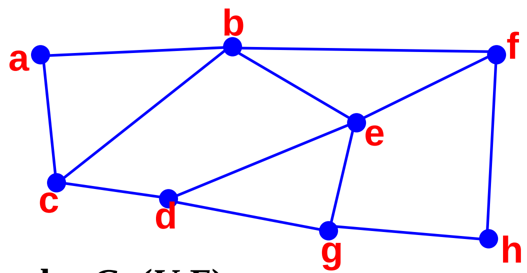
Unit 7
Elementary Graph Algorithms

T.H. Cormen et al., "Introduction to Algorithms", 3rd ed., Chapter 22 & Appendex B4, B5

無向圖

- What is a (undirected) graph?



- A graph : $G=(V,E)$

  vertex set : V = {a, b, c, d, e, f, g, h}

  edge set : E = {ab, ac, bc, cd, de, be, bf, … }

  或是 E = {(a,b), (a,c), (b,c), … }

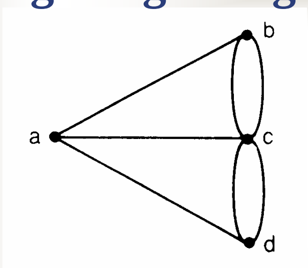
Problems & Modeling

Problems in real world

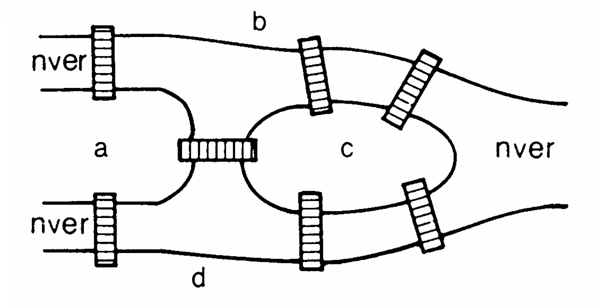
Graph ↓ **Modeling**

Graph Problems

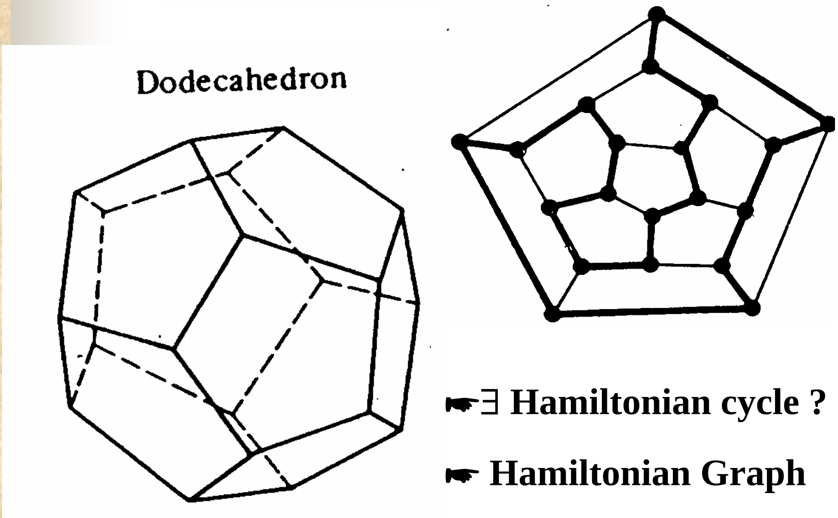
Königsberg Bridge Problem



- ∃ Euler circuit ?
  (一筆畫問題)
- Euler Graph

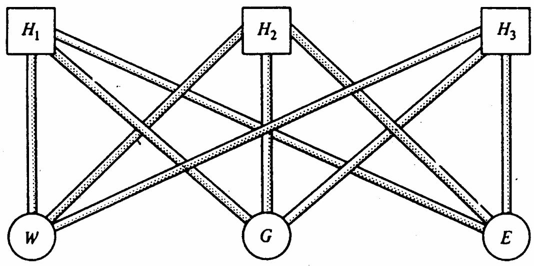


Hamiltonian Cycle Problem

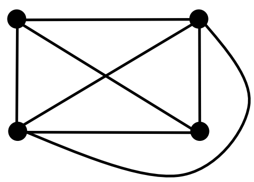


- ∃ Hamiltonian cycle ?
- Hamiltonian Graph

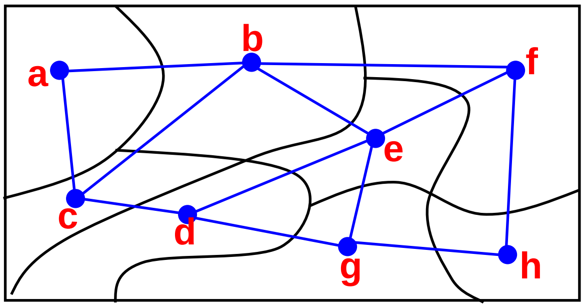
