Tarjan
連通分量系列

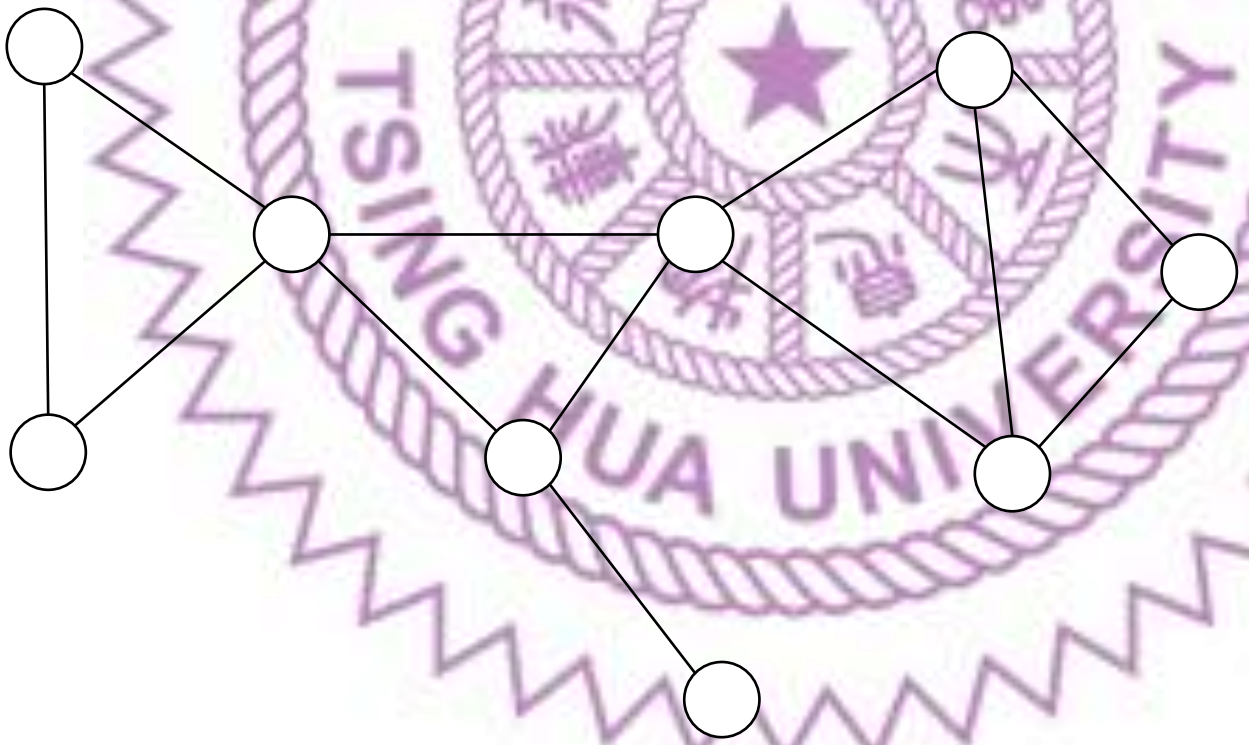
日月卦長

點雙連通分量

Bi-Connected Components (BCC)

