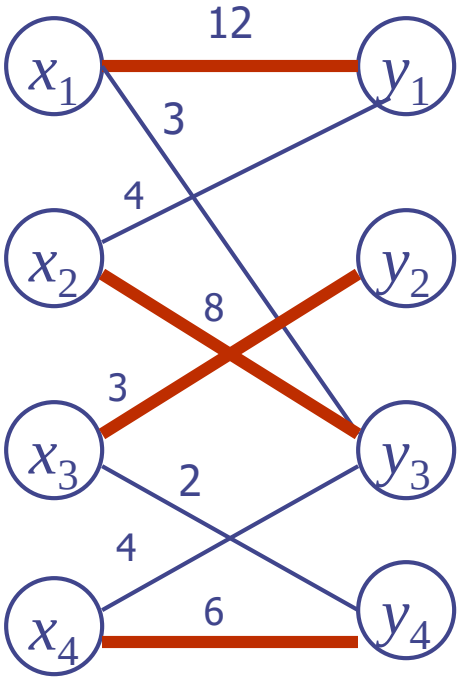
Cặp ghép





Cặp ghép hoàn hảo

Ví dụ



- Cặp ghép lớn nhất:

$M = \{(x_1, y_1), (x_2, y_3), (x_3, y_2), (x_4, y_4)\}$

Có trọng lượng 29.

Nội dung

7.1. Giới thiệu bài toán

7.2. Bài toán cặp ghép cực đại

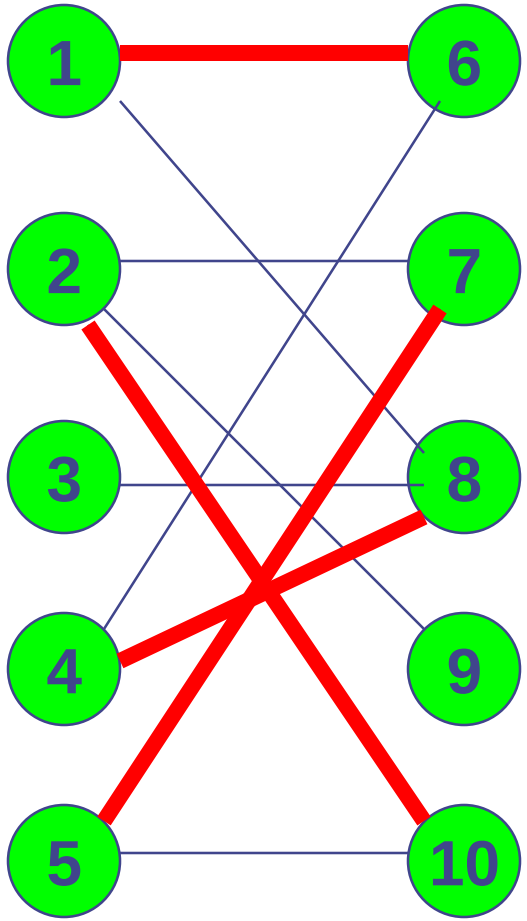
7.3. Bài toán cặp ghép lớn nhất

Bµi to¸n cÆp ghÐp cùc ®¹i trªn ®å thÞ hai phÝa

XÐt ®å thÞ hai phÝa
$G = (X \cup Y, E)$.

CÆp ghÐp lµ tËp c¹nh mµ kh«ng cã hai c¹nh nµo cã chung ®Ønh

***Bµi to¸n*: T×m cÆp ghÐp kÝch thíc lín nhÊt**



Bµi to¸n cÆp ghÐp cùc ®¹i trªn ®å thÞ hai phÝa

- Giả sử M là một cặp ghép trên G.
- Nếu cạnh $e = (x, y) \in M$, ta nói e là cạnh của cặp ghép (hay cạnh đậm) và các đỉnh x, y là các đỉnh ***đậm*** (hay ***không tự do***).
- Nếu cạnh $e = (x, y) \notin M$, thì ta nói e là cạnh nhạt còn các đỉnh x, y là các đỉnh ***nhạt*** (hay ***tự do***).

Đường tăng cặp ghép

- *Một đường đi trên đồ thị G mà trong đó hai cạnh liên tiếp là không cùng đậm hay nhạt sẽ được gọi là* ***đường đi luân phiên đậm/nhạt (hay gọi ngắn gọn là đường đi luân phiên).***
- *Đường đi luân phiên bắt đầu từ một đỉnh tự do thuộc tập X và kết thúc ở một đỉnh tự do thuộc tập Y được gọi là* ***đường tăng cặp ghép.***

Định lý Berge

- **Định lý 1 (Berge C).** ***Cặp ghép M là cực đại khi và chỉ khi không tìm được đường tăng cặp ghép.***
- **CM:** ***Điều kiện cần.*** Bằng phản chứng. Giả sử M là cặp ghép cực đại nhưng vẫn tìm được đường tăng cặp ghép

  $$P \equiv x_0, y_1, x_1, y_2, \ldots, x_k, y_0$$

  trong đó x_0 và y_0 là các đỉnh tự do.

  Gọi E_P là tập các cạnh của đồ thị nằm trên đường đi P

  $$E_P = \{ (x_0,y_1), (y_1, x_1), \ldots, (x_k, y_0) \}.$$

  Dễ thấy số lượng cạnh nhạt trong E_P là bằng số lượng cạnh đậm của nó cộng với 1. Để đơn giản trong phần dưới đây ta đồng nhất ký hiệu đường đi P với tập cạnh E_P của nó. Xây dựng cặp ghép M' theo qui tắc:

  $$M' = (M \cup P) \setminus (M \cap P).$$

  Dễ thấy M' cũng là cặp ghép và rõ ràng $|M'| = |M| + 1$. Mâu thuẫn thu được đã chứng minh điều kiện cần.