Utilities Problem



- ∃ planar embedding ?
- Planar Graph



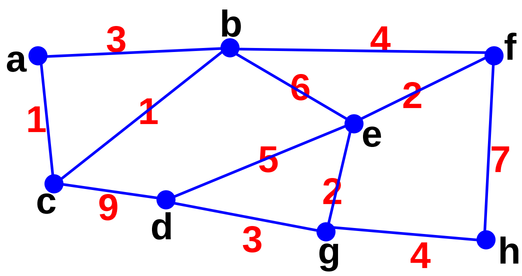
四 色 問 題



- ☛ **Are all maps 4-colorable?**

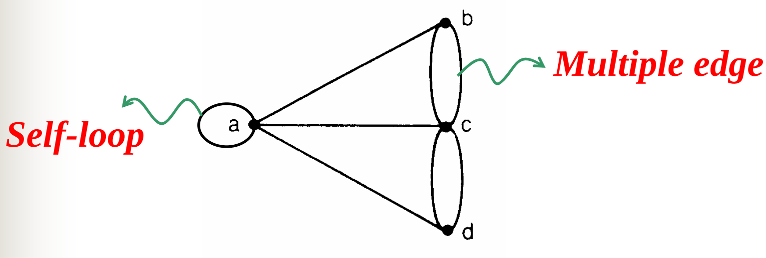
≅ **Are all planar graphs 4-colorable?**

Weighted Graphs



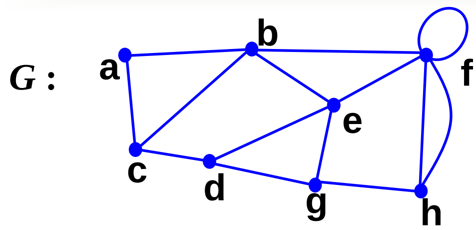
- ☛ **Shortest path, minimum spanning tree, maximum flow, ...**

Multiple Edges, Self-loops & Simple Graphs



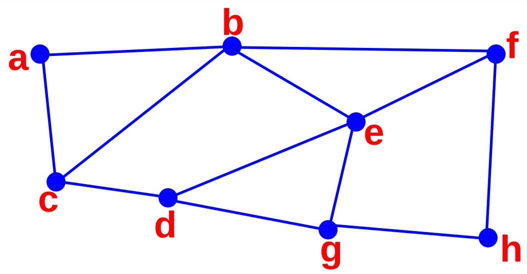
- ***Simple graph*** **: a graph without self-loops and multiple edges.**

Neighbors & Degrees



- ***Neighbors***: $N(v) = \{u \in V : uv \in E\}$ **Or** $Adj[v]$
  e.g. N(f) = {b, e, h, f}, N(a) = {b, c}
- ***Degrees***: **deg(*v*) = # edges incident to *v***
  (= |*N*(*v*)| if *G* is a simple graph)
  e.g. deg(f) = 6 **A property :** $\sum_{v \in V} \deg(v) = 2|E|$.