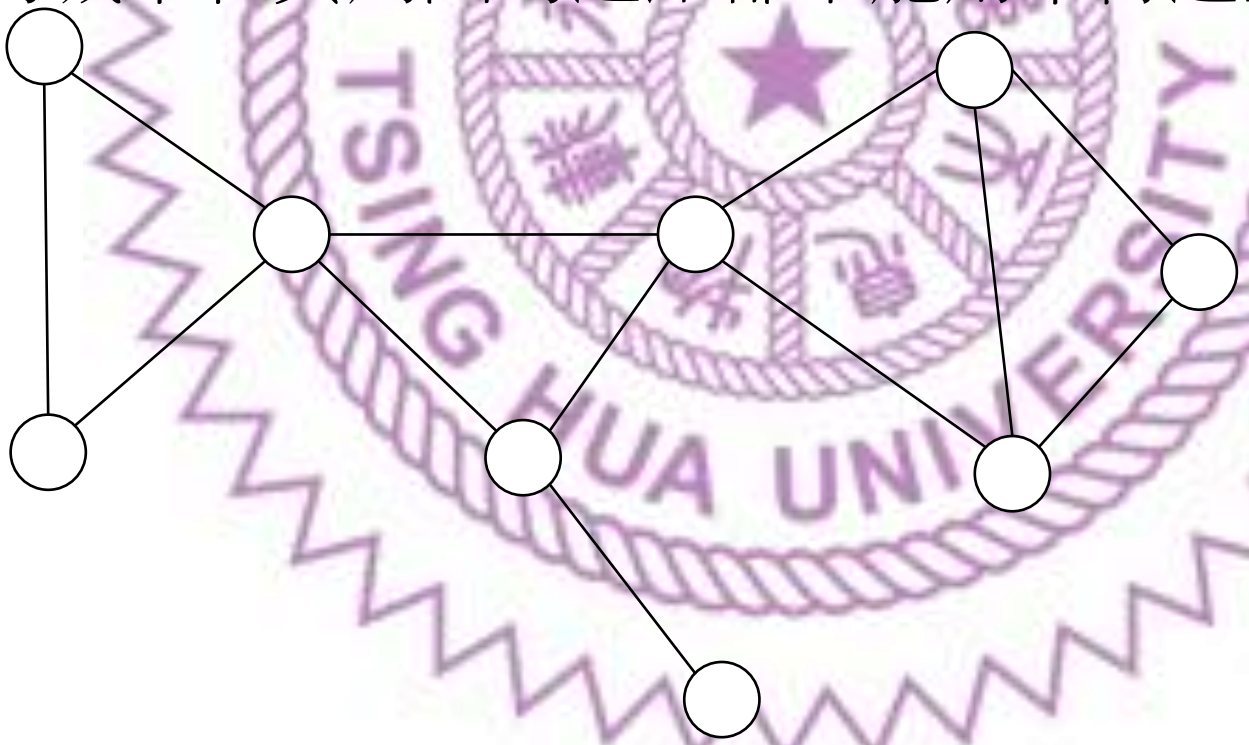
金坷垃運輸問題

- 身為聖地亞戈的負責人
  你的工作就是讓每座城市之間可以互相傳遞金坷垃



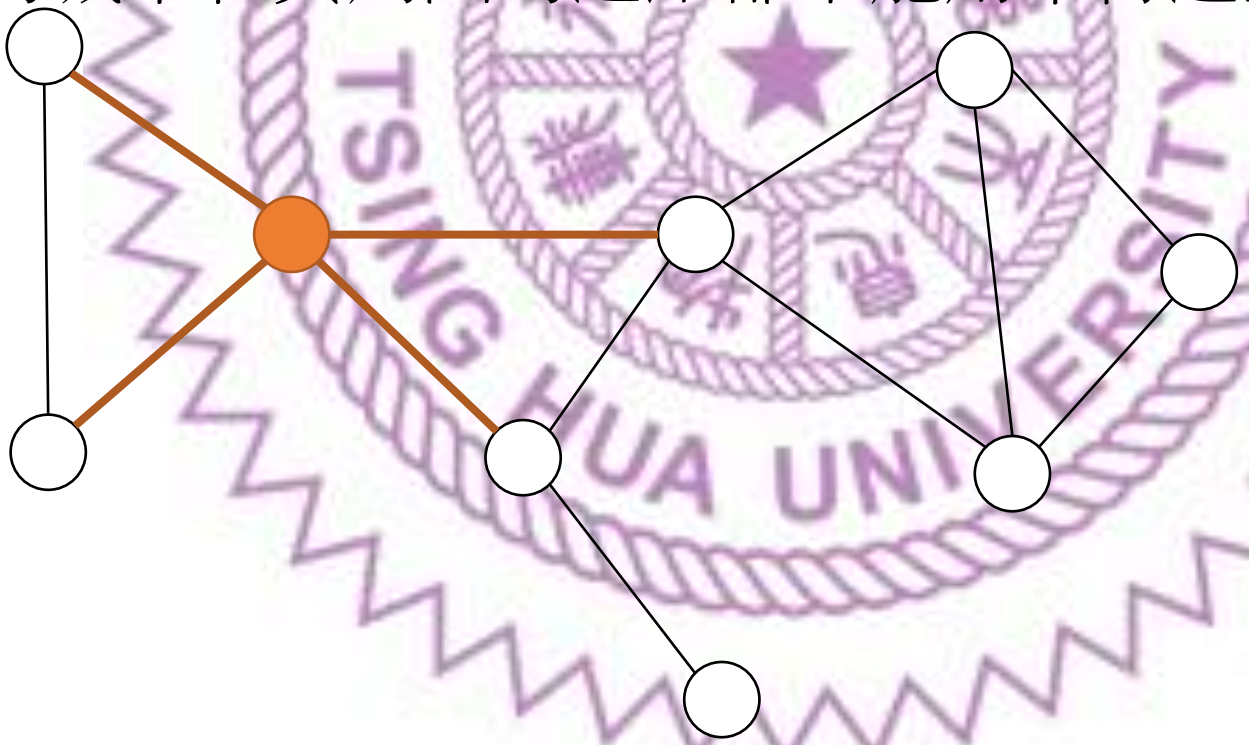
金坷垃運輸問題

- 現在你有一個危機
  有一個小日本鬼子打算佔領你的任意一座城市
- 被佔領的城市和其周圍的道路都不能用來傳遞金坷垃



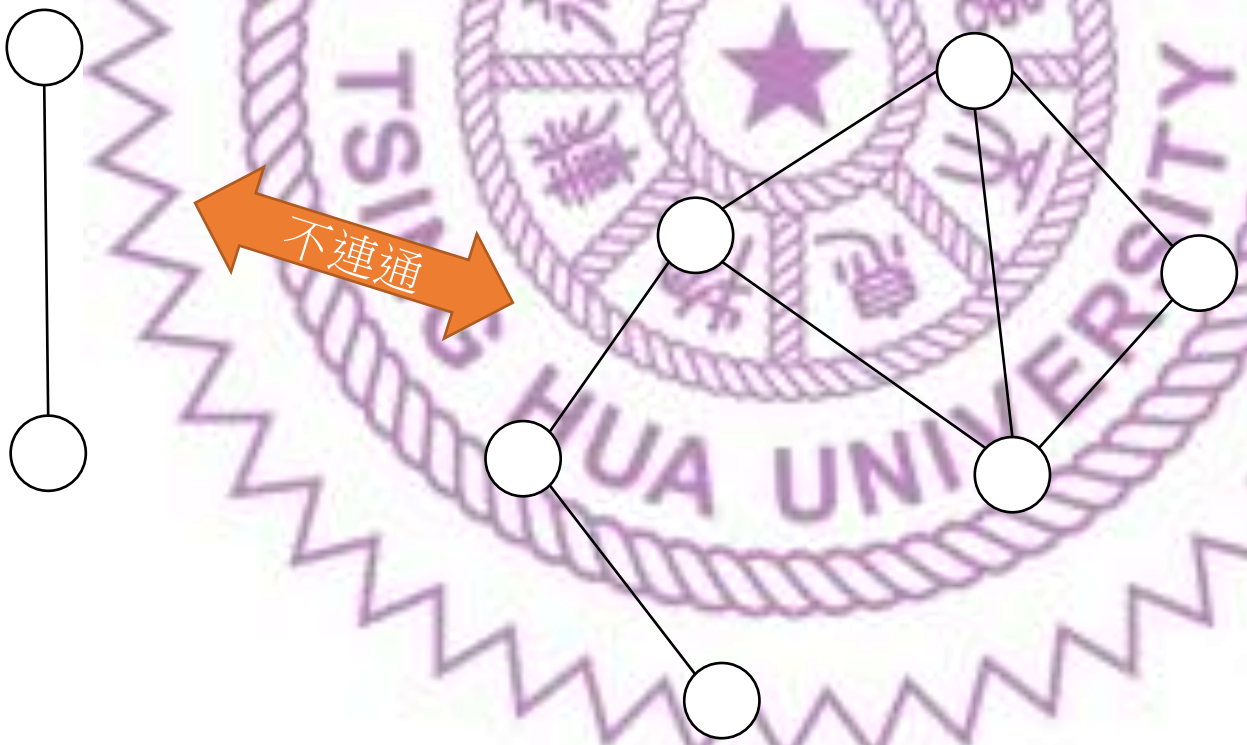
金坷垃運輸問題

- 現在你有一個危機
  有一個小日本鬼子打算佔領你的任意一座城市
- 被佔領的城市和其周圍的道路都不能用來傳遞金坷垃



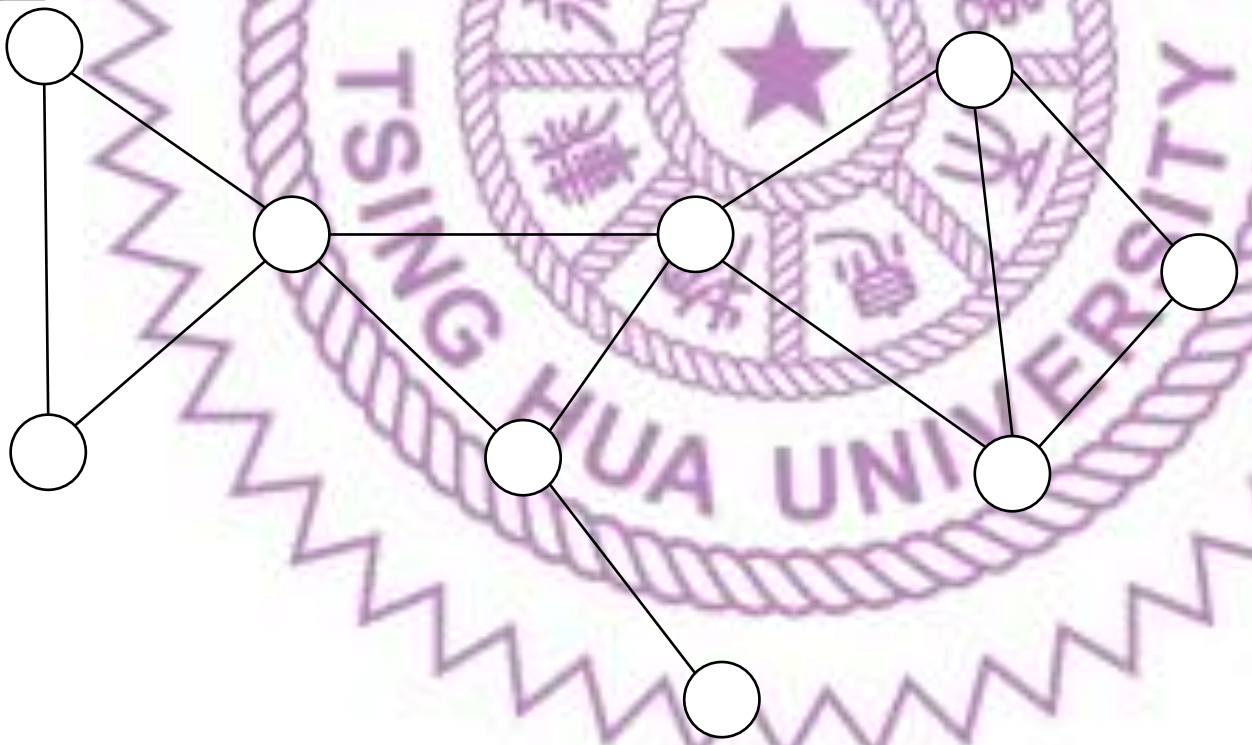
金坷垃運輸問題

- 這樣可能會導致有些城市之間無法互相傳遞金坷垃，平衡就會崩壞



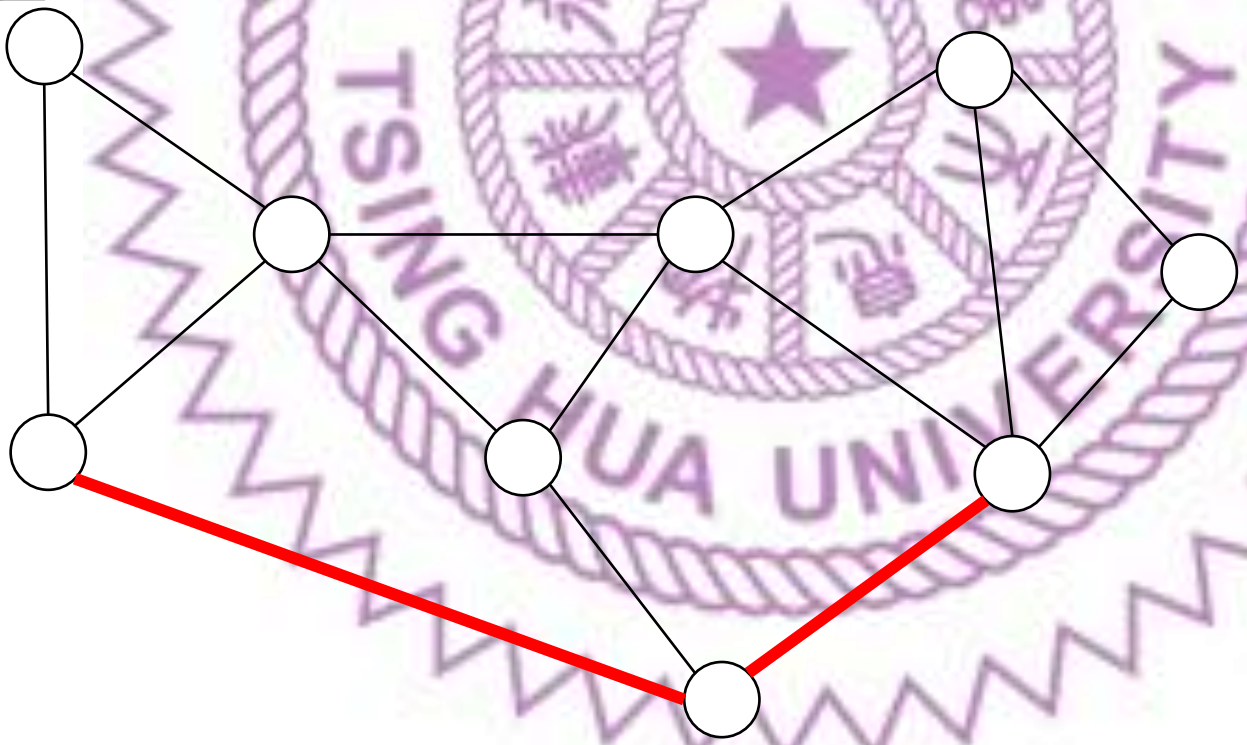
金坷垃運輸問題

- 你打算增加一些道路 (Edge) 來避免這種事情發生
- 請問你最少需要增加幾條路才能讓剩下的城市之間能繼續互相傳遞金坷垃？



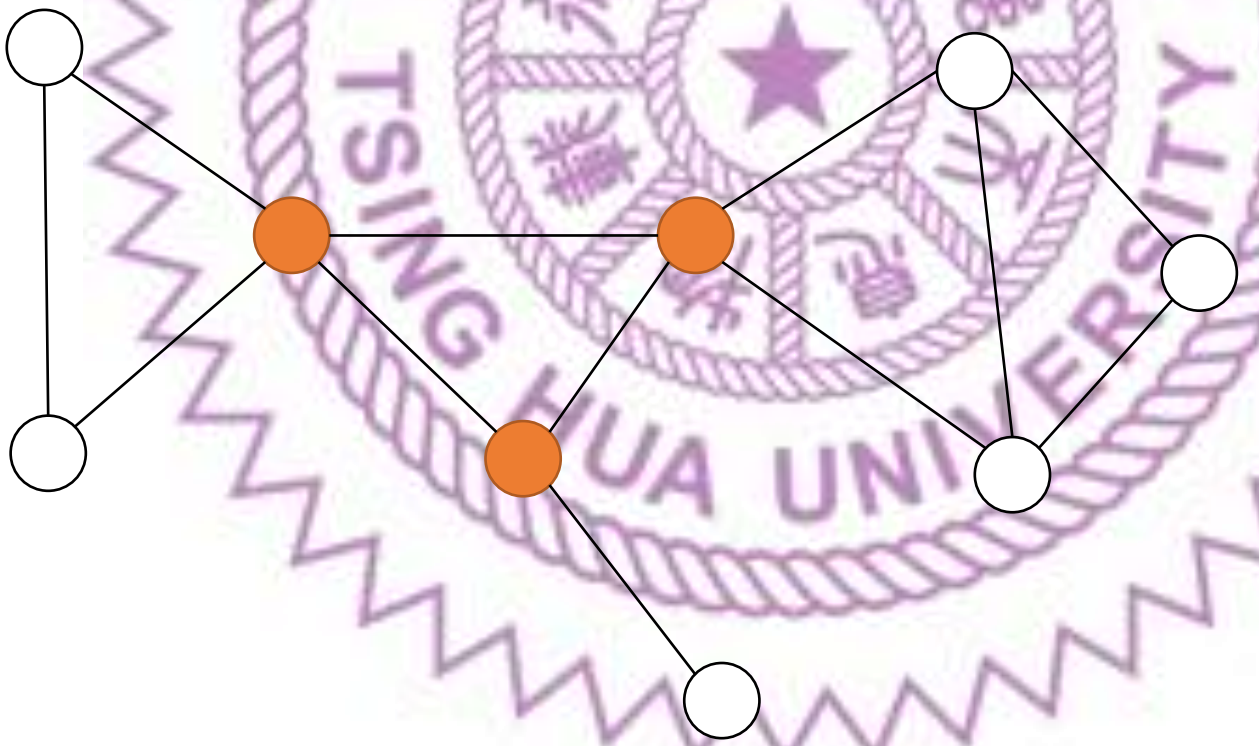
金坷垃運輸問題

- 你打算增加一些道路 (Edge) 來避免這種事情發生
- 請問你最少需要增加幾條路才能讓剩下的城市之間能繼續互相傳遞金坷垃？



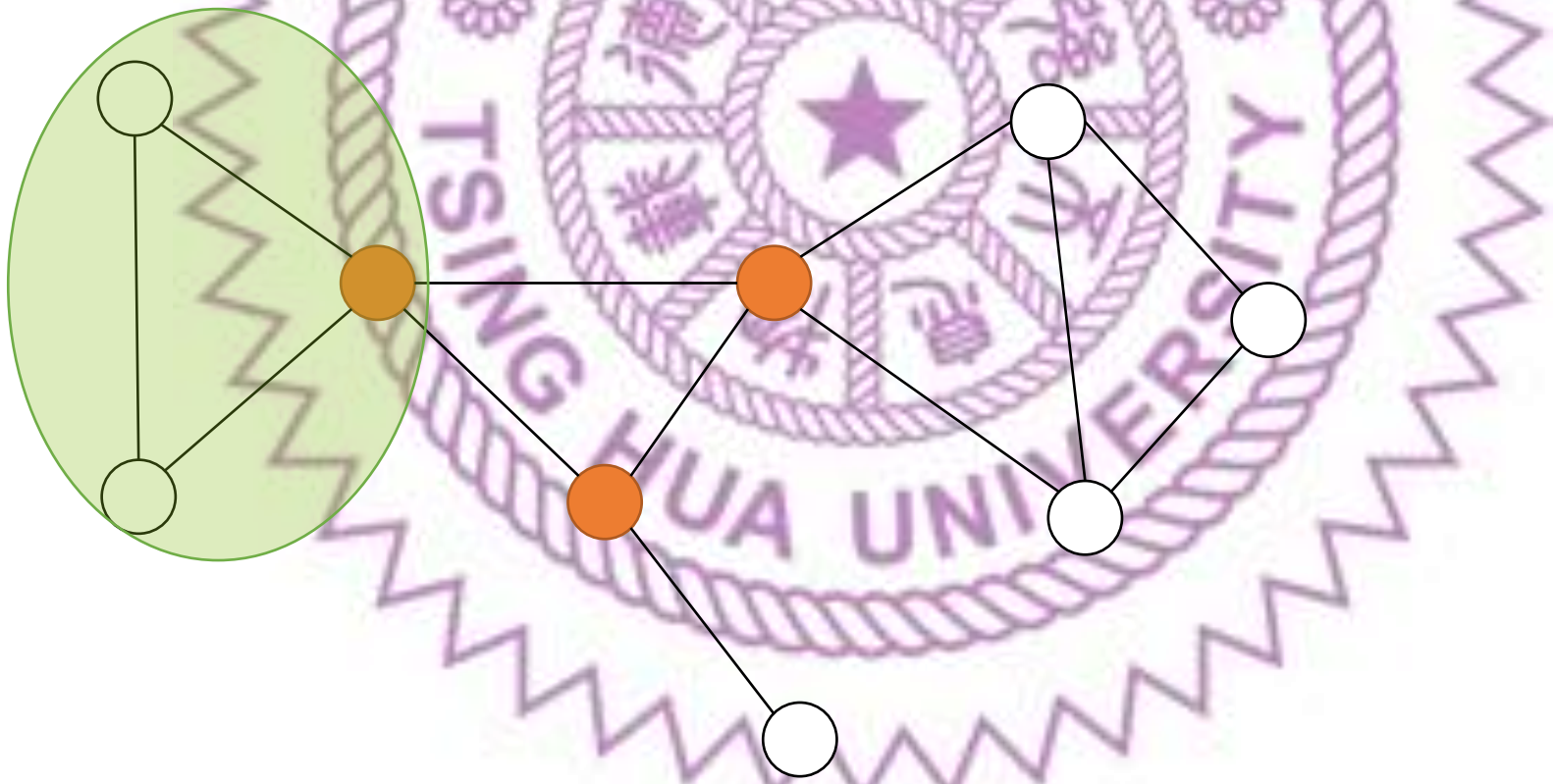
想法

- 先把割點 (cut vertex) 找出來
- 目標是要增加最少的 Edge 讓割點消失！



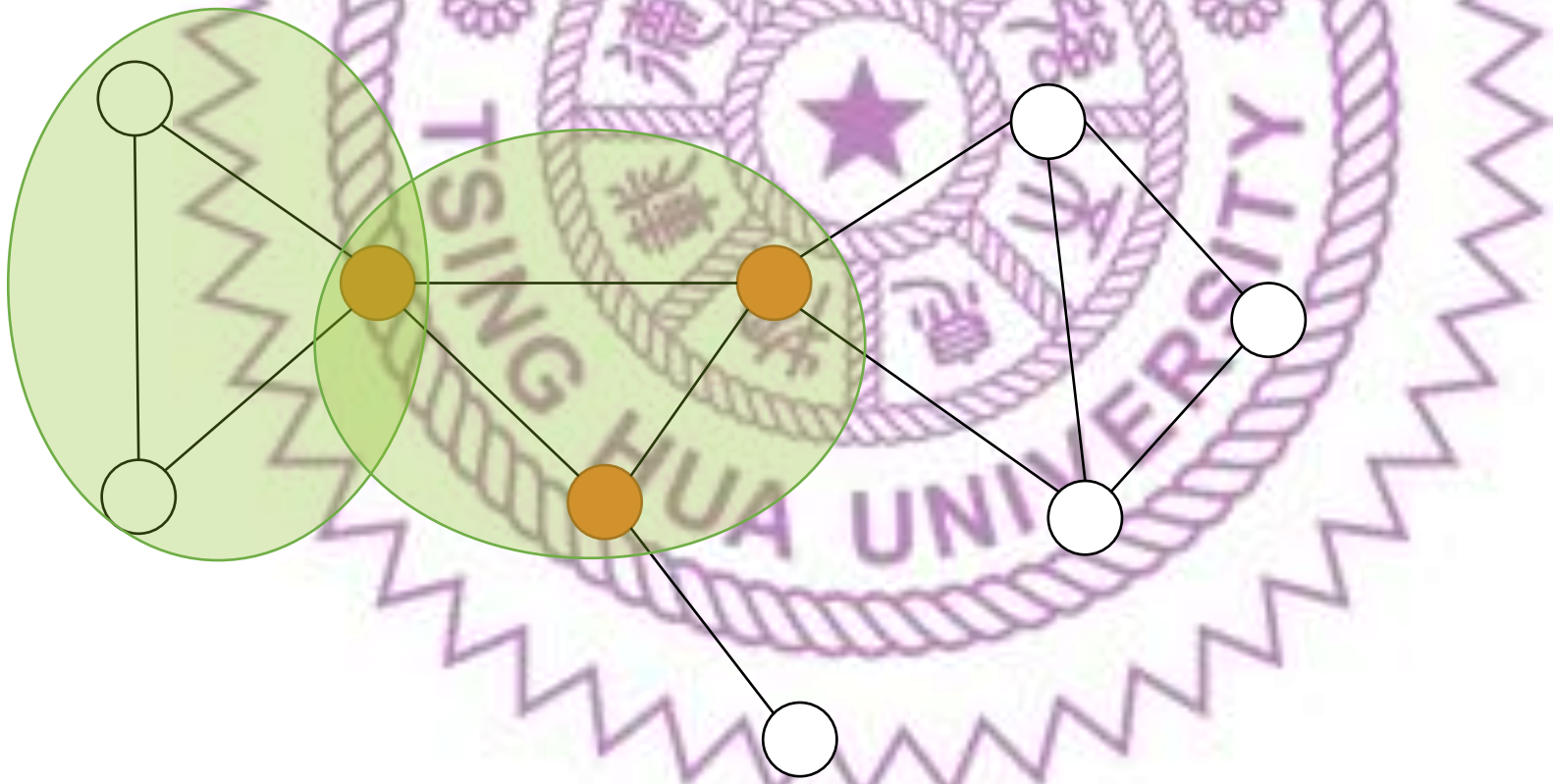
想法

- 可以發現可以將 Graph 分成一些區塊
  在這些區塊裡面增加 Edge 對減少割點沒有任何幫助



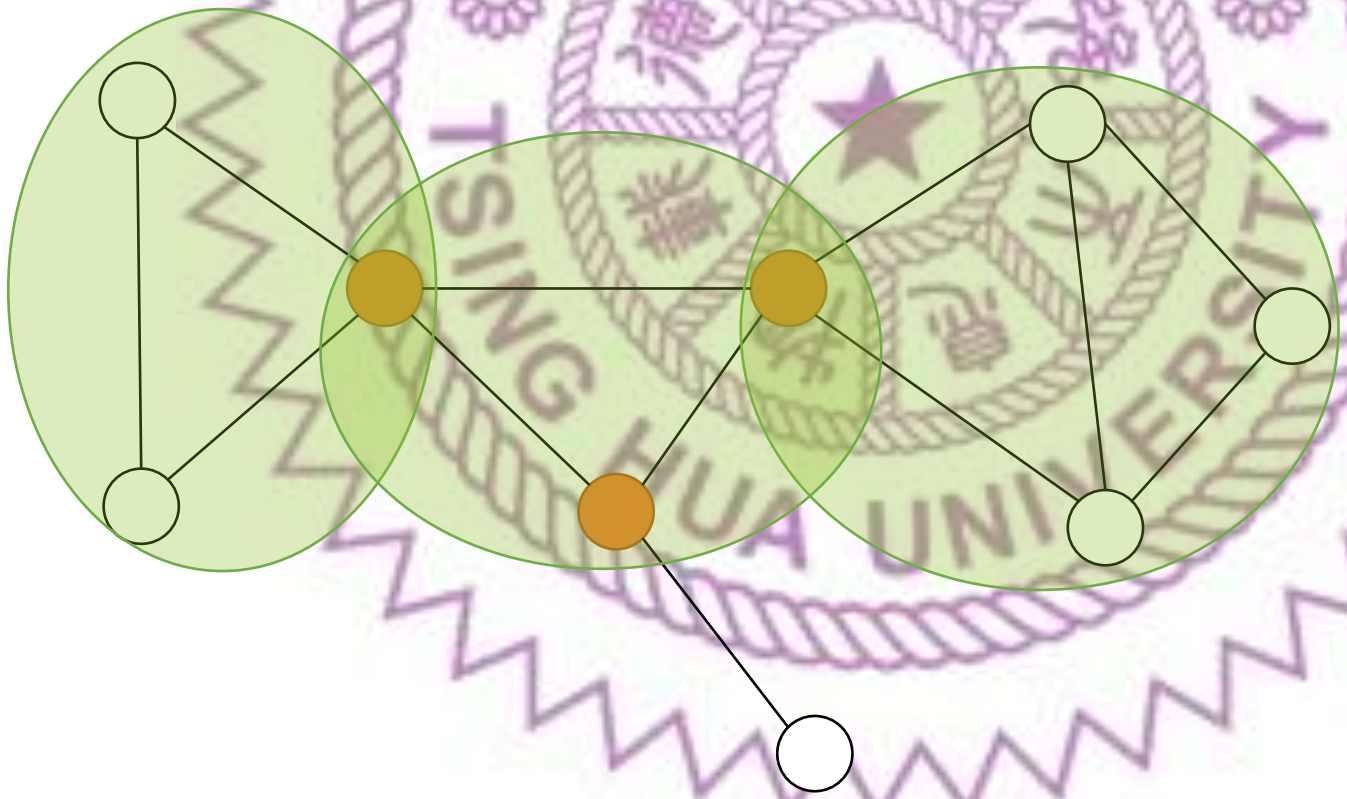
想法

- 可以發現可以將 Graph 分成一些區塊
  在這些區塊裡面增加 Edge 對減少割點沒有任何幫助