Định lý Berge

- **Điều kiện đủ.** Giả sử cặp ghép M chưa là cặp ghép cực đại. Gọi M^* là cặp ghép cực đại. Xét đồ thị $G' = (V, M \cup M^*)$. Rõ ràng hai cạnh liên tiếp trong mỗi đường đi cũng như mỗi chu trình trong G' không thể thuộc cùng một cặp ghép M hoặc M^*. Vì vậy, mỗi đường đi cũng như mỗi chu trình trong G' đều là đường luân phiên M/M^*. Do $|M^*| > |M|$, nên rõ ràng là luôn tìm được ít nhất một đường đi luân phiên M/M^* mà trong đó số lượng cạnh thuộc M^* là lớn hơn số lượng cạnh thuộc M. Đường đi đó chính là đường tăng cặp ghép trên đồ thị G.
- Định lý được chứng minh.
- Chú ý: Trong chứng minh định lý ta không sử dụng tính hai phía của G. Do đó, Định lý 1 là đúng với đồ thị vô hướng bất kỳ.

Thuật toán tìm cặp ghép cực đại

- **Đầu vào:** Đồ thị vô hướng $G = (V, E)$.
- ***Bíc khëi t¹o.*** X©y dùng cÆp ghĐp M trong ®å thÞ G (cã thÓ b¾t ®Çu tõ $M = \varnothing$).
- ***Bíc lÆp.***
- Kiểm tra tiêu chuẩn tối ưu: Nếu đồ thị G không chứa đường tăng cặp ghép thì M là cặp ghép cực đại, thuật toán kết thúc.
- Ngược lại, gọi P là một đường tăng cặp ghép xuất phát từ đỉnh tự do $x_0 \in X$, kết thúc ở đỉnh tự do $y_0 \in Y$. Tăng cặp ghép theo qui tắc

$$M := (M \cup P) \setminus (M \cap P),$$

rải lÆp l¹i bíc lÆp.

Tìm đường tăng

- Từ đồ thị G ta xây dựng đồ thị có hướng $G_M = (X \cup Y, E_M)$ với tập cung E_M được bằng cách định hướng lại các cạnh của G theo quy tắc sau:

  i) Nếu $(x,y) \in M \cap E$, thì $(y,x) \in E_M$;

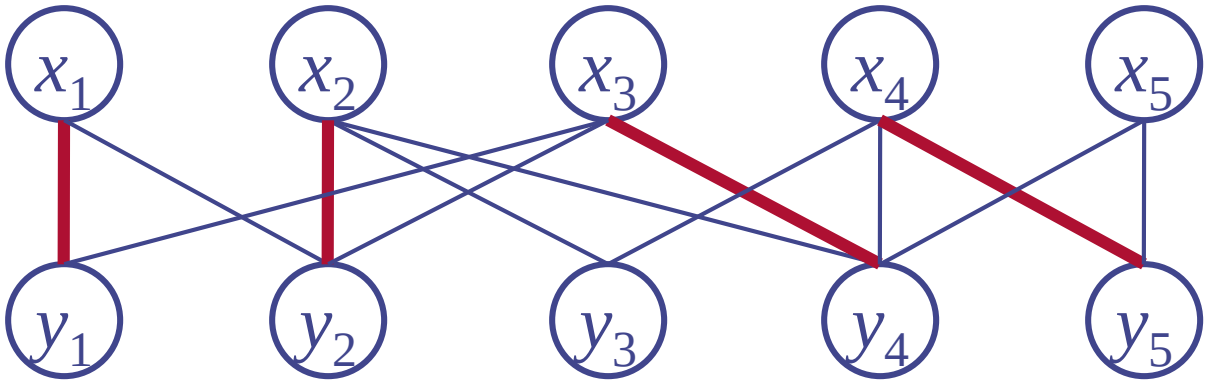
  ii) Nếu $(x,y) \in E \setminus M$, thì $(x,y) \in E_M$.

- Đồ thị G_M sẽ được gọi là **đồ thị tăng cặp ghép.**
- Dễ thấy:
- Đường tăng cặp ghép tương ứng với một đường đi xuất phát từ một đỉnh tự do $x_0 \in X$ kết thúc tại một đỉnh tự do $y_0 \in Y$ trên đồ thị G_M.
- Ngược lại, một đường đi trên đồ thị G_M xuất phát từ một đỉnh tự do $x_0 \in X$ kết thúc tại một đỉnh tự do $y_0 \in Y$ sẽ tương ứng với một đường tăng cặp ghép trên đồ thị G.
- Vì vậy, để xét xem đồ thị G có chứa đường tăng cặp ghép hay không, có thể thực hiện thuật toán tìm kiếm theo chiều rộng trên đồ thị G_M bắt đầu từ các đỉnh tự do thuộc tập X.

Thuật toán

- Sử dụng cách tìm đường tăng cặp ghép theo nhận xét vừa nêu, từ sơ đồ tổng quát dễ dàng xây dựng thuật toán để giải bài toán tìm cặp ghép cực đại trên đồ thị hai phía với thời gian tính $O(n^3)$, trong đó $n = \max(|X|, |Y|)$.

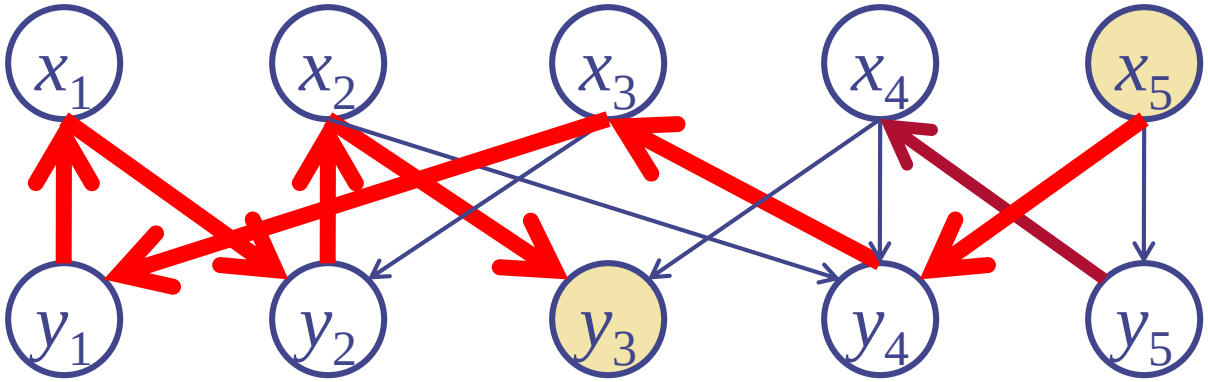
Ví dụ



- Sử dụng thuật toán tham lam tìm cặp ghép xuất phát

  $M = \{(x_1,y_1), (x_2,y_2), (x_3,y_4), (x_4,y_5)\}$.