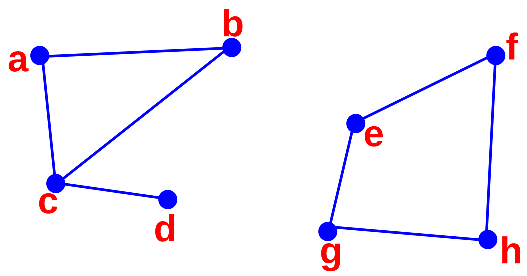
Paths, Trails, & Walks



- An a-f *path* : a c d e f
- An a-f *trail* : a b e g d e f
- An a-f *walk* : a b e g e b f
- 若頭尾相同則分別得 : *cycle*, *circuit*, *closed walk*.

本書分別稱為 simple path 與 path

Connectedness & Connected Components



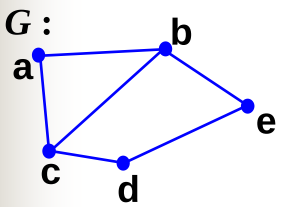
- A graph is ***connected*** if for any two vertices ***u***, ***v*** in the graph there exists a ***u-v*** path.
- ***Connected component*** : a maximal connected subgraph.

Adjacency Matrix

- A graph G can be represented by a matrix a[ ]:

  for $u, v \in V$, if $uv \in E$, then $a[u, v] = 1$

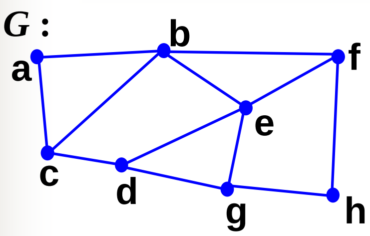
  otherwise $a[u, v] = 0$



	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	0	1
c	1	1	0	1	0
d	0	0	1	0	1
e	0	1	0	1	0

Adjacency matrix 為對稱的, 且對角線上的值均為 0 (for simple graph)

Adjacency Lists



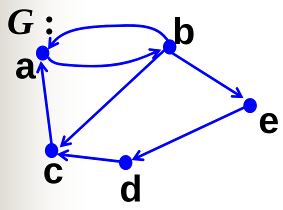
注意：每個 edge 被記錄兩次

a : b-c
b : a-c-e-f
c : a-b-d
⋮

We can use linked list or arrays to implement the adjacency lists, depending on the applications.

Directed Graphs

☛ 前面所討論的圖形均為無向圖 (undirected graph)

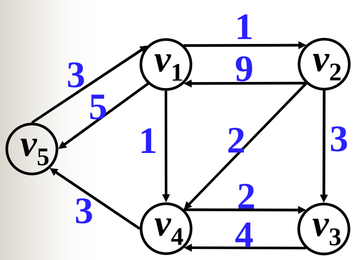


	a	b	c	d	e
a	0	1	0	0	0
b	1	0	1	0	1
c	1	0	0	0	0
d	0	0	1	0	0
e	0	0	0	1	0

a : b
b : a-c-e
c : a
⋮

在無向圖中每一個邊 *uv* ：
1.可將 *uv*, *vu* 視為同一個邊,
2.或是有 *uv* & *vu* 兩個方向,
∴很多有向圖演算法均適用於無向圖 (反過來不一定對).

Weighted Graphs Representations



	1	2	3	4	5
1	x	1	∞	1	5
2	9	x	3	2	∞
3	∞	∞	x	4	∞
4	∞	∞	2	x	3
5	3	∞	∞	∞	x

1 : ⟨2,1⟩ - ⟨5,5⟩ - ⟨4,1⟩
2 : ⟨4,2⟩ - ⟨1,9⟩ - ⟨3,3⟩
⋮

Each non-edge entry stores ∞. This is suitable for some problems such as shortest path problem. (Then each diagonal element x should store 0 or ∞?)