想法

- 可以發現可以將 Graph 分成一些區塊
  在這些區塊裡面增加 Edge 對減少割點沒有任何幫助



想法

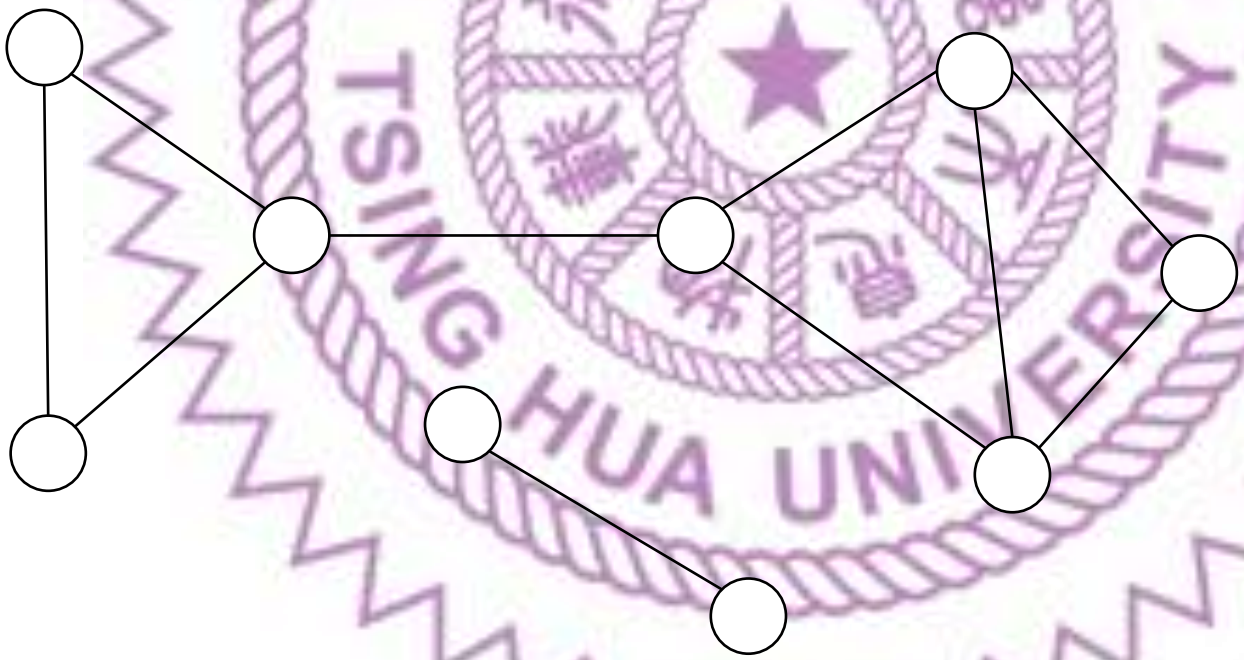
- 可以發現可以將 Graph 分成一些區塊
  在這些區塊裡面增加 Edge 對減少割點沒有任何幫助

Block

- 這樣的區塊我們稱其為 點雙連通分量(BCC) 或 Block
- 更嚴格的定義如下：
- Definition.
  A block of a graph G is a maximal connected subgraph of G that has no cut-vertex.

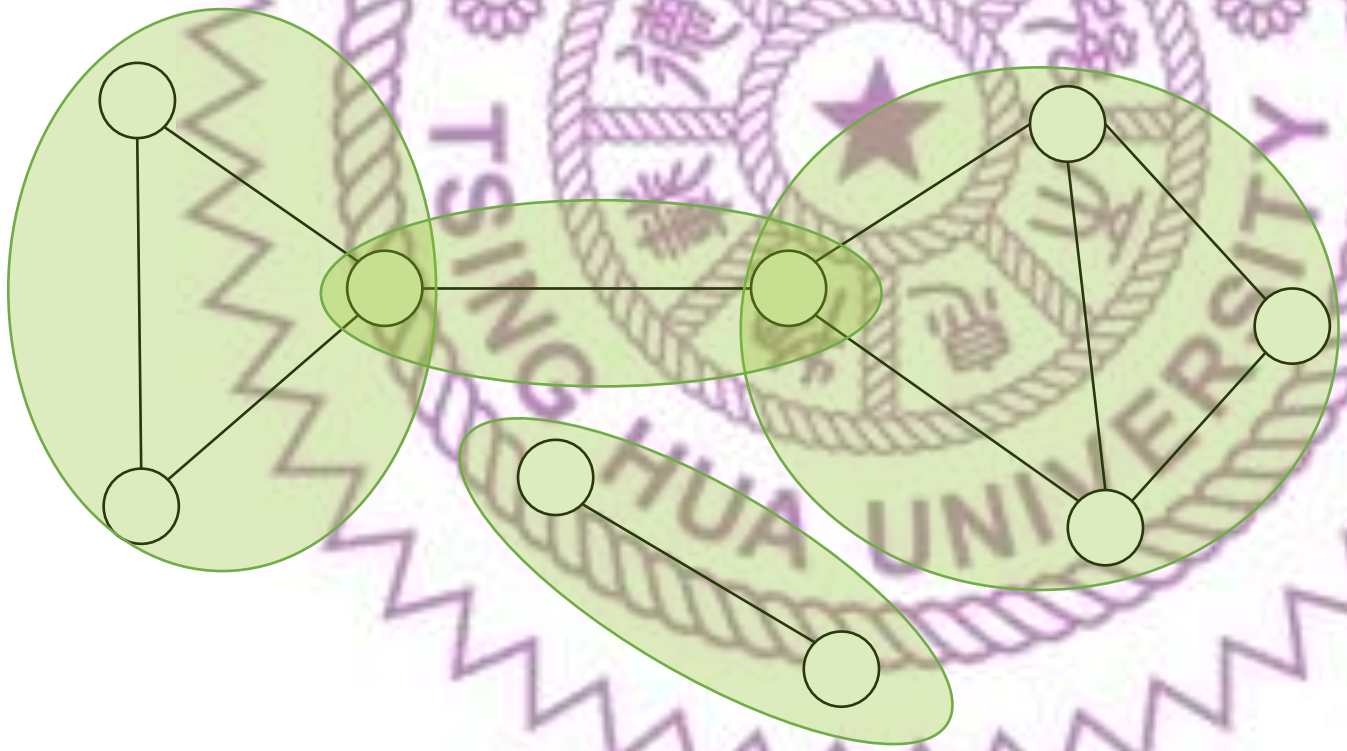
小練習

- 找出下面 graph 的所有 block



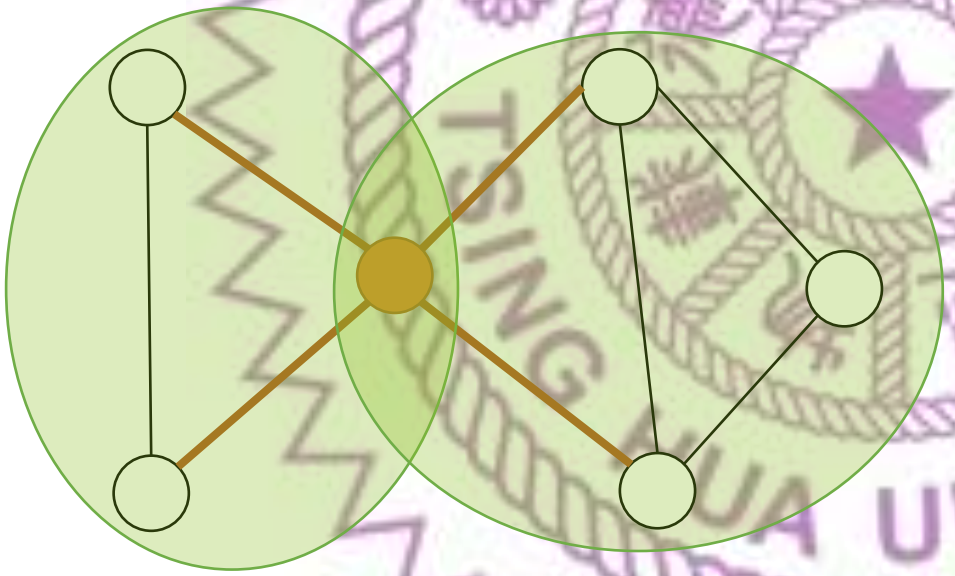
小練習

- 找出下面 graph 的所有 block

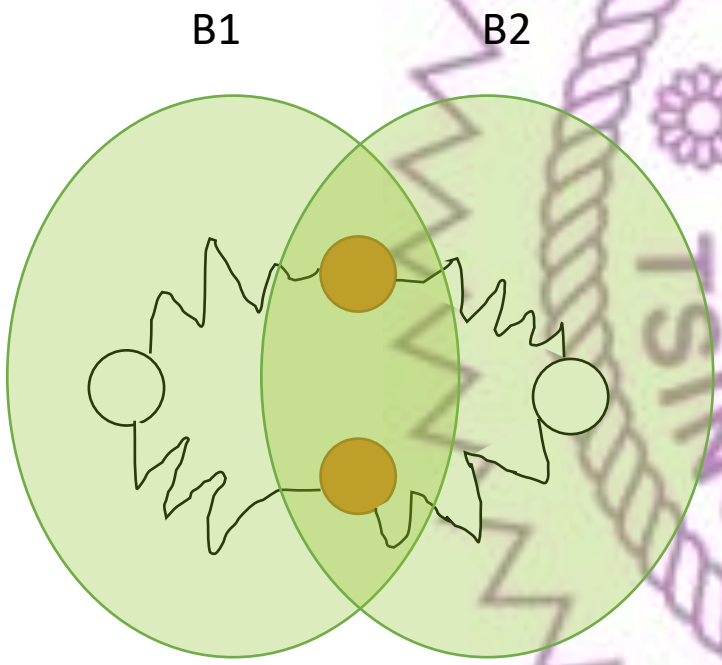


Block 性質

- 兩個 Block 之間最多只會共用一個 vertex

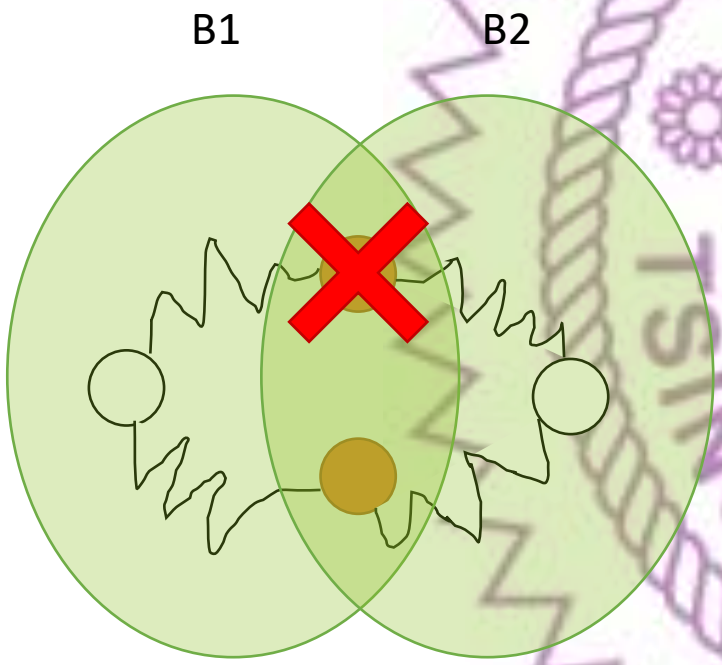


proof



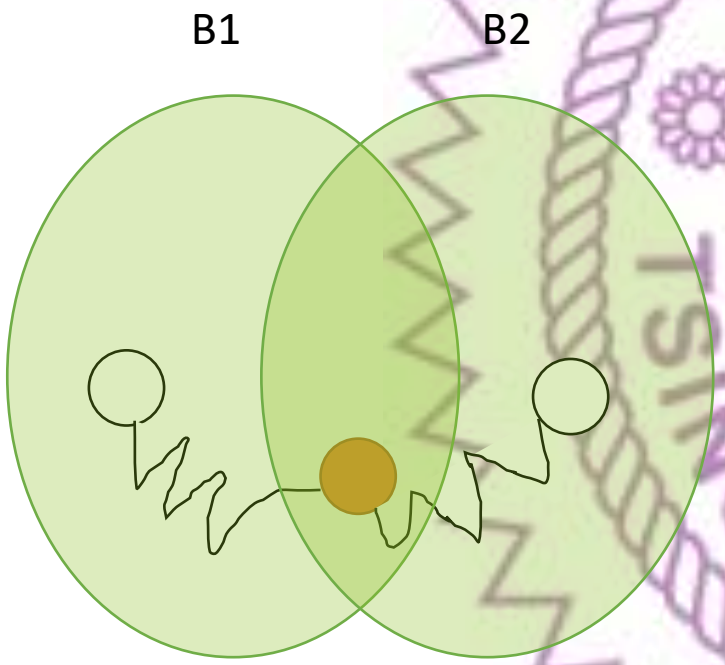
- 反證法
- 現在有兩個 block $B1, B2$
- 假設 $\|B1 \cap B2\| > 1$

proof



- 刪除 $B1 \cup B2$ 中任意一個 vertex p

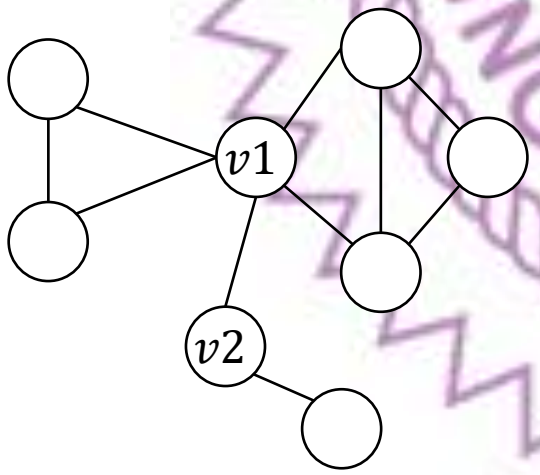
proof



- 刪除 $B1 \cup B2$ 中任意一個 vertex p
- 由於 $B1 - p$ 和 $B2 - p$ 都 connected
- 且 $(B1 \cap B2) - p \neq \emptyset$
- 得到 $B1 \cup B2$ no cut-vertex 與 block 的性質矛盾