Ví dụ



Đồ thị tăng cặp ghép G_M

- Tìm đường tăng nhờ tìm kiếm theo chiều sâu bắt đầu từ đỉnh tự do x_5
- Tìm được đường tăng:

$$x_5 \rightarrow y_4 \rightarrow x_3 \rightarrow y_1 \rightarrow x_1 \rightarrow y_2 \rightarrow x_2 \rightarrow y_3.$$

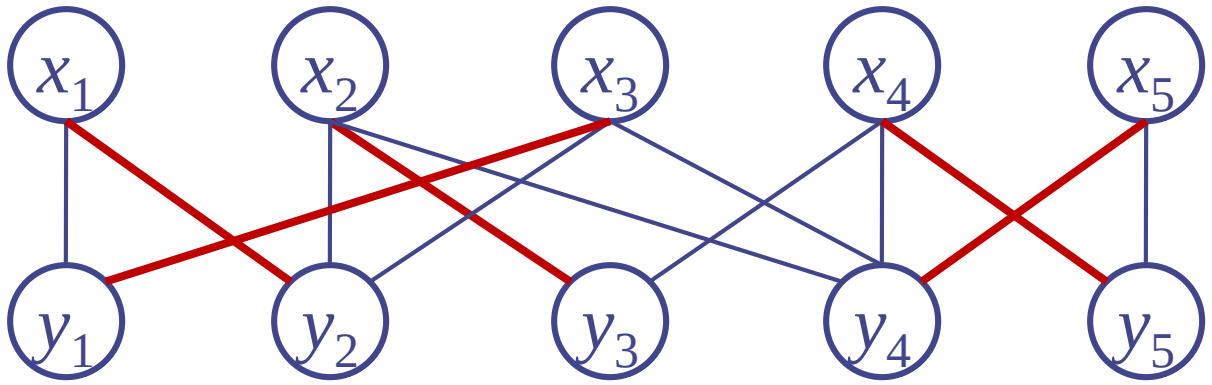
Tìm được đường tăng:

$$x_5 \rightarrow y_4 \rightarrow x_3 \rightarrow y_1 \rightarrow x_1 \rightarrow y_2 \rightarrow x_2 \rightarrow y_3.$$

- Đảo ngược màu đậm nhạt của các cạnh trên đường tăng:

$$x_5 \rightarrow y_4 \rightarrow x_3 \rightarrow y_1 \rightarrow x_1 \rightarrow y_2 \rightarrow x_2 \rightarrow y_3.$$

Ta thu được cặp ghép



Không còn đỉnh tự do (cặp ghép thu được là hoàn hảo), thuật toán kết thúc.

Cài đặt

Cấu trúc dữ liệu

```
Var
    A : Array[1..100,1..100] of Byte;   (* Ma trận kề của đồ thi hai phía G *)
    Truoc,                              (* Ghi nhận đường đi *)
    Vo,                                 (* Vo[x]- đỉnh được ghép với x∈X *)
    Chong : Array[1..100] of Byte;      (* Chong[y]-đỉnh được ghép với y ∈Y *)
    N, x0, y0, Cnt : Byte;
    Stop : Boolean;

(* Nếu (x, y) ∈ M thì Vo[x]=y; Chong[y]=x.
  Vo[x]=0 => x là đỉnh nhạt; Chong[y]=0 => y là đỉnh nhạt *)
```

Tìm đường tăng

```
Procedure Tim(x:Byte);
  var y: Byte;
   begin
     For y:=1 to N do
       If (A[x,y]=1)and(Truoc[y]=0)and(y0=0) then
      begin
           Truoc[y]:=x;
           If Chong[y]<>0 then Tim(Chong[y])
           else
           begin
                  y0:=y;
                  Exit;
           end;
       end;
   end;
```

```
Procedure Tim_Duong_Tang;
begin
     Fillchar(Truoc,Sizeof(Truoc),0);
     y0:=0;
     For x0:=1 to N do
     begin
        If Vo[x0]=0 then Tim(x0);
        If y0<>0 then exit;
     end;
     Stop:=true;
end;
```

Thủ tục MaxMatching

```
Procedure Tang;
var temp: Byte;
begin
    Inc(Cnt);
    While Truoc[y0]<>x0 do
    begin
        Chong[y0]:=Truoc[y0];
        Temp:=Vo[Truoc[y0]];
        Vo[Truoc[y0]]:=y0;
        y0:=Temp;
    end;
    Chong[y0]:=x0;
    Vo[x0]:=y0;
end;
```

```
Procedure MaxMatching;
begin
     Stop:=false;
    Fillchar(Vo,Sizeof(Vo),0);
    Fillchar(Chong,Sizeof(Chong),0);
    Cnt:=0;
    While not Stop do
     begin
         Tim_duong_tang;
         If not Stop then Tang;
     end;
end;
```