The comparison of the two structures

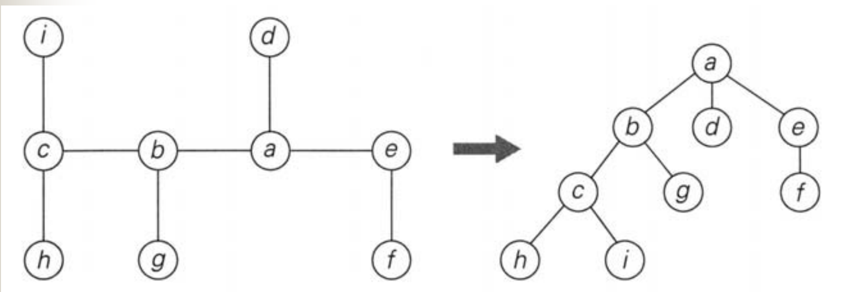
- Let $n = |V|$, $m = |E|$

	Space	Test $uv \in E$	Find $N(v)$
Adj-Matrix	$O(n^2)$	$O(1)$	$O(n)$
Adj-List	$O(n + m)$	$O(\|N(u)\|)$	$O(\|N(v)\|)$

$0 \leqq m \leqq n(n\text{-}1)/2$, for simple graph

Adjacency matrix 適合 dense graph
Adjacency lists 適合 sparse graph

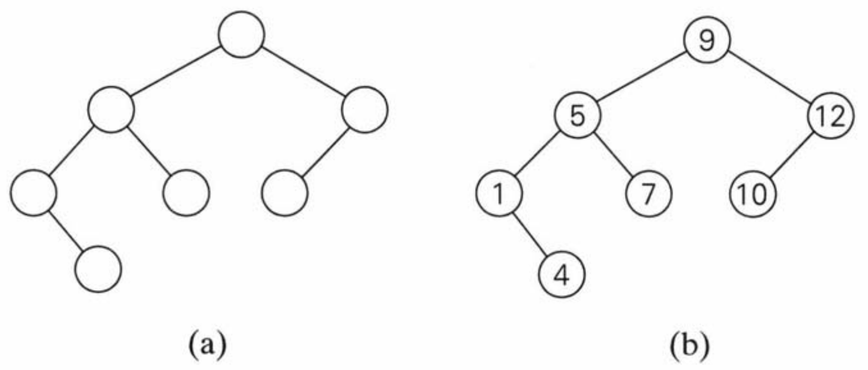
(Free) Tree & Rooted Tree



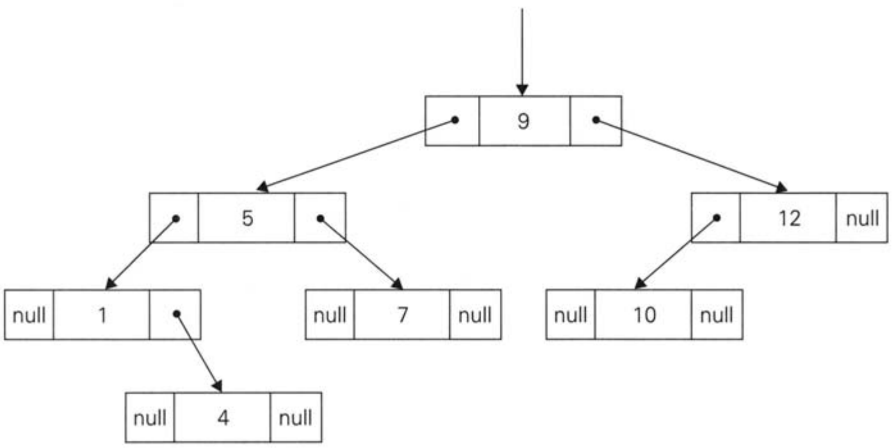
Characterizations of a tree:
1. Minimal connected graph
2. Connected & $|E| = |V| - 1$