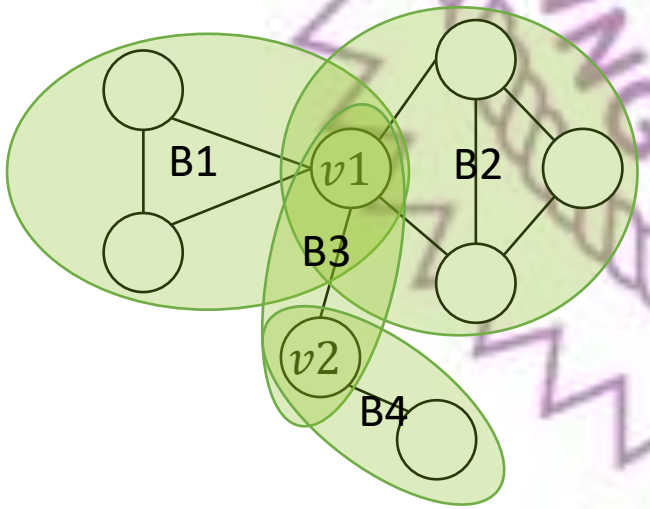
Block-cutpoint graph

- The block-cutpoint graph of a graph G is a bipartite graph H which one partite set consists of the cut-vertices of G, and the other has a vertex bi for each block Bi of G. We include (v, bi) as an edge of H if and only if $v \in Bi$



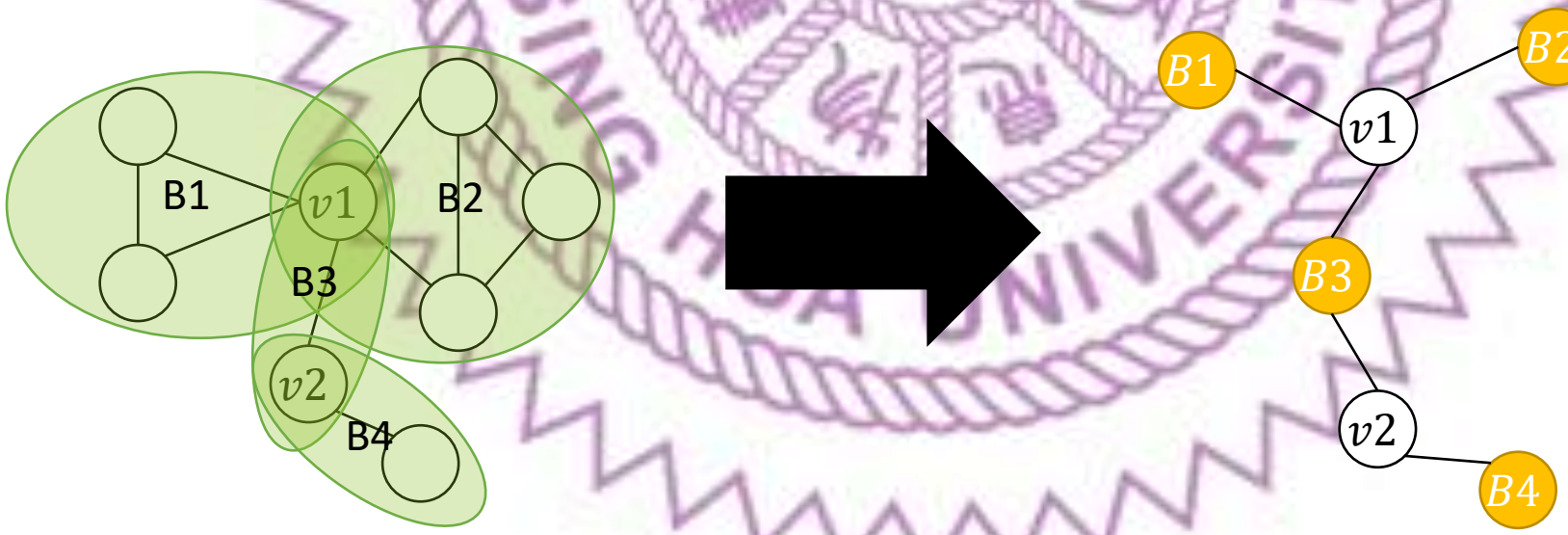
Block-cutpoint graph

- The block-cutpoint graph of a graph G is a bipartite graph H which one partite set consists of the cut-vertices of G, and the other has a vertex bi for each block Bi of G. We include (v, bi) as an edge of H if and only if $v \in Bi$



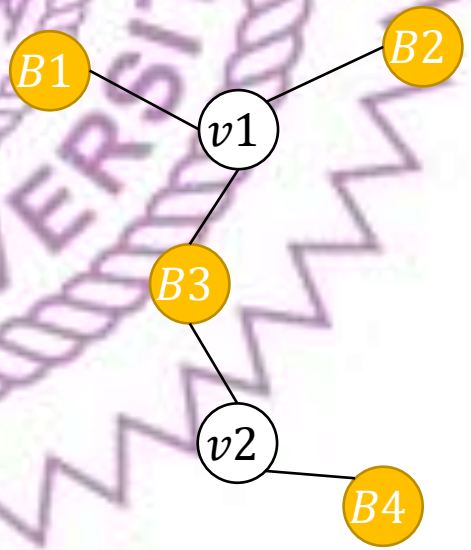
Block-cutpoint graph Tree

- The block-cutpoint graph of a graph G is a bipartite graph H which one partite set consists of the cut-vertices of G, and the other has a vertex bi for each block Bi of G. We include (v, bi) as an edge of H if and only if $v \in Bi$



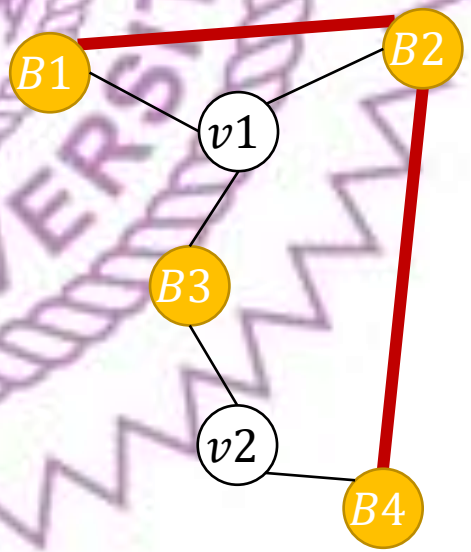
金坷垃運輸問題

- 這個問題變得更簡化：
  1.計算該 graph 的 block-cutpoint tree H
  2.問題 reduce 成增加最少的邊使得 H 中沒有割點
  3.會發現存在 $O(n)$ 的算法 (想一想怎麼做)



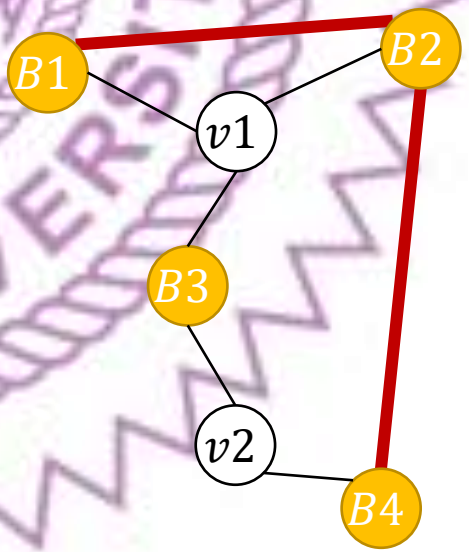
金坷垃運輸問題

- 這個問題變得更簡化：
  1.計算該 graph 的 block-cutpoint tree H
  2.問題 reduce 成增加最少的邊使得 H 中沒有割點
  3.會發現存在 $O(n)$ 的算法 (想一想怎麼做)



金坷垃運輸問題

- 這個問題變得更簡化：
  1.計算該 graph 的 block-cutpoint tree H
  2.問題 reduce 成增加最少的邊使得 H 中沒有割點
  3.會發現存在 $O(n)$ 的算法 (想一想怎麼做)



- 如何構造 block-cutpoint tree?
  - 顯然找出所有 block 就行了

暴力法

- 暴力檢查每個點是否為割點
- $O\big(V(E+V)\big)$

暴力法

- 暴力檢查每個點是否為割點
- $O\big(V(E+V)\big)$

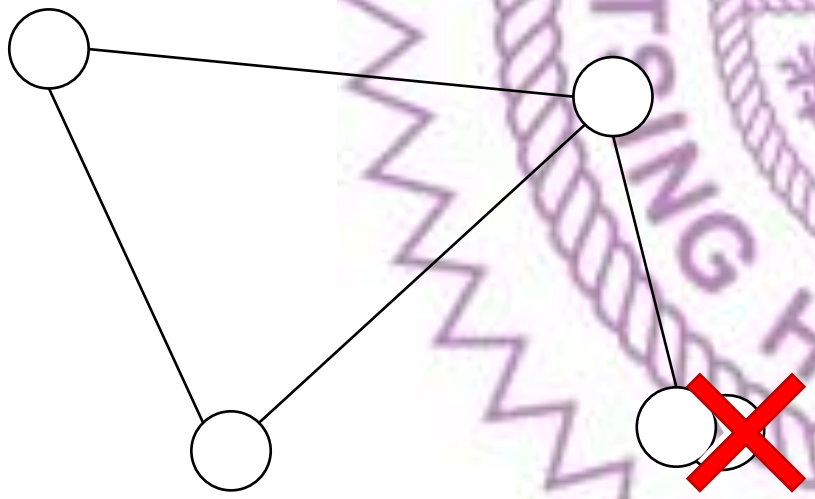
- 某位大師利用 Depth First Search (DFS) 在 $O(E+V)$時間找出所 block

Robert Endre Tarjan

- 這個人有得**圖靈獎**
- 發現很多圖論問題用一次 DFS 就能解決
- 很多演算法都是他的名子
  容易混淆

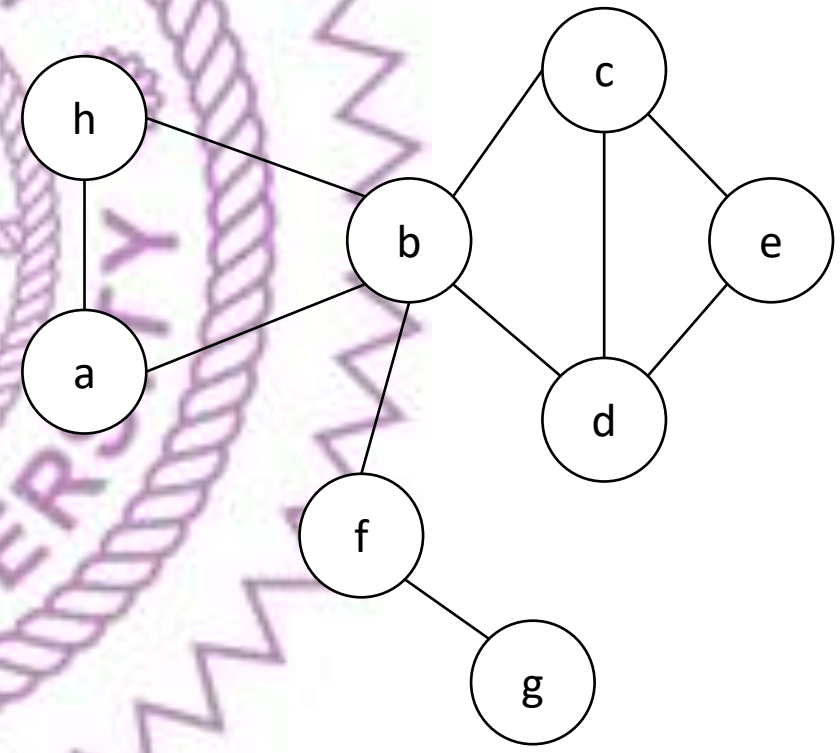
No self cycle

- 假設 graph 中沒有自環對接下來的說明比較方便



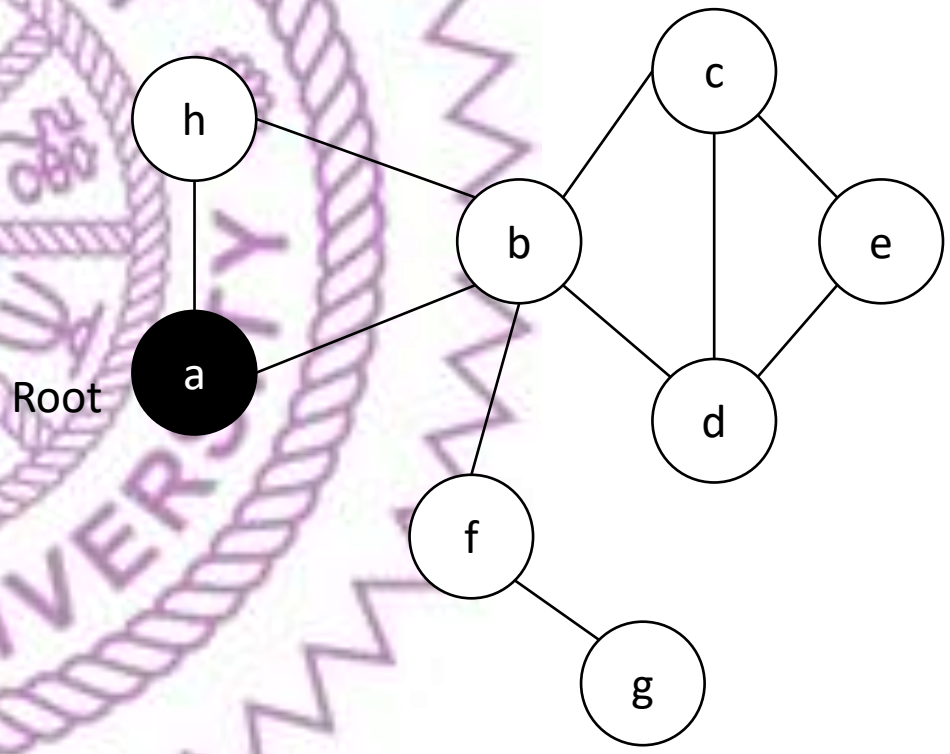
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