Nội dung

7.1. Giới thiệu bài toán

7.2. Bài toán cặp ghép cực đại

7.3. Bài toán cặp ghép lớn nhất

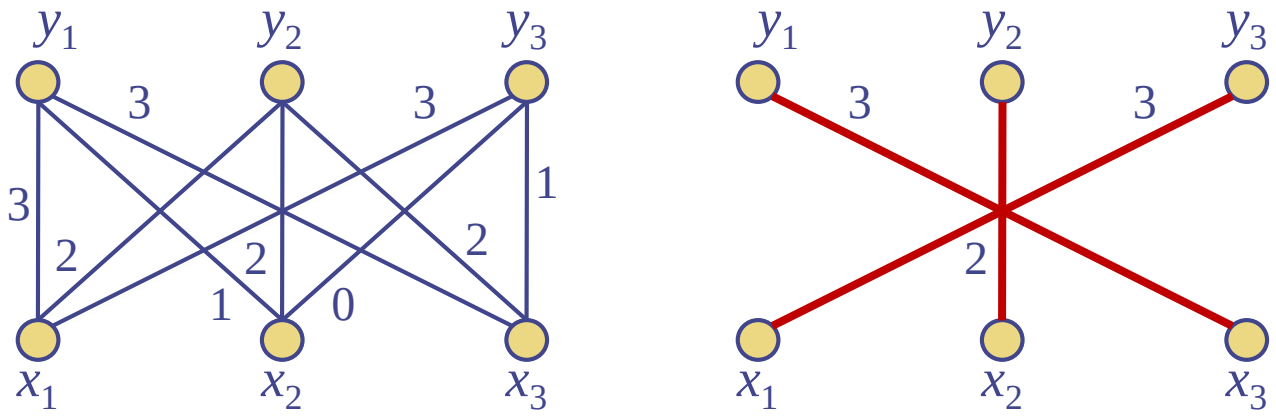
Phát biểu bài toán

- Xét $G = (X \cup Y, E)$ là đồ thị hai phía với trọng số trên cạnh $w(x, y)$, $\forall x \in X, y \in Y$.
- Ta gọi trọng lượng của cặp ghép M, ký hiệu là $w(M)$, là tổng trọng số trên các cạnh của M:

$$w(M) = \sum_{e \in M} w(e)$$

- **Bài toán cặp ghép lớn nhất:** Cho đồ thị hai phía với trọng số trên cạnh G, hãy tìm cặp ghép với trọng lượng lớn nhất.
- **Chú ý:** Bằng cách bổ sung các cạnh với trọng số 0 nếu cần, ta sẽ giả thiết là G là đồ thị hai phía đầy đủ.

Ví dụ



- Đồ thị G và cặp ghép tối ưu

$$M^* = \{ (x_1, y_3), (x_2, y_2), (x_3, y_1)\}$$

- với trọng lượng:

$$w(M^*) = 3 + 2 + 3 = 8.$$

Bµi to¸n ph©n c«ng

- Có n công việc và n thợ. Mỗi thợ đều có khả năng thực hiện tất cả các công việc. Biết

  w_{ij} - *hiệu quả phân công thợ i làm việc j,*

  $(i, j = 1, 2,..., n)$.

- Cần tìm cách phân công thợ thực hiện các công việc sao cho mỗi thợ chỉ thực hiện một việc và mỗi việc chỉ do một thợ thực hiện, đồng thời tổng hiệu quả thực hiện các công việc là lớn nhất.

- **Chú ý:** Do phương án tối ưu của bài toán là không thay đổi nếu như ta cộng tất cả các số w_{ij} với cùng một hằng số α.Vì thế ta có thể giả thiết là $w_{ij} \geq 0$, $i, j = 1, 2,..., n$.

Mô hình toán học

- Đưa vào biến số
- $x_{ij} = 1$ phân thợ i thực hiện việc j,
  0, nếu trái lại

Mô hình toán học của bài toán:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}x_{ij} \to \max$$

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j = 1,\ldots,n;$$

$$\sum_{j=1}^{n} x_{ij} = 1, \quad i = 1,\ldots,n;$$

$$x_{ij} \geq 0, \quad i, j = 1,\ldots,n$$

$$x_{ij} - \text{nguyên}, \quad i, j = 1,\ldots,n$$

Mô hình toán học

- Bài toán đối ngẫu

$$\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}x_{ij} \to \max$$

$$\sum_{i=1}^{n} x_{ij} = 1, \quad j = 1,\ldots,n;$$

$$\sum_{j=1}^{n} x_{ij} = 1, \quad i = 1,\ldots,n;$$

$$x_{ij} \geq 0, \quad i, j = 1,\ldots,n,$$

$$x_{ij} - \text{nguyên}, \quad i, j = 1,\ldots,n$$

$$\sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j \to \min$$

$$v_j <> 0$$

$$u_i <> 0$$

$$u_i + v_j \geq w_{ij}$$

Qui về bài toán cặp ghép lớn nhất

- Xây dựng đồ thị hai phía đầy đủ $G = (X \cup Y, E)$
  - $X = \{x_1, x_2, ..., x_n\}$ tương ứng với các thợ,
  - $Y = \{y_1, y_2, ..., y_n\}$- tương ứng với các công việc.
- Mỗi cạnh (x_i, y_j) được gán cho trọng số $w(x_i, y_j) = w_{ij}$.
- Khi đó trong ngôn ngữ đồ thị, bài toán phân công có thể phát biểu như sau: Tìm trong đồ thị hai phía đầy đủ G cặp ghép đầy đủ có tổng trọng số là lớn nhất. Cặp ghép như vậy được gọi là **cặp ghép tối ưu**.
- Phần tiếp theo ta sẽ trình bày thuật toán giải bài toán phân công được phát biểu trong ngôn ngữ của bài toán cặp ghép lớn nhất.