⋮ ⋮

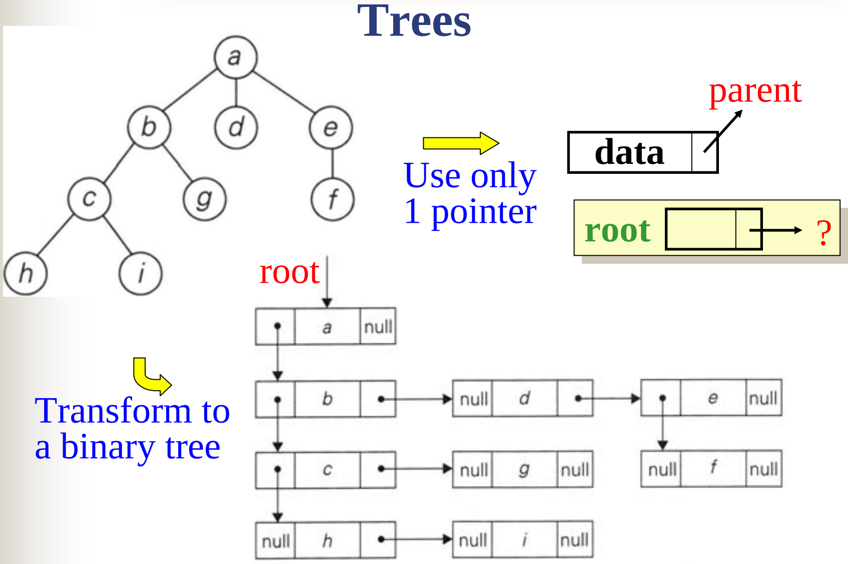
Binary Tree & Binary Search Tree



Standard Implementation of Binary Trees



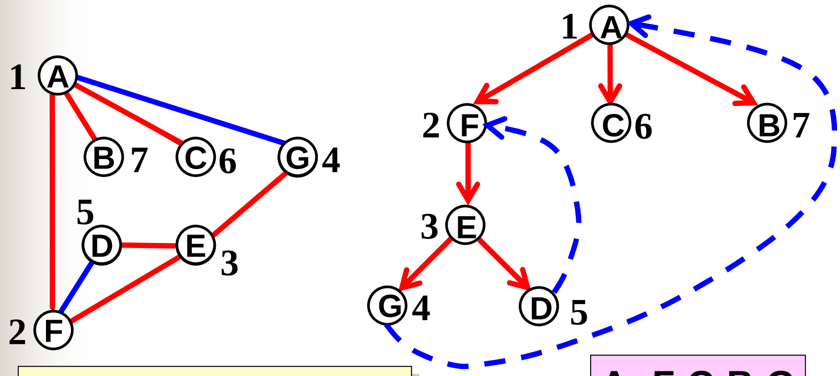
Space-Efficient Representations of Rooted Trees



Explore a Graph

- Visit every node and edge of a graph in a systematic and efficient way.
- Two basic methods:
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
- Can be used to solve some basic problems : reachablity, finding c.c. , cycle detecting, traversing mazes, …, and many complex ones.

DFS Trees (Forests)

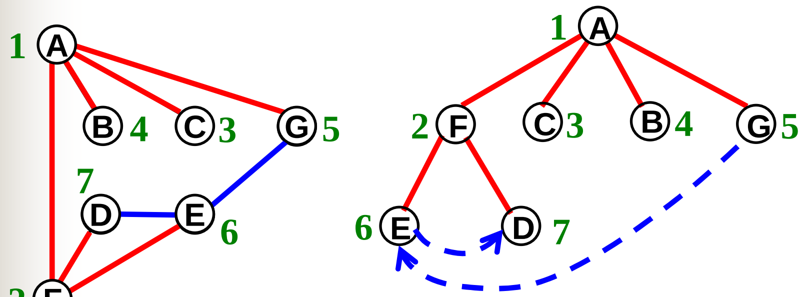


: Tree edges

: Non-tree edges or ***back edges*** (指到 ________)

A: F-C-B-G
F : E-A-D
E : F-G-D
⋮

BFS Trees (Forests)



: Non-Tree edge or ***cross edge*** (指到 ?)