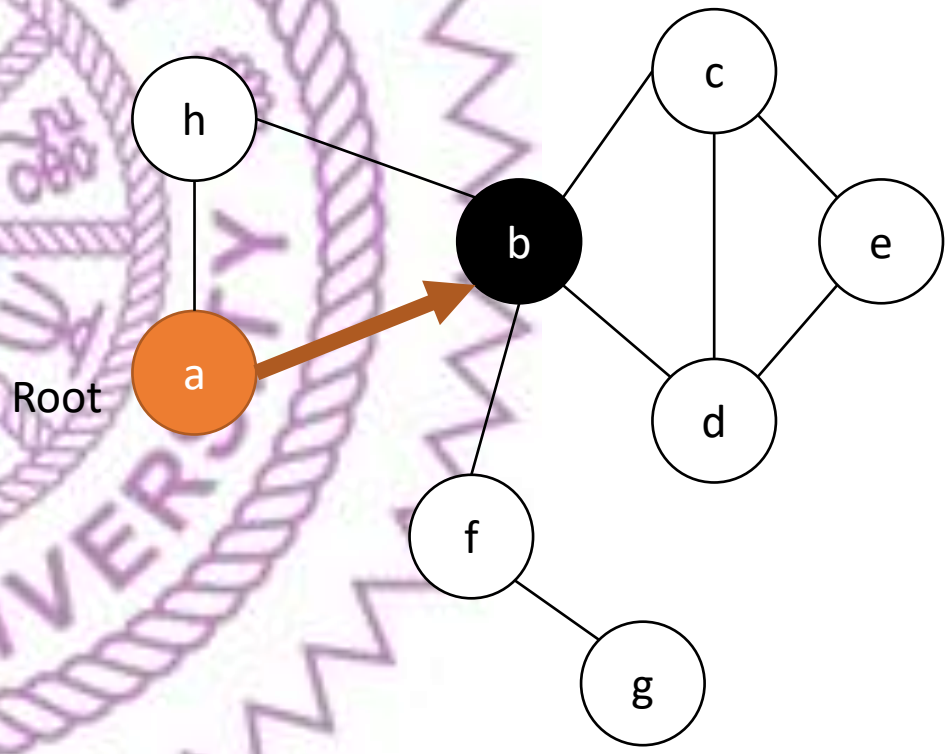
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



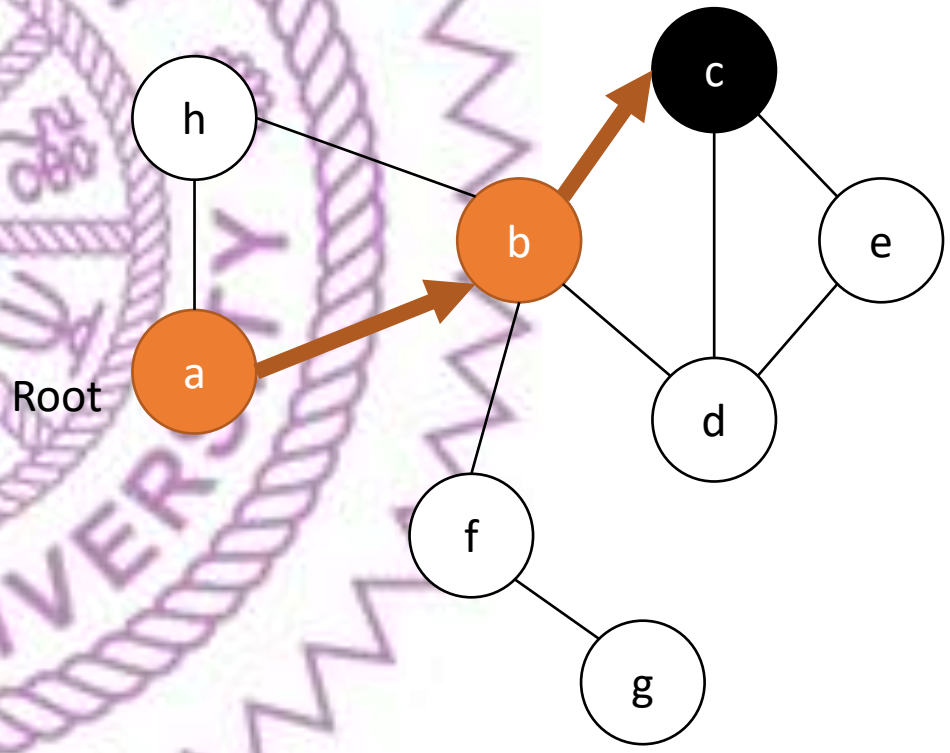
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



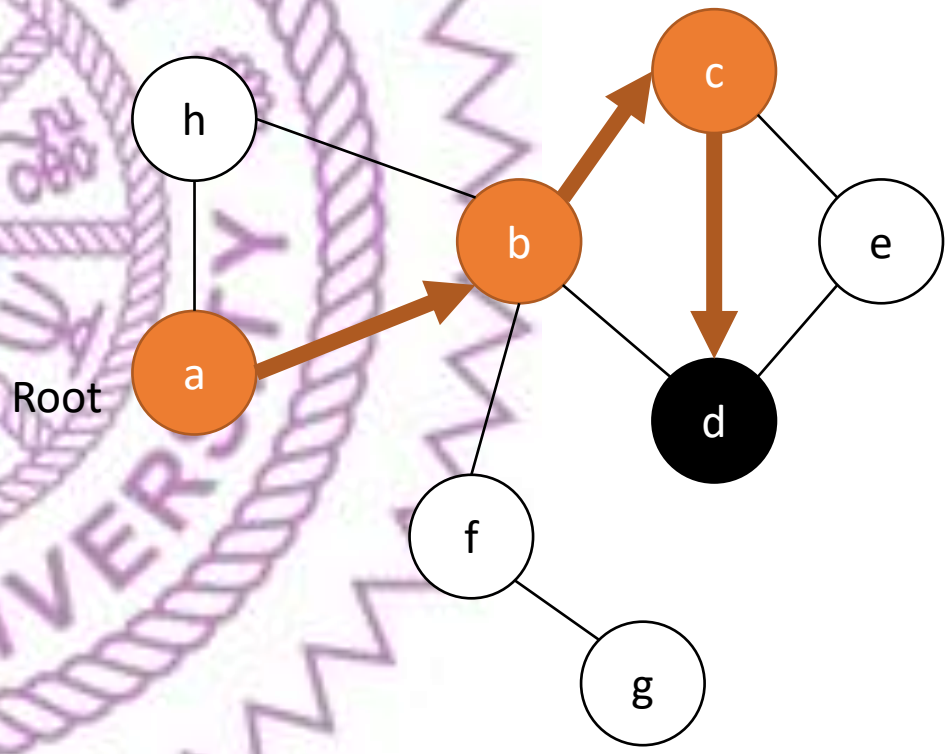
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



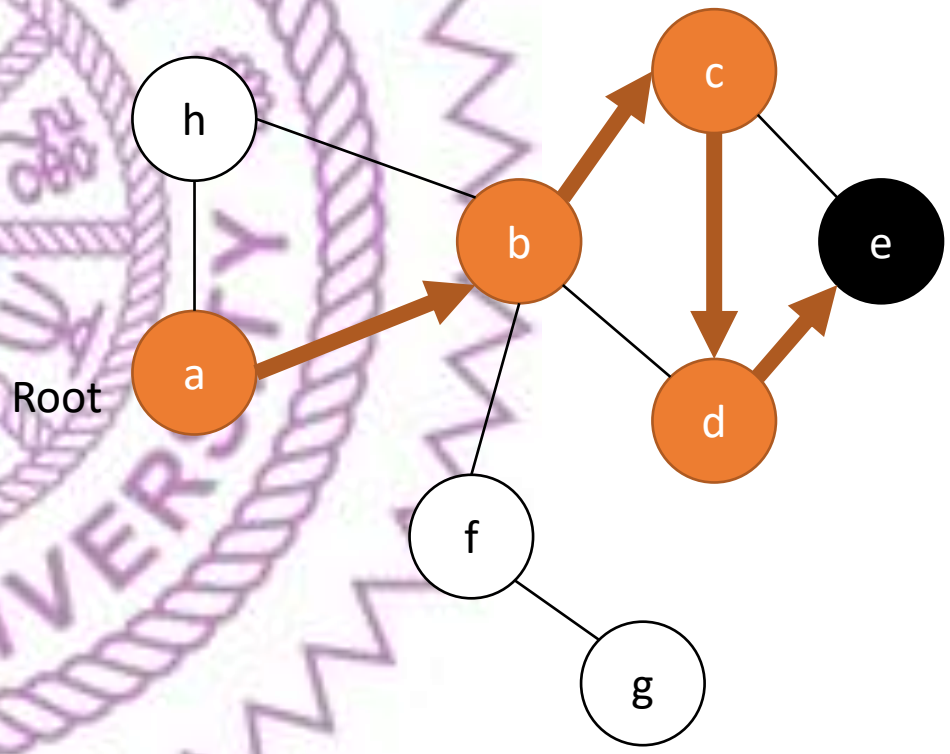
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



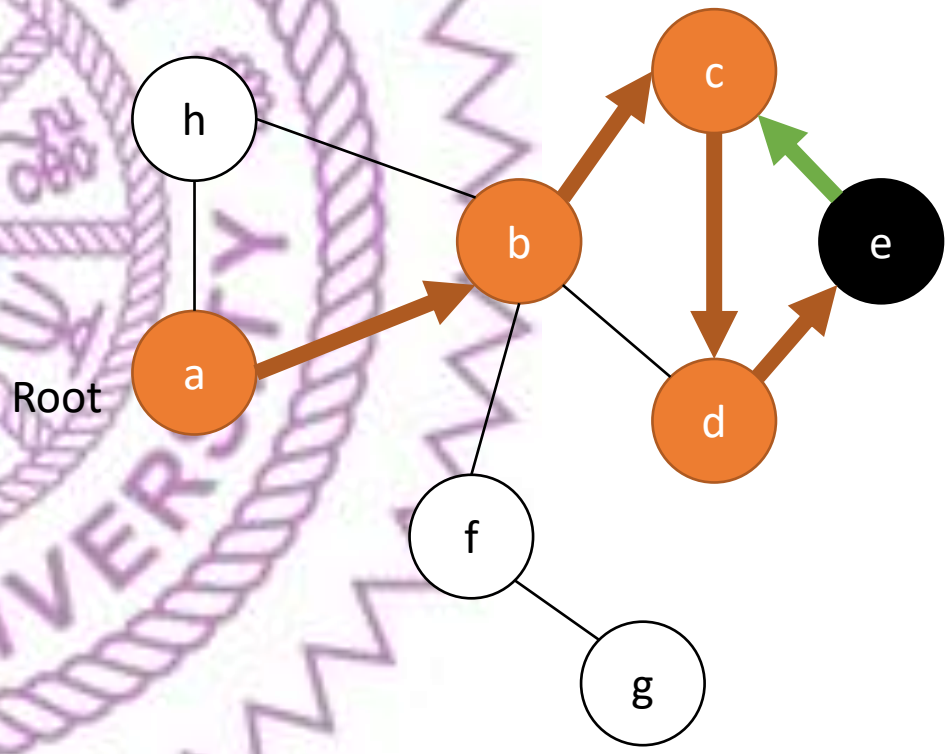
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



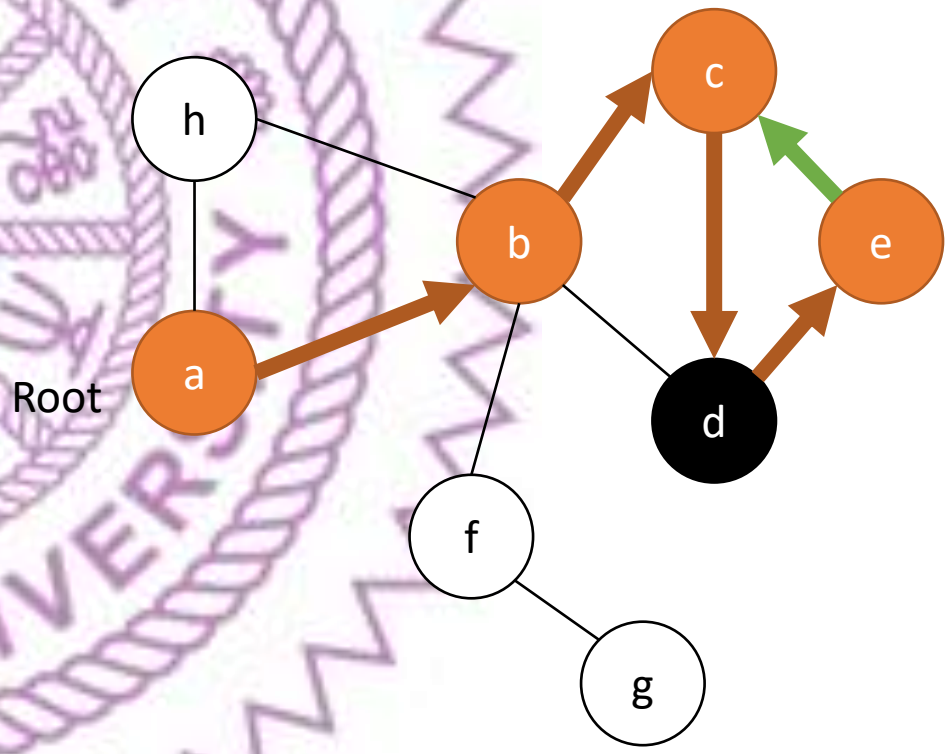
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



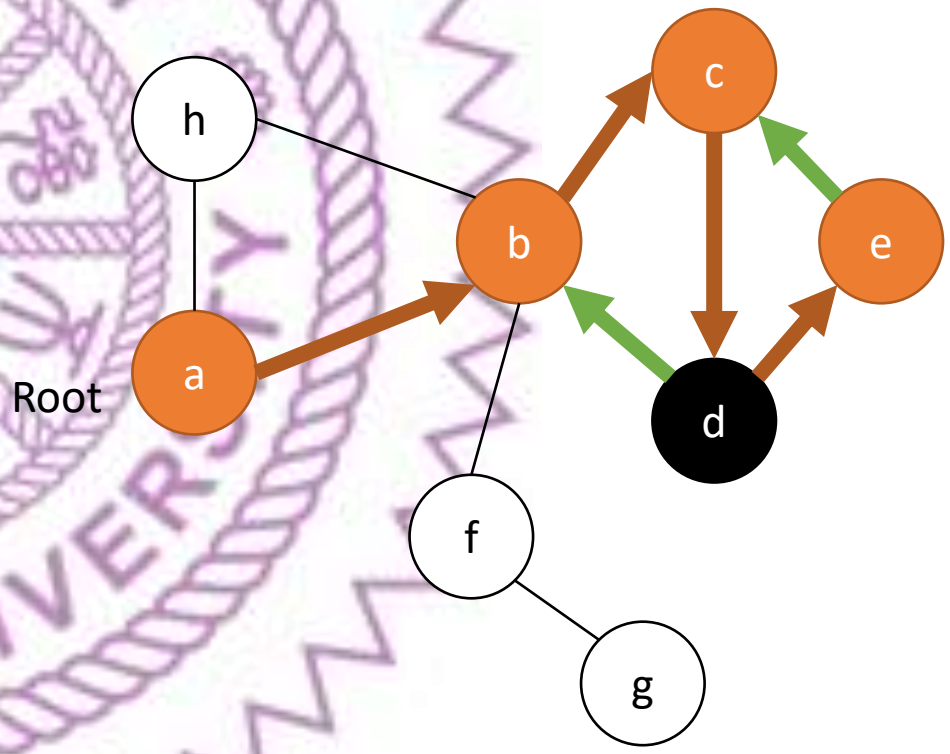
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