C¬ së thuËt to¸n

- **Định nghĩa.** *Ta gọi một* ***phép gán nhãn chấp nhận được*** *cho các đỉnh của đồ thị* $G=(X \cup Y, E)$ *là một hàm số* f *xác định trên tập đỉnh* $X \cup Y$: $f : X \cup Y \to R$, *thoả mãn*

$$f(x) + f(y) \geq w(x,y), \forall x \in X, \forall y \in Y.$$

Một phép gán nhãn chấp nhận được như vậy dễ dàng có thể tìm được, chẳng hạn phép gán nhãn sau đây là chấp nhận được

$$f(x) = \max \{ w(x,y): y \in Y \}, \quad x \in X,$$

$$f(y) = 0 , \ y \in Y.$$

Đồ thị cân bằng

* Giả sử có f là một phép gán nhãn chấp nhận được, ký hiệu
  - $$E_f = \{(x,y) \in E:\ f(x) + f(y) = w(x,y)\}.$$
* Ký hiệu G_f là đồ thị con của G sinh bởi tập đỉnh $X \cup Y$ và tập cạnh E_f. Ta sẽ gọi G_f là ***đồ thị cân bằng***.

Tiêu chuẩn tối ưu

* **Định lý (Kuhn-Munkres)** ***Giả sử f là phép gán nhãn chấp nhận được. Nếu G_f chứa cặp ghép đầy đủ M*, thì M* là cặp ghép tối ưu.***

- **Chứng minh.** Giả sử G_f chứa cặp ghép đầy đủ M^*. Khi đó từ định nghĩa G_f suy ra M^* cũng là cặp ghép đầy đủ của đồ thị G. Do mỗi cạnh $e=(x,y)\in M^*$ đều là cạnh của G_f và mỗi đỉnh của G kề với đúng một cạnh của M^*, nên

$$w(M^*)=\sum_{e\in M^*} w(e)=\sum_{(x,y)\in M^*} w(x,y)=\sum_{(x,y)\in M^*}(f(x)+f(y))=\sum_{v\in V} f(v)$$

- Giả sử M là một cặp ghép đầy đủ tuỳ ý của G, khi đó

$$w(M)=\sum_{e\in M} w(e)=\sum_{(x,y)\in M} w(x,y)\le\sum_{(x,y)\in M}(f(x)+f(y))=\sum_{v\in V} f(v)$$

Suy ra $w(M^*)\ge w(M)$. Vậy M^* là cặp ghép tối ưu.

Sơ đồ thuật toán

- Ta sẽ bắt đầu từ một phép gán nhãn chấp nhận được f. Xây dựng đồ thị G_f. Bắt đầu từ một cặp ghép M nào đó trong G_f ta xây dựng cặp ghép đầy đủ trong G_f. Nếu tìm được cặp ghép đầy đủ M^*, thì nó chính là cặp ghép tối ưu. Ngược lại, ta sẽ tìm được cặp ghép cực đại không đầy đủ M'. Từ M' ta sẽ tìm cách sửa phép gán nhãn thành f' sao cho M' vẫn là cặp ghép của $G_{f'}$ và có thể tiếp tục phát triển M' trong $G_{f'}$, $v.v...$
- Quá trình được tiếp tục cho đến khi thu được cặp ghép đầy đủ trong đồ thị cân bằng.

Sơ đồ thuật toán

- **Kuhn-Munkres Algorithm**

  Bắt đầu từ phép gán nhãn f và cặp ghépM trong G_f

  while (M chưa là cặp ghép đầy đủ)

  Tìm đường tăng cặp ghép M trong G_f

  if (tìm được đường tăng)

  Tăng cặp ghép M

  else Điều chỉnh f thành f' sao cho $E_f \subset E_{f'}$

- **Nhận xét:**
  - Để ý rằng sau mỗi lần lặp, ta sẽ tăng được kích thước của M hoặc của E_f, do đó thuật toán phải kết thúc.
  - Khi thuật toán kết thúc, M là cặp ghép đầy đủ của G_f đối với một phép gán nhãn chấp nhận được f.
  - Do đó theo định lý Kuhn-Munkres , M sẽ là cặp ghép tối ưu.

Điều chỉnh nhãn

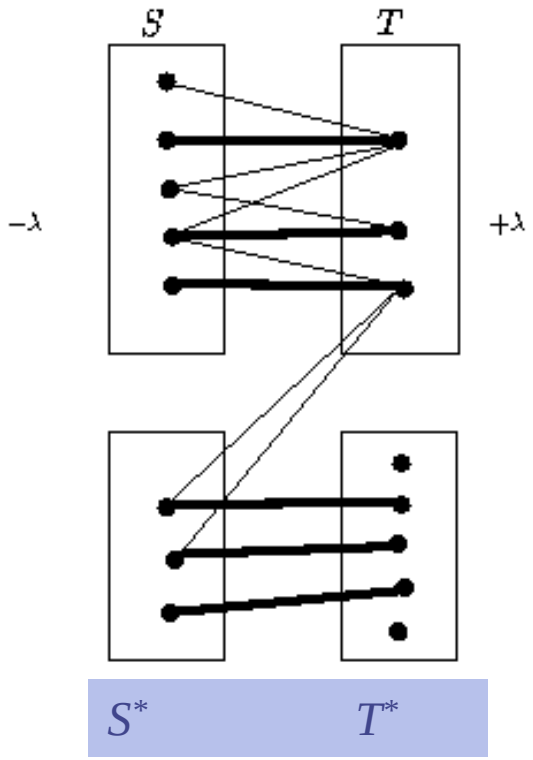
* Giả sử M là cặp ghép cực đại trong đồ thị G_f và M chưa là cặp ghép đầy đủ của G. Ta cần tỡm cách điều chỉnh phép gán nhãn f thoả mãn các yêu cầu đặt ra.
* Thực hiện tỡm kiếm theo chiều rộng từ các đỉnh tự do trong X. Gọi S là các đỉnh được thăm trong X, còn T là các đỉnh được thăm trong Y trong quá trỡnh thực hiện tỡm kiếm. Ký hiệu

$$S^* = X \setminus S, \quad T^* = Y \setminus T.$$

- $|S| > |T|$ (do mỗi đỉnh trong T đạt được từ một đỉnh nào đó trong S).