注意 BFS Tree 的 levels

Analysis of DFS & BFS

- For undirected graphs: # search trees = # c.c.'s
- Other applications:

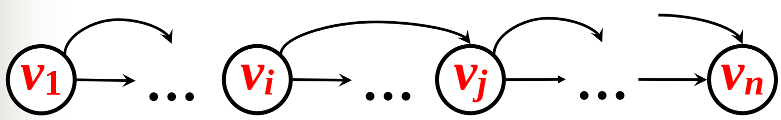
  BFS: single-source-shortest-path (unweighted case)

  DFS: biconnected components , strongly connected components, planarity testing, ….
- Execution time : ($n = |V|$, $m = |E|$)

  $O(n^2)$ (Adj-Matrix) or $O(n + m)$ (Adj-List)
- Space : $O(n)$.

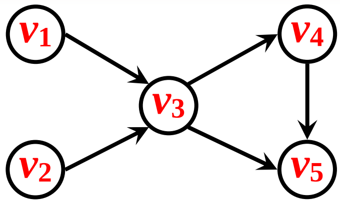
DAG & Topological Sorting (定義)

- Directed acyclic graph (dag) : A di-graph that contains no directed cycles.
- Topological sorting of a dag: list vertices of a di-graph $\boldsymbol{G}$ in such an order $\boldsymbol{v_1, v_2, \ldots, v_n}$ such that for each edge $\boldsymbol{v_i v_j}$ in $\boldsymbol{E(G)}$, we have $\boldsymbol{i < j}$.

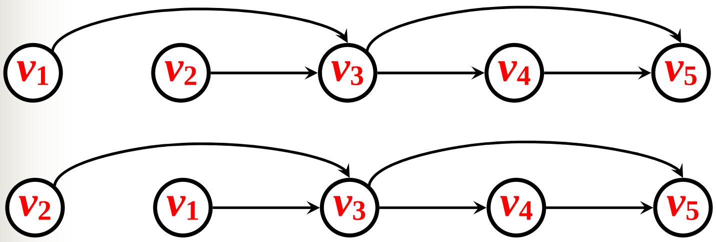


A di-graph is a dag iff. it has such an ordering.

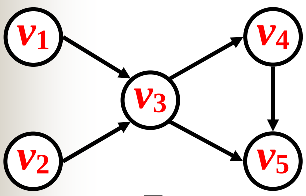
DAG & Topological Sorting (例)



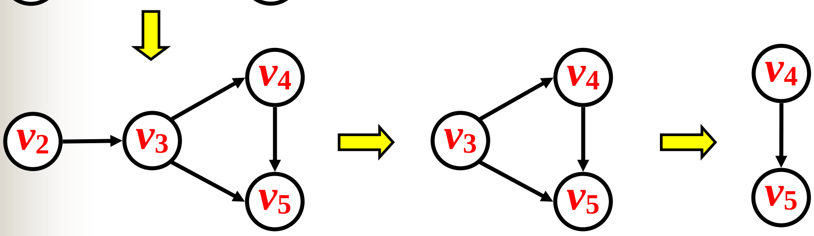
Topological sorted lists:



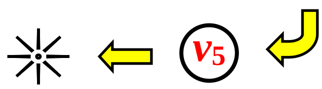
A Direct method for topological sorting



while ($V(G) \neq \varnothing$) do
    find a vertex v of in-degree 0;
    $G = G - v$;



Time: $O(|V|+|E|)$. How?