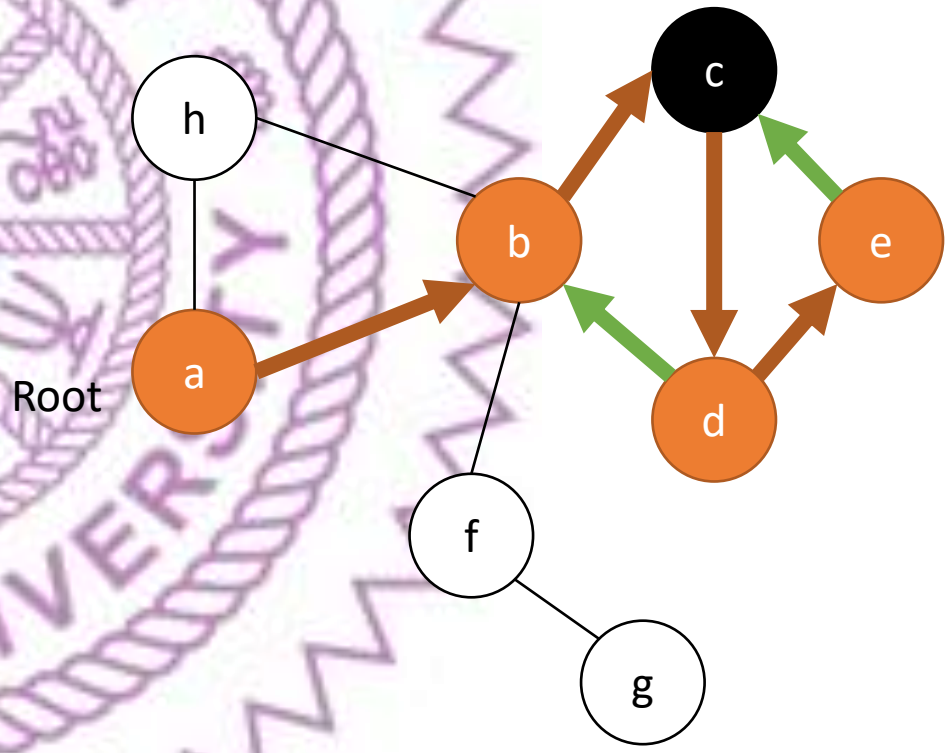
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



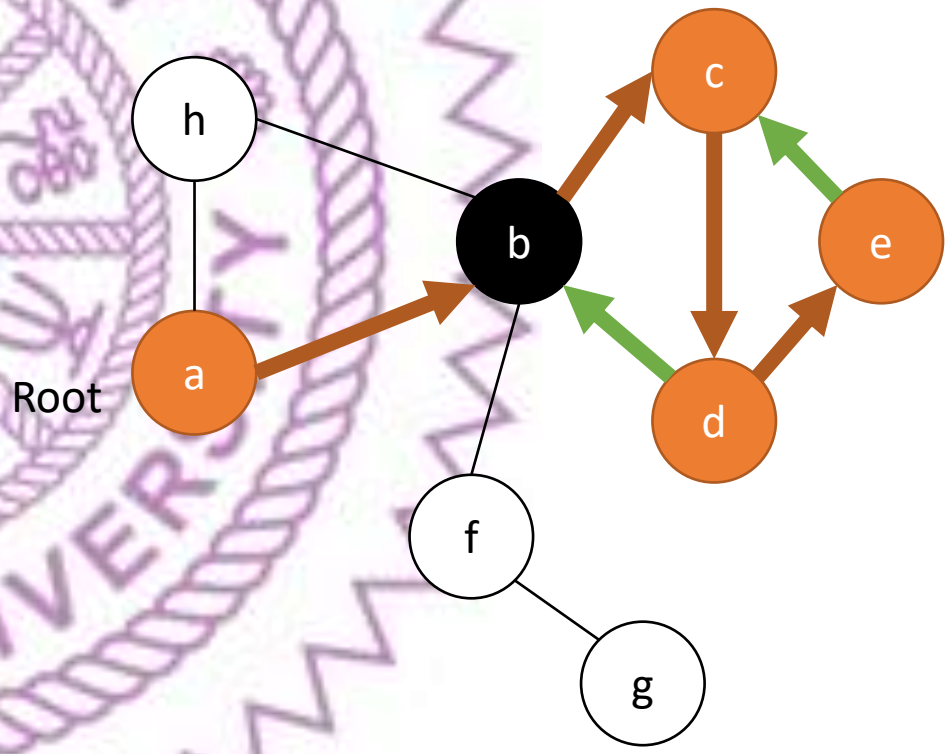
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



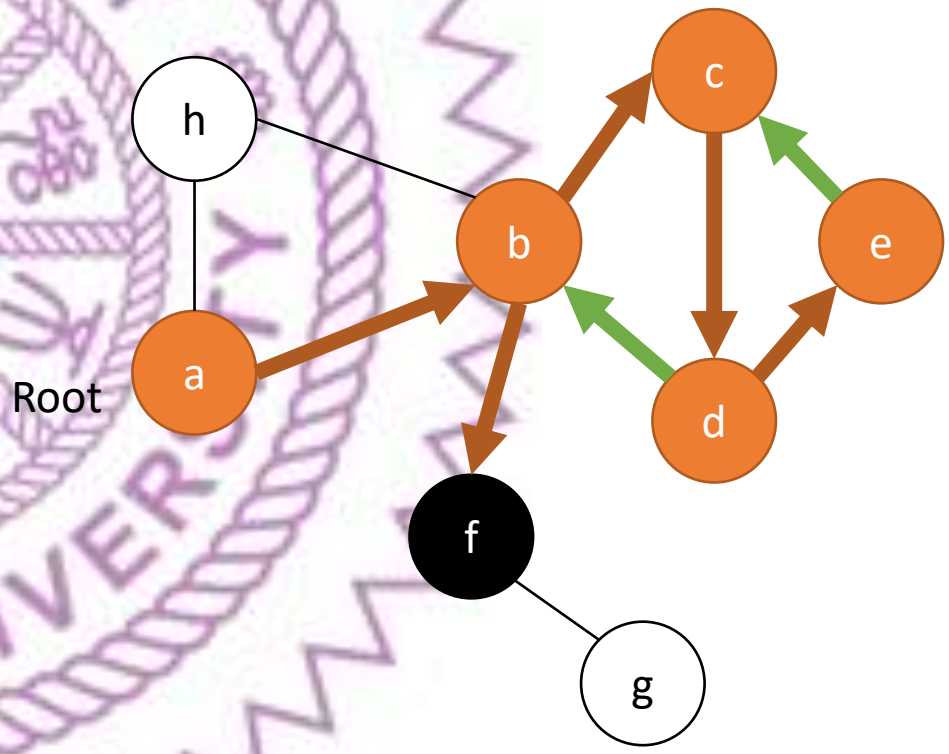
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



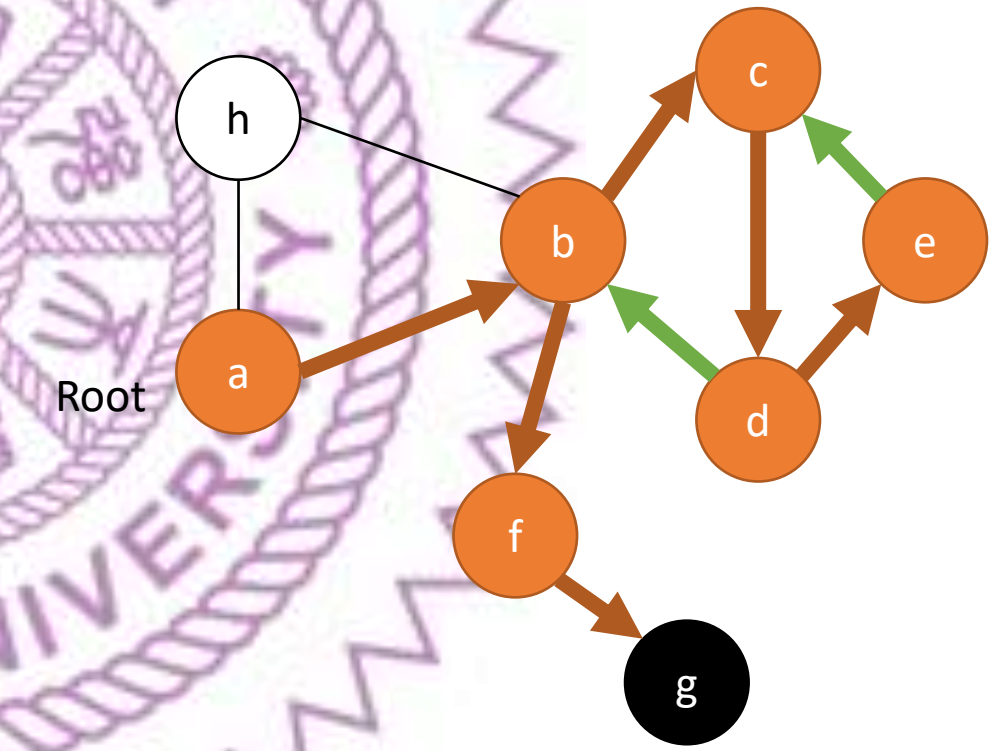
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



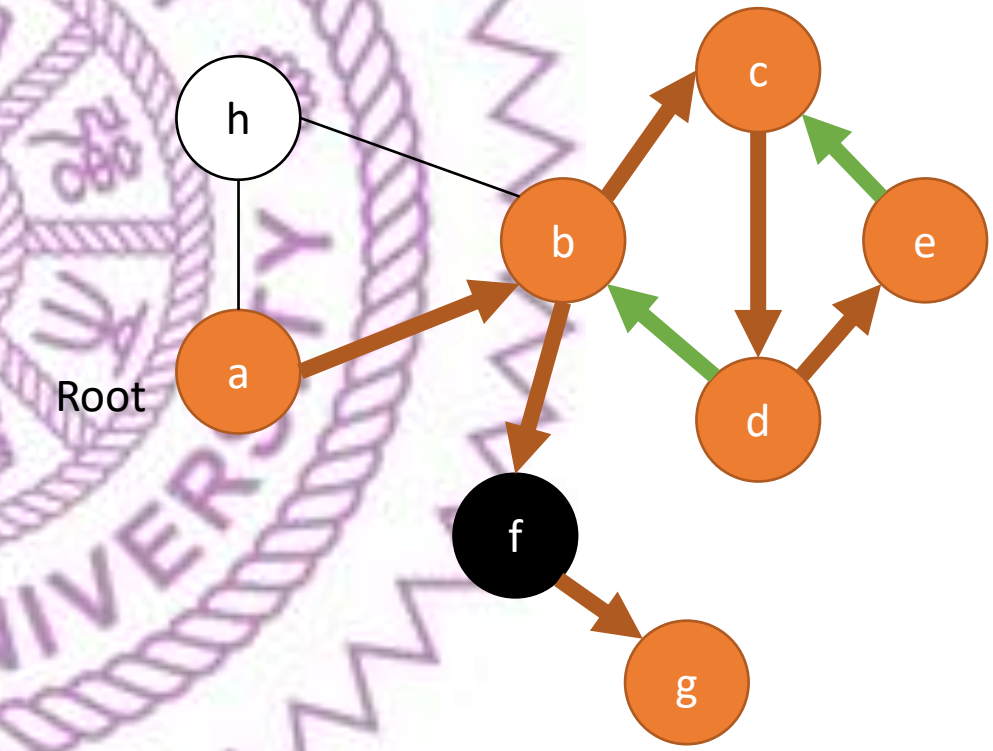
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



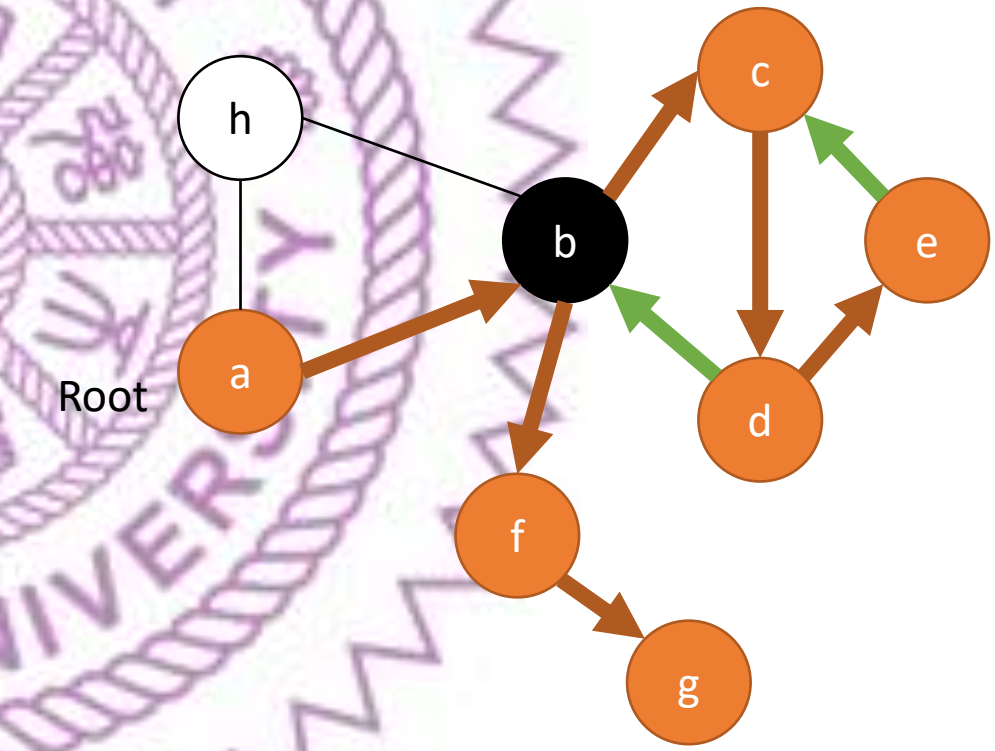
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



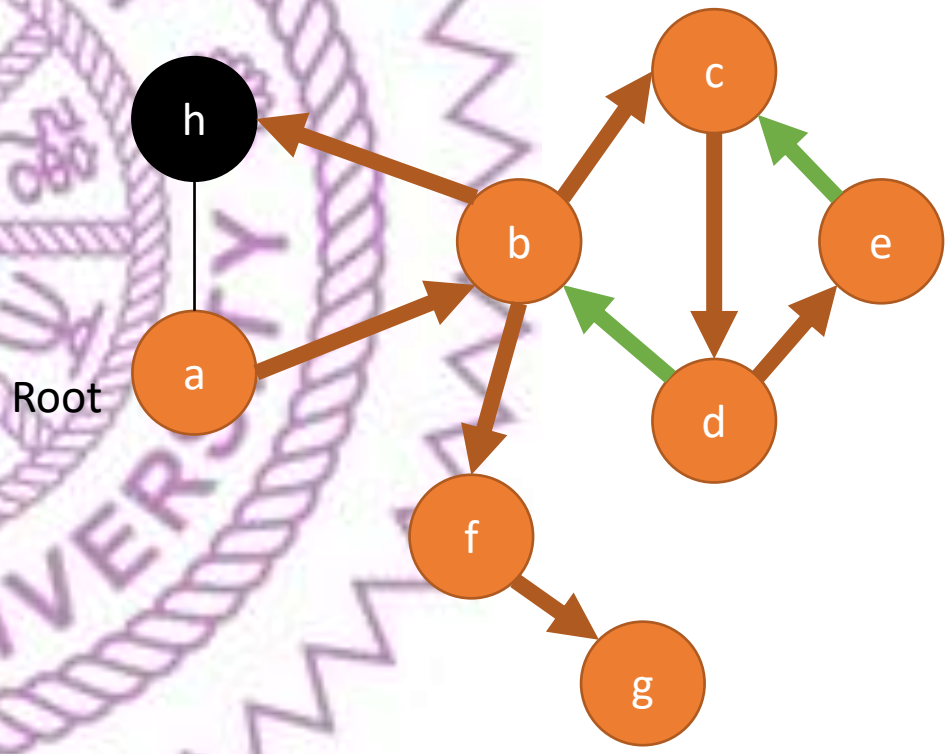
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