Điều chỉnh nhãn

* Từ tính chất của thuật toán tỡm kiếm theo chiều rộng, rõ ràng, không có cạnh nào từ S đến T^*. Để sửa chữa nhãn, chúng ta sẽ tiến hành giảm đồng loạt các nhãn trong S đi cùng một giá trị λ nào đó, và đồng thời sẽ tăng đồng loạt nhãn của các đỉnh trong T lên λ. Điều đó đảm bảo các cạnh từ S sang T (nghĩa là những cạnh mà một đầu mút thuộc S còn một đầu mút thuộc T) không bị loại bỏ khỏi đồ thị cân bằng



Các tập S và T trong thực hiện thuật toán. Chỉ vẽ các cạnh trong G_f.

Điều chỉnh nhãn

* Khi các nhãn trong S bị giảm, các cạnh trong G từ S sang T^* sẽ có khả năng gia nhập vào đồ thị cân bằng G_f. Ta sẽ tăng λ đến khi có thêm ít nhất một cạnh mới gia nhập đồ thị cân bằng. Cụ thể, chỳng ta cú kết quả được trỡnh bày trong bổ đề sau đây.

- Giả sử f là phép gán nhãn chấp nhận được. Ta định nghĩa lân cận của đỉnh $u \in V$ và tập $S \subseteq V$ là các tập sau

$$N_f(u) = \{v : (u, v) \in E_f\},$$

$$N_f(S) = \cup_{u \in S} N_f(u).$$

Điều chỉnh nhãn

- **Bổ đề:** Giả sử $S \subseteq X$ và $T = N_f(S) \neq Y$. Đặt

$$\lambda = \min \{ f(x) + f(y) - w(x, y): x \in S,\ y \notin T \}$$
$$= f(x_0) + f(y_0) - w(x_0, y_0) \text{ với } x_0 \in S, y_0 \notin T$$

và

$$f'(v) = \begin{cases} f(v) - \lambda, & v \in S \\ f(v) + \lambda, & v \in T \\ f(v), & v \notin S \cup T \end{cases}$$

Khi đó f' là phép gán nhãn chấp nhận được và

(i) Nếu $(x, y) \in E_f$ với $x \in S, y \in T$ thì $(x, y) \in E_{f'}$.

(ii) Nếu $(x, y) \in E_f$ với $x \notin S, y \notin T$ thì $(x, y) \in E_{f'}$.

(iii) $(x_0, y_0) \in E_{f'}$.

Chú ý: Nếu $x \in S, y \notin T$ thì từ định nghĩa T suy ra $(x, y) \notin E_f$. Do đó

$$f(x) + f(y) > w(x, y), \text{ với } x \in S, y \notin T.$$

Suy ra $\lambda > 0$.

Chứng minh bổ đề

- Theo định nghĩa f ta có:

$$f(x) + f(y) \geq w(x, y),\ \forall x \in X, y \in Y.$$

thêm vào đó

$$f(x) + f(y) = w(x, y),\ \forall\ (x, y) \in E_f\ .$$

- Xét cạnh (x, y), khi đó có 4 tình huống

1. $x \in S, y \in T$: Khi đó

$$f'(x) + f'(y) = f(x) - \lambda + f(y) + \lambda\ = f(x) + f(y) \geq w(x, y).$$

và nếu $(x, y) \in E_f$ thì $(x, y) \in E_{f'}$.

2. $x \notin S, y \notin T$: Khi đó

$$f'(x) + f'(y) = f(x) + f(y) \geq w(x, y)$$

và nếu $(x, y) \in E_f$ thì $(x, y) \in E_{f'}$.

3. $x \notin S, y \in T$:

$$f'(x) + f'(y) = f(x) + \lambda + f(y)\ \geq w(x, y) + \lambda \geq w(x, y).$$

Chứng minh bổ đề

4. $x \in S, y \notin T$:

Ta có:

$$f'(x) + f'(y) - w(x, y) = f(x) + f(y) - \lambda - w(x, y)$$
$$= [f(x) + f(y) - w(x, y)] - \lambda \geq 0$$

Suy ra

$$f'(x) + f'(y) \geq w(x, y).$$

- Cuối cùng do

$$f'(x_0) + f'(y_0) - w(x_0, y_0) = [f(x_0) + f(y_0) - w(x_0, y_0)] - \lambda$$
$$= \lambda - \lambda = 0.$$

Suy ra (x_0, y_0) là thuộc vào $E_{f'}$.

Bổ đề được chứng minh.

Điều chỉnh nhãn

- Kết thúc việc điều chỉnh nhãn theo qui tắc mô tả trong bổ đề có hai khả năng:
  - Nếu cạnh mới gia nhập đồ thị cân bằng (cạnh (x_0, y_0) trong bổ đề) giúp ta thăm được một đỉnh không tự do $y_0 \in T^*$ thì từ y_0 ta sẽ thăm được một đỉnh được ghép với nó trong cặp ghép $x \in S^*$, và cả hai đỉnh này được bổ sung vào S và T tương ứng, và như vậy việc tìm kiếm đường tăng sẽ được tiếp tục mở rộng.
  - Nếu cạnh mới gia nhập đồ thị cân bằng cho phép thăm được một đỉnh tự do $y_0 \in T^*$ thì ta tìm được đường tăng cặp ghép, và kết thúc một pha điều chỉnh nhãn.

Kuhn-Munkres Algorithm

Bắt đầu từ phép gán nhãn f và cặp ghépM trong G_f

while (M chưa là cặp ghép đầy đủ)

while (Không tìm đường tăng cặp ghép M trong G_f)

Điều chỉnh f thành f' sao cho $E_f \subset E_{f'}$

Tăng cặp ghép M

Điều chỉnh nhãn

- Ta gọi ***một pha điều chỉnh là tất cả các lần sửa nhãn cần thiết để tăng được kích thước của cặp ghép M.***
- Vì sau mỗi pha điều chỉnh kích thước của cặp ghép tăng lên 1, nên ta phải thực hiện nhiều nhất n pha điều chỉnh.
- Trong mỗi pha điều chỉnh, do sau mỗi lần sửa nhãn có ít nhất hai đỉnh mới được bổ sung vào danh sách các đỉnh được thăm, nên ta phải thực hiện việc sửa nhãn không quá n lần.
- Mặt khác, trong thời gian $O(n^2)$ ta có thể xác định được cạnh nào từ S sang T^* là cạnh gia nhập đồ thị cân bằng (bằng việc duyệt hết các cạnh), nên mỗi lần sửa nhãn đòi hỏi thời gian $O(n^2)$.
- Từ đó suy ra đánh giá thời gian tính của thuật toán là $O(n^4)$.