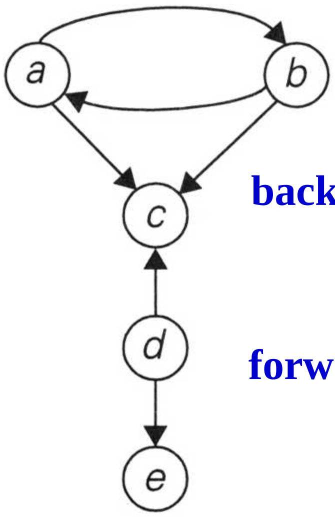
An implementation for the direct method

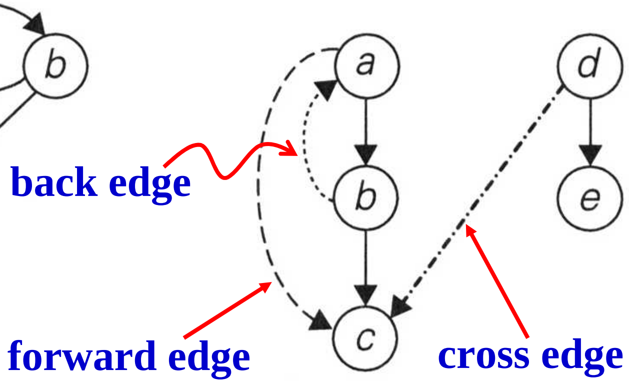
```
Topological_Sort(G)
// Assume that G is a dag

Compute id[v] for each vertex v;
for each vertex v do
   if( id[v] == 0) put v in Q;

while(Q is not empty) do
   remove a vertex v from Q;
   output v;
   for each vertex u in N(v) do
      if(--id[u] == 0) put u in Q;
```

Time: $O(|V|+|E|)$.

If G is not a dag, then …

DFS Forest of Di-Graph



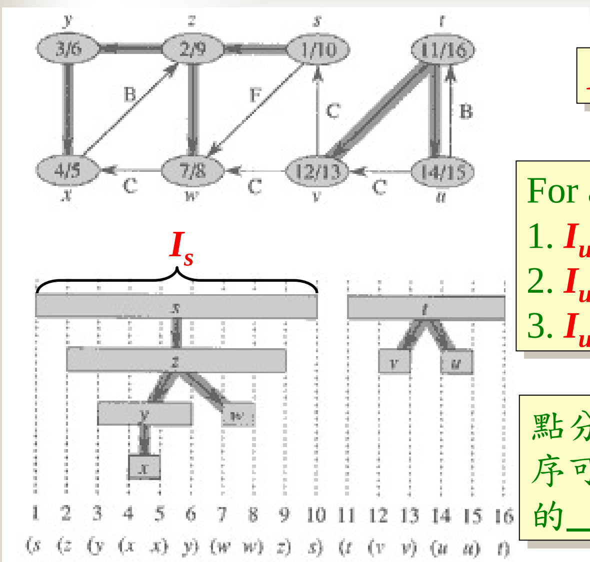


If G is a dag, then no ______ edges exist.

An Implementation for DFS

```
DFS(G)
Initially set c[u] ← White and p[u] ← Nil for each u ∈ V
time ← 0
for each u ∈ V do if c[u] = White then Visit(u)

Visit(u)
c[u] ← Gray
d[u] ← ++time
for each v ∈ Adj[u] do   // Explore edge uv
    if c[v] = White
    then p[v] ← u
            Visit(v)
c[u] ← Black
f[u] ← ++time   // 計算d[u], f[u]非必要視應用而定
```


Parenthesis Theorem



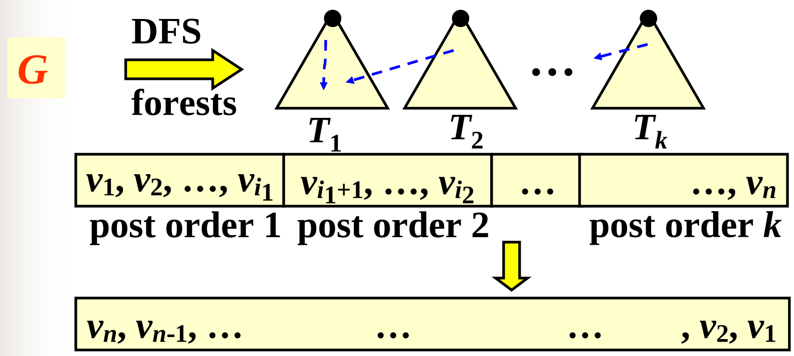
I_v = [d[v], f[v]]

For any u, v either
1. $I_u \cap I_v = \varnothing$ or
2. $I_u \subset I_v$, u 為 v 的子孫
3. $I_u \supset I_v$, v 為 u 的子孫