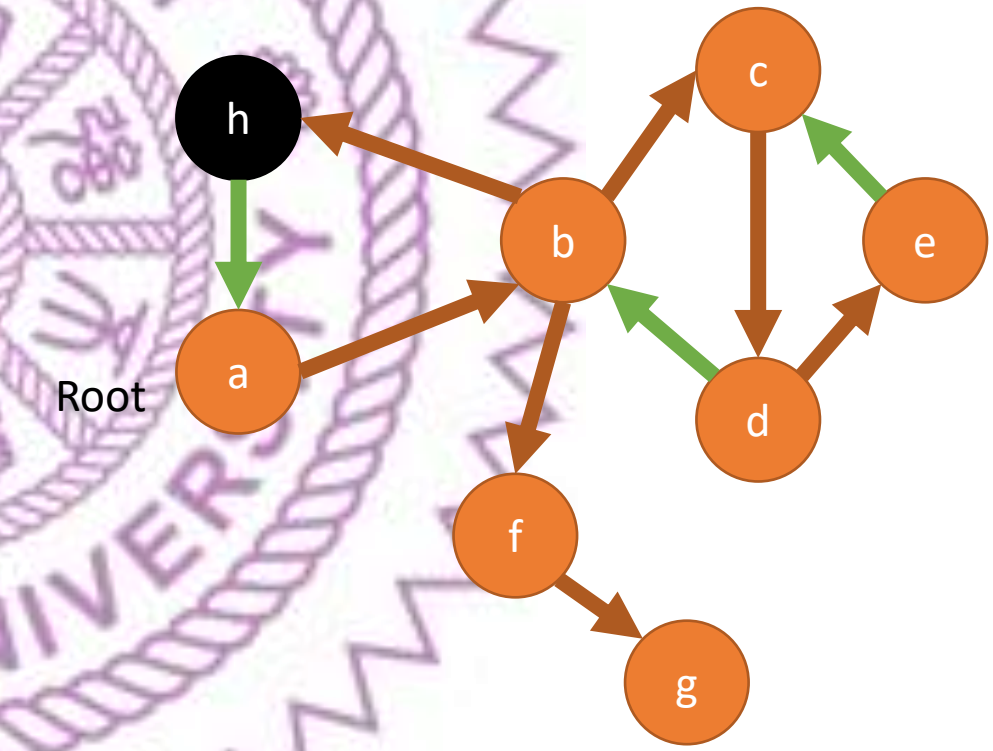
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



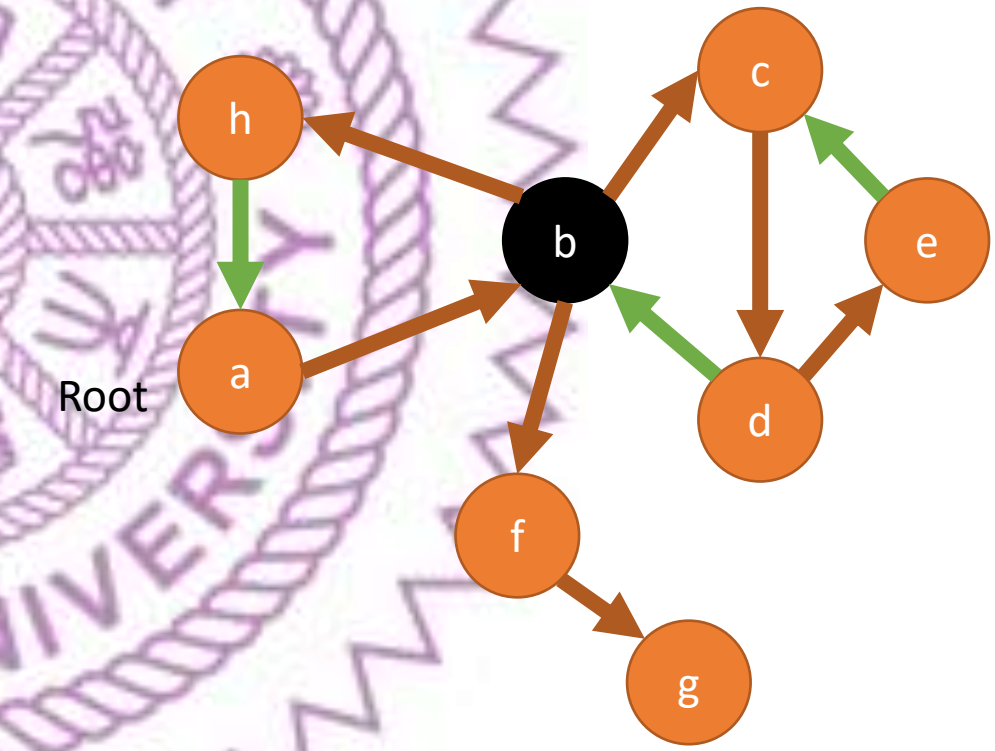
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



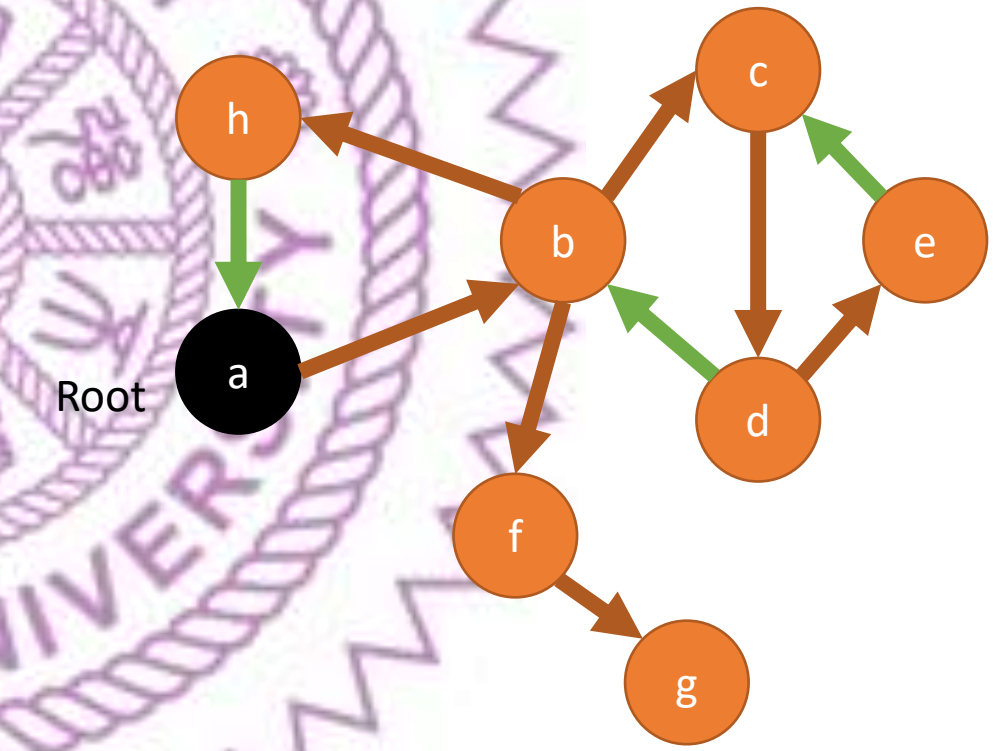
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```



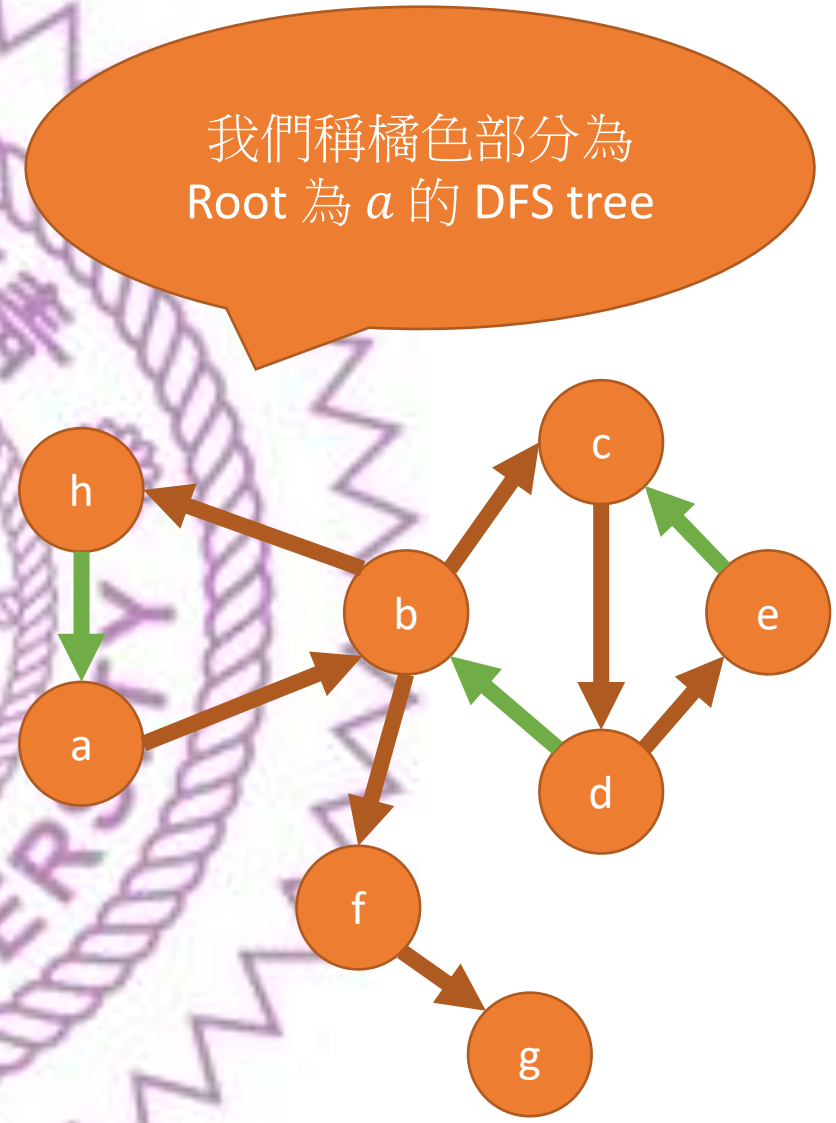
DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```

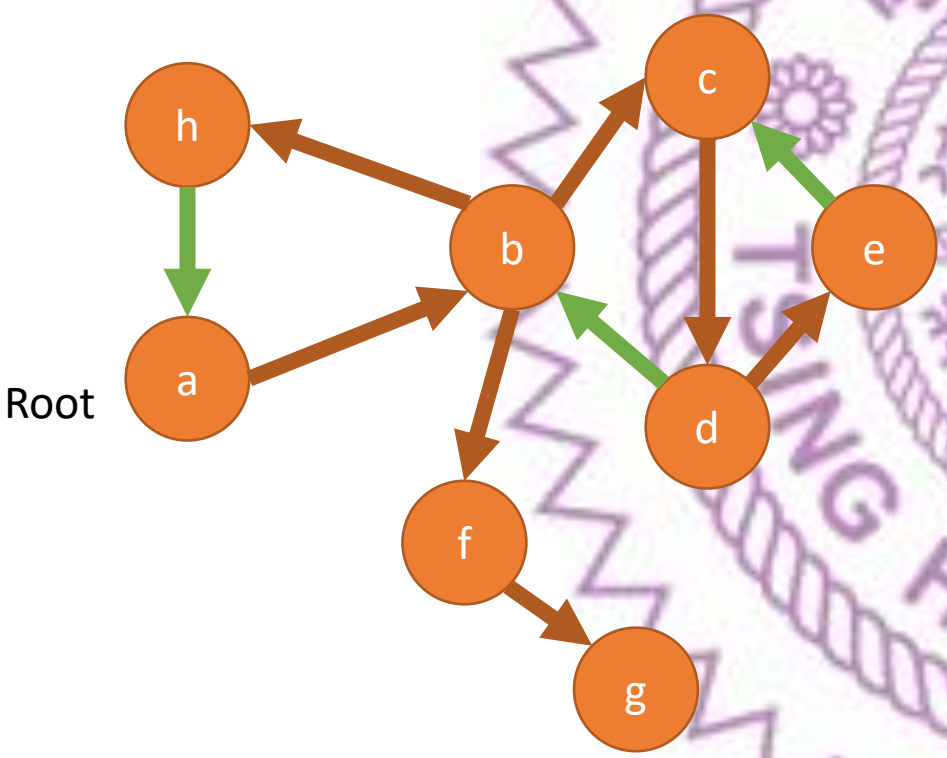


DFS

```
void DFS(Vertex u) {
  visit[u] := true;
  for ( Vertex v : neighbors[u] ) {
    if (not visit[v]){
      DFS(v);
    }
  }
}
```