Thuật toán

- **Bước 1:** Tìm một phép gán nhãn chấp nhận được f.

  Tìm cặp ghép cực đại M trong G_f.
- **Bước 2:** Nếu M là cặp ghép đầy đủ thì nó là cặp ghép lớn nhất cần tìm. Thuật toán kết thúc.
- **Bước 3:** Gọi S là tập các đỉnh tự do trong X. Thực hiện tìm kiếm từ các đỉnh trong S. Gọi T là tập các đỉnh được thăm trong quá trình tìm kiếm. Bổ sung các đỉnh trong X được thăm trong quá trình tìm kiếm vào S.
- **Bước 4:** Nếu tìm được đỉnh tự do $y_0 \in T$ (tìm được đường tăng) thì tăng cặp ghép M và quay lại bước 2.
- **Bước 5:** Tiến hành điều chỉnh nhãn f theo thủ tục mô tả trong bổ đề, ta sẽ bổ sung các đỉnh mới được thăm vào S và T. Quay lại bước 4.

Thuật toán

Bước 1. Tìm phép gán nhãn f và một cặp ghép M trong G_f.

Bước 2. Nếu M là đầy đủ, thì M là tối ưu và thuật toán kết thúc.

Ngược lại chọn một đỉnh tự do $x_0 \in X$. Đặt $S = \{x_0\}$, $T = \varnothing$.

Bước 3. Nếu $N_f(S) = T$, thì điều chỉnh nhãn (tất nhiên, ta có $N_f(S) \neq Y$)

$$\lambda = \min \{ f(x) + f(y) - w(x, y): \ x \in S, y \notin T \}$$

$$f'(v) = \begin{cases} f(v) - \lambda, & v \in S \\ f(v) + \lambda, & v \in T \\ f(v), & v \notin S \cup T \end{cases}$$

Bước 4. Nếu $N_f(S) \neq T$, tìm $y_0 \in N_f(S) - T$.

- Nếu y_0 là tự do thì $x_0 \to y_0$ là đường tăng. Tăng M và quay lại bước 2.
- Nếu y_0 được ghép với đỉnh z, thì ta mở rộng tìm kiếm:

  $S = S \cup \{z\}$, $T = T \cup \{y_0\}$.

  Quay lại bước 3.

Tăng hiệu quả

* Để có được thuật toán với đánh giá thời gian tính tốt hơn, vấn đề đặt ra là làm thế nào có thể tính được giá trị λ tại mỗi lần sửa nhãn của pha điều chỉnh một cách nhanh chóng.
* Ta xác định độ lệch của các cạnh theo công thức

$$slack(x, y) = f(x) + f(y) - w(x, y).$$

Tăng hiệu quả

- Khi đó

$$\lambda = \min \{slack(x,y): x \in S, y \in T^*\}$$

* Rõ ràng việc tính trực tiếp λ theo công thức đòi hỏi thời gian $O(n^2)$. Bây giờ, nếu với mỗi đỉnh trong T^* ta ghi nhận lại cạnh với độ lệch nhỏ nhất

$$slack(y) = \min \{slack(x,y): x \in S\}, \quad y \in T^*$$

thì λ có thể tính được sau thời gian $O(n)$:

$$\lambda = \min \{slack(y): y \in T^*\}$$

Tăng hiệu quả

- Việc tính giá trị độ lệch *slack*(y) đòi hỏi thời gian $O(n^2)$ ở đầu pha điều chỉnh.
- Tuy nhiên, trong một pha điều chỉnh:
  - Ở mỗi lần sửa nhãn ta có thể sửa lại tất cả các độ lệch trong thời gian $O(n)$, do chúng bị thay đổi cùng một giá trị (do nhãn của các đỉnh trong S giảm đồng loạt đi cùng một giá trị λ).
  - Khi một đỉnh x được chuyển từ S^* sang S ta cần tính lại các độ lệch của các đỉnh trong T^*, việc đó đòi hỏi thời gian $O(n)$. Tuy nhiên, sự kiện một đỉnh được chuyển từ S^* sang S chỉ xảy ra nhiều nhất n lần.
- Như vậy, mỗi pha điều chỉnh có thể cài đặt với thời gian $O(n^2)$. Do có không quá n pha điều chỉnh trong thuật toán, nên cách cài đặt này cho ta thuật toán với thời gian tính $O(n^3)$.

Ví dụ

- Xét bài toán với ma trận hiệu quả

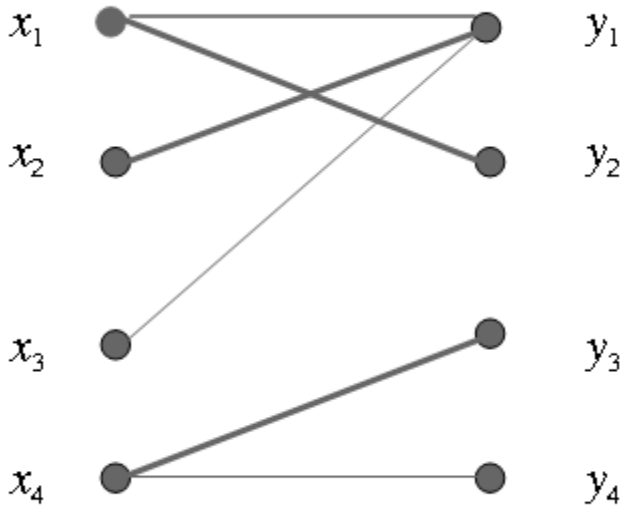
$$W = \begin{vmatrix} 4 & 4 & 1 & 3 \\ 3 & 2 & 2 & 1 \\ 5 & 4 & 4 & 3 \\ 1 & 1 & 2 & 2 \end{vmatrix}$$