點分別依d[v], f[v]排序可分別得DFS For.的____&____ orders

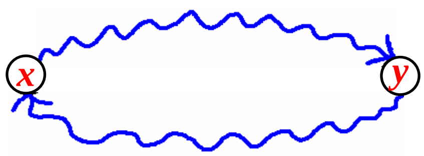
A DFS Method for Topological Sorting

1. Perform a DFS and output vertices in the order of computing $f[u]$. (see)
2. Reverse this order.



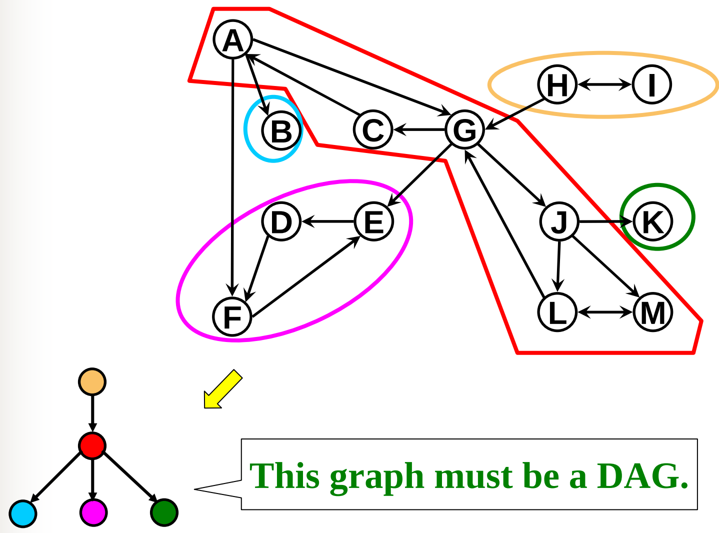
Strongly Connected Components

- Two vertices x and y are ***mutually accessible*** if there are paths from x to y and from y to x



- The relation of mutually accessible is an equivalence relation, i.e. the vertex set can be partitioned into several sets called ***strongly connected components*** (SCC) s.t. vertices in the same SCC are mutually accessible.

Strongly Connected Components (例)



This graph must be a DAG.

Observation 1

- An SCC with only outgoing (resp. incoming) edges is called a ***source*** (resp. ***sink***) component.
- If we start a DFS from vertex ***v*** in a sink component ***C***, then the set of vertices of the DFS tree rooted at ***v*** is exactly ***C***.
- If ***v*** is the last vertex in the post order of a DFS forest, then ***v*** must be in a_______ component.

Observation 2

- If we transpose the direction of each edge of a directed graph ***G***, then we obtain the ***transpose*** of ***G*** denoted as G^T.
- G^T has the same decomposition structure as that of ***G***. Moreover, each source (resp. sink) component of ***G*** becomes a sink (resp. source) component of G^T and vice versa.

An Algorithm for finding SCC's

- Combining the 2 observations, we have:

Strongly-Connected-Components(***G***)

1 Call DFS(***G***) to computing ***f***[***u***] for each $u \in V$.

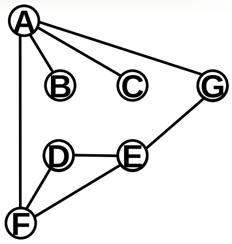
2 Compute G^T.

3 Call DFS(G^T) but consider the vertices in order of decreasing ***f***[***u***] computed in line 1.

4 Output the vertices of each tree in the DFS forest formed in line 3 as a separate SCC.

Another Implementation for DFS

```
public Vertex next( )
{
   u = stack.pop( );
   for each v ∈ Adj[u] :
      if v has not yet been reached {
         mark v as reached;
         push v onto stack;
      }// if
   return u;
}// algorithm for method next
```



A: F-C-B-G
F : E-A-D
E : F-G-D
⋮

Vertices returned by next( ): **A G E D B C F**

Discussions of the two implementations

- The second implementation (called ***depth-first iterator***) can be found in page 660 of the book: "DS and the Java Collections Framework" W.J. Collins, p.660, 2nd. ed. McGraw Hill, 2005.
- Both implementations use stacks and take $O(\ n + m\ )$ execution time.
- Most SE guys favor the second implementation.
- Is the second implementation still useful for solving those problems that can be solved by applying the first implementation?