Ví dụ

- Bắt đầu từ phép gán nhãn

$f(y)$ / $f(x)$	0	0	0	0
4	4	4	1	3
3	3	2	2	1
5	5	4	4	3
2	1	1	2	2

Đồ thị cân bằng G_f



Cặp ghép cực đại tìm được

$$M = \{(x_1,y_2), (x_2,y_1), (x_4, y_3)\}.$$

Tìm kiếm theo chiều rộng bắt đầu từ đỉnh tự do x_3 ta có

$$S = \{x_3, x_2\}, \quad T = \{y_1\}$$

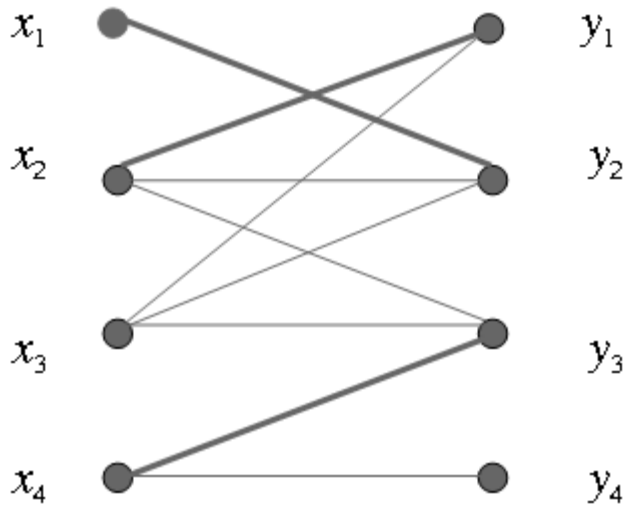
Ví dụ

- Tính

  $\lambda = \min \{f(x)+f(y)-w(x,y): x \in \{x_2, x_3\}, y \in \{y_2, y_3, y_4\}\} = 1.$

* Tiến hành sửa nhãn, ta đi đến phép gán nhãn mới

$f(y)$ / $f(x)$	1	0	0	0
4	4	4	1	3
2	3	2	2	1
4	5	4	4	3
2	1	1	2	2



Ví dụ

* Theo đường tăng cặp ghép

$$x_3, y_3, x_4, y_4$$

ta tăng cặp ghép M thành cặp ghép đầy đủ

$$M = \{(x_1,y_2), (x_2,y_1), (x_3,y_3), (x_4,y_4)\},$$

đồng thời là cặp ghép tối ưu với trọng lượng

$$w(M) = 4 + 3 + 4 + 2 = 13.$$

Cài đặt trên Pascal

```
const maxn = 170;

type    data1=array [1..maxn,1..maxn] of integer;
        data2=array [1..2*maxn] of integer;
        data3=array [1..2*maxn] of longint;

var   c: data1;
      px, py, q, queue: data2;
      a, b, f: data3;
      n, n2, k, u, z: integer;
```

Khởi tạo

procedure init;

```
var i, j: integer;
begin
     n2:= n+n; fillchar(f,sizeof(f),0);
     for i:=1 to n do
      for j:=1 to n do
       if f[i]<c(i,j) then f[i]:=c(i,j);
     k:=0;
     fillchar(px,sizeof(px),0); fillchar(py,sizeof(py),0);
     for i:=1 to n do
      for j:=1 to n do
       if (py[j]=0) and (f[i]+f[j+n]=c(i,j)) then
       begin
          px[i]:=j; py[j]:=i;  inc(k);
          break;
       end;
end;
```

Tìm đường tăng

```
function FoundIncPath: boolean;
var dq, cq, v, w: integer;
begin
   fillchar(q,sizeof(q),0);
   dq:=1; cq:=1; queue[dq]:=u; q[u]:=u;
   while dq<=cq do
   begin
      v:=queue[dq]; inc(dq);
      if v<=n then
      begin
         for w:=n+1 to n2 do
          if (f[v]+f[w]=c(v,w-n)) and (q[w]=0) then
          begin  inc(cq); queue[cq]:=w; q[w]:=v; end;
      end  else
      if (py[v-n]=0) then begin  FoundIncPath:=true;z:=v;exit; end
      else  begin w:=py[v-n];  inc(cq); queue[cq]:=w; q[w]:=v; end;
   end;
   FoundIncPath:=false;
end;
```

Tìm đỉnh tự do

```
function FreeNodeFound :boolean;
var i:integer;
begin
   for i:=1 to n do
      if px[i]=0 then
      begin
          u:=i;
          FreeNodeFound:=true;
          exit;
      end;
  FreeNodeFound :=false;
end;
```

Tăng cặp ghép và Sửa nhãn

```
procedure Tangcapghep;
var i, j: integer;
    ok: boolean;
begin
   j:=z; ok:=true;
   while j<>u do
   begin
      i:=q[j];
      if ok then
      begin
         px[i]:=j-n;
         py[j-n]:=i;
      end;
      j:=i;
      ok:= not ok;
   end;
   inc(k);
end;
```

```
procedure Suanhan;
var i, j: integer;
    d: longint;
begin
     d:= maxlongint;
     for i:=1 to n do
        if q[i]>0 then
           for j:=n+1 to n2 do
              if q[j]=0 then
                 if d>longint(f[i]+f[j]-c(i,j-n)) then
                       d:=longint(f[i]+f[j]-c(i,j-n));
     for i:=1 to n do
        if q[i]>0 then dec(f[i],d);
     for j:=n+1 to n2 do
        if q[j]>0 then inc(f[j],d);
end;
```

Main Procedure

```
procedure Solve;
begin
    init;
    while FreeNodeFound do
    begin
        while not FoundIncPath do Suanhan;
        Tangcapghep;
    end;
end;
```

Questions?