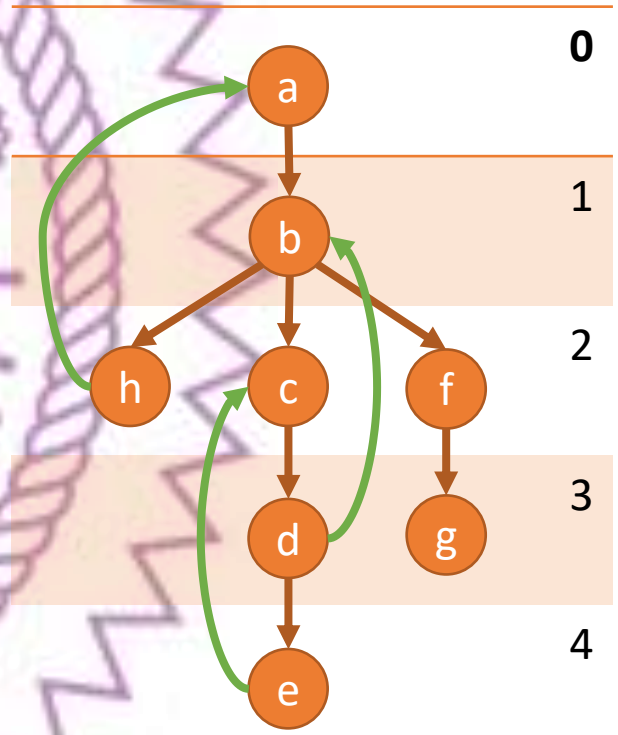


DFS tree 性質



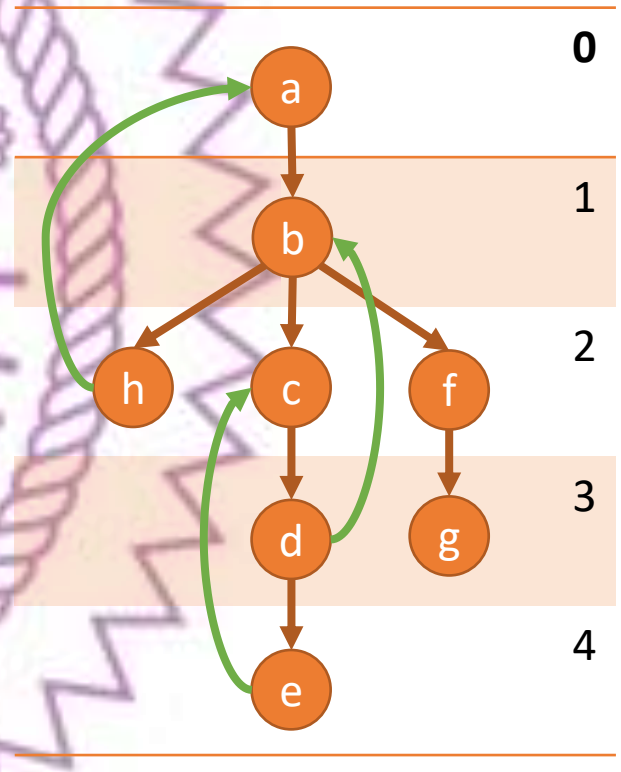
按深度排列

這張有向圖
我們稱為DFS圖



DFS tree 性質

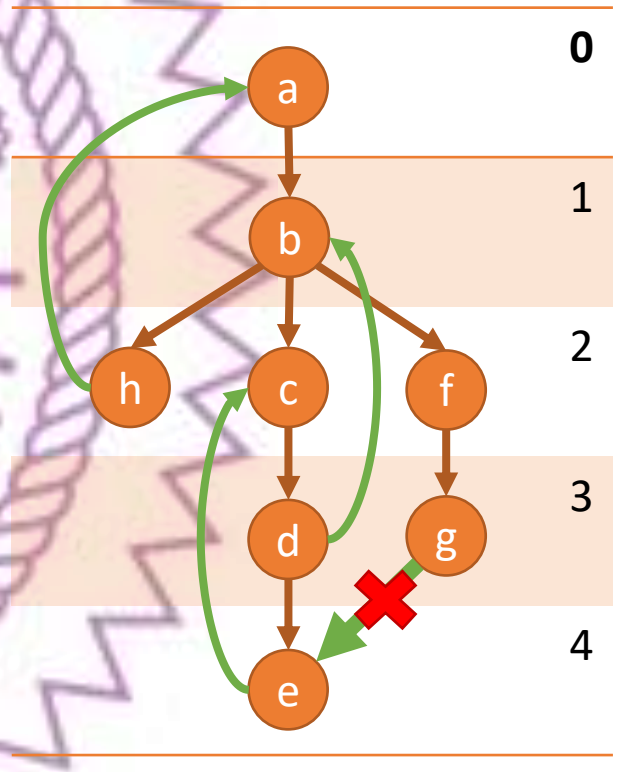
- 我們可以把 graph 中所有 edge 分成
  - 在 DFS tree 上的 (橘色邊)
  - 不在 DFS tree 的 (綠色邊)
- 可以發現所有綠色邊的終點都是起點的祖先
  - a 是 h 的祖先
  - b 是 d 的祖先
  - c 是 e 的祖先



DFS tree 性質

- 那會不會出現像 $g \to e$
  這種從一個子樹連到另一個子樹的 edge 呢？

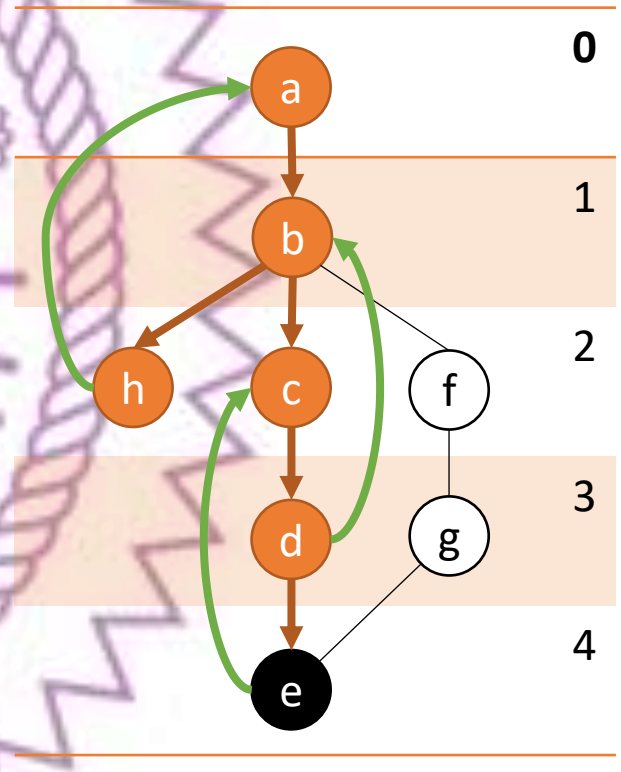
- 絕對不會發生



DFS tree 性質

- DFS 走到一個 vertex 後
  一定會把相連的 edge 都走完才會離開

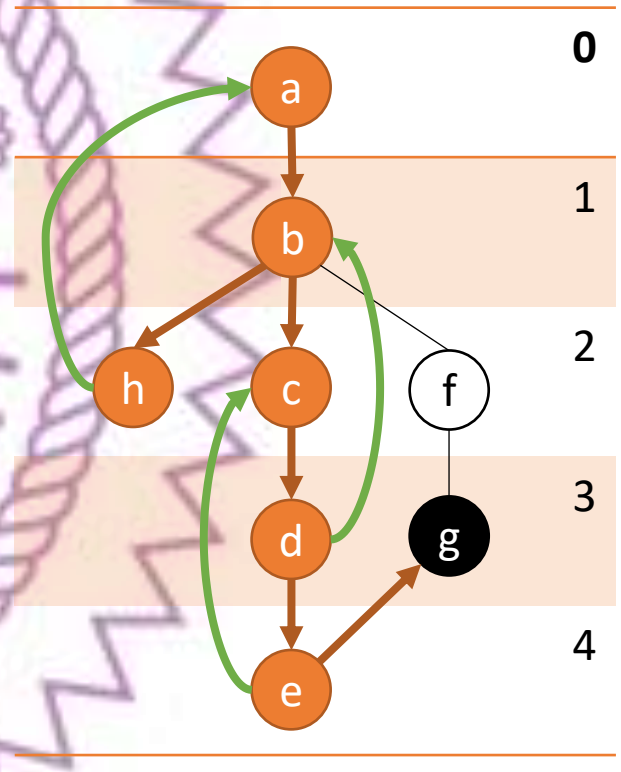
- 例如現在走到 e
  發現有 $e \to g$ 這條 edge 就會直接走到 g



DFS tree 性質

- DFS 走到一個 vertex 後
  一定會把相連的 edge 都走完才會離開

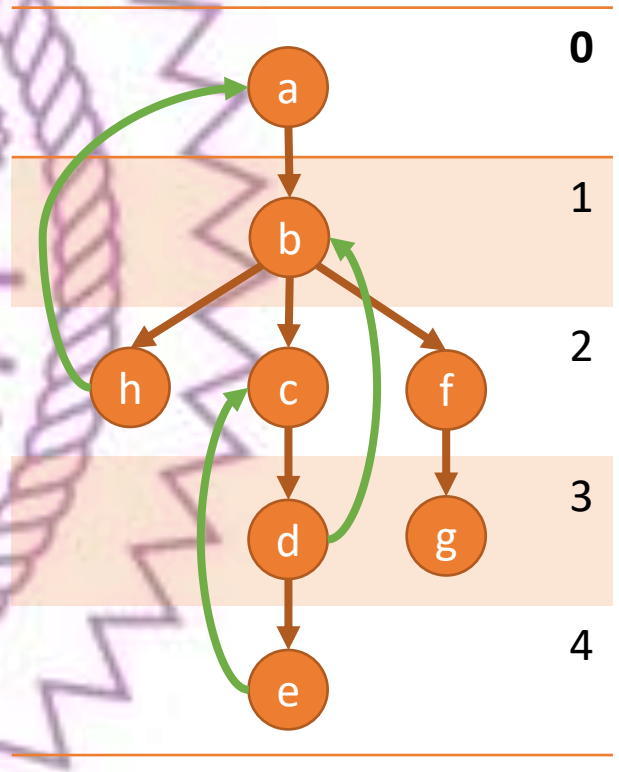
- 例如現在走到 e
  發現有 $e \to g$ 這條 edge 就會直接走到 g



LOW

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2

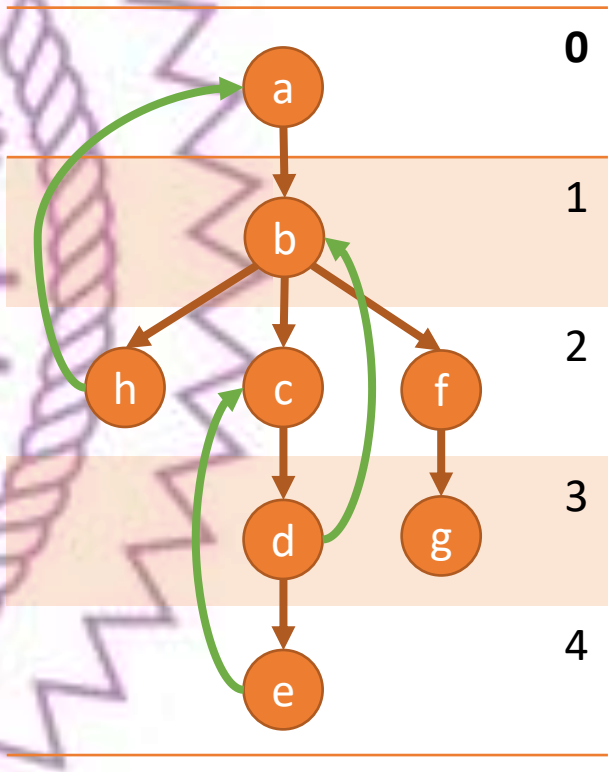
- 用表格 $deep$ 紀錄深度



LOW

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	1	1	2	2	3	0

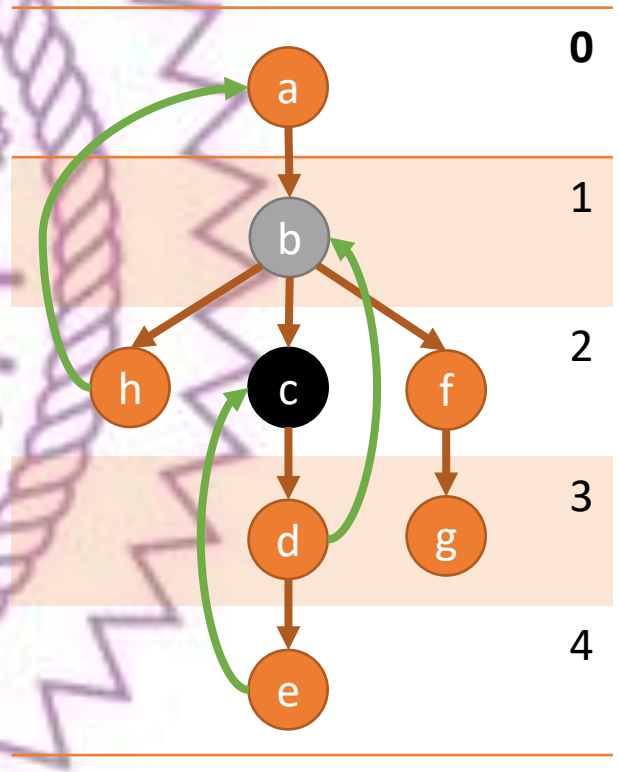
- 用表格 $deep$ 紀錄深度
- $low[x]$ 表示 x 透過其子樹的“綠色邊”能走到的最小深度
- 若 x 的子樹沒有綠色邊則 $low[x] = deep[x]$



LOW

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	1	1	2	2	3	0

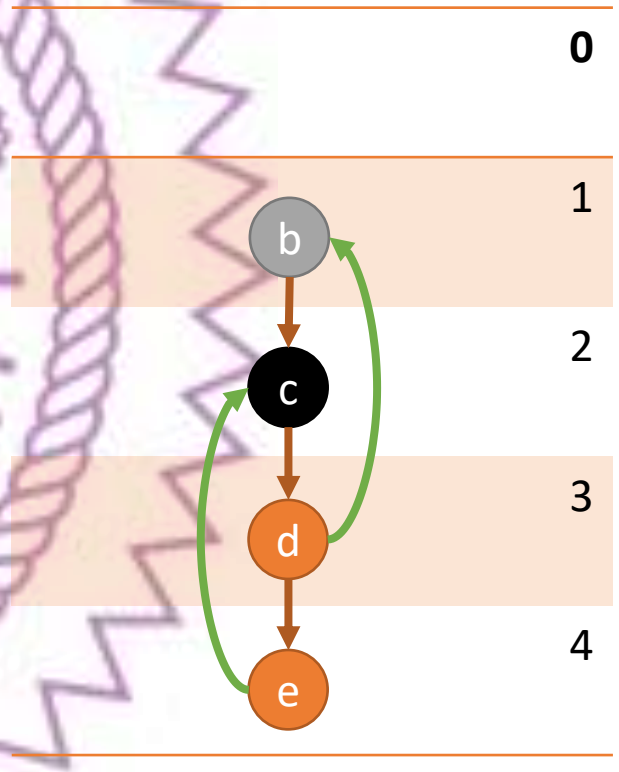
- 以 c 為例
- 他的後代 e 的綠色邊可以走到深度為 1 的 b
- 因此 $low[c] = 1$



LOW

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	1	1	2	2	3	0

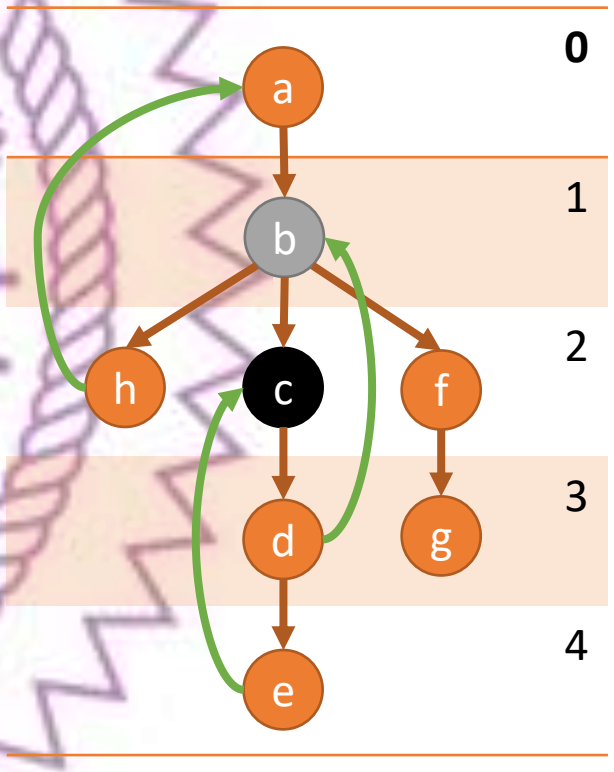
- 以 c 為例
- 他的後代 e 的綠色邊可以走到深度為 1 的 b
- 因此 $low[c] = 1$



LOW

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	1	1	2	2	3	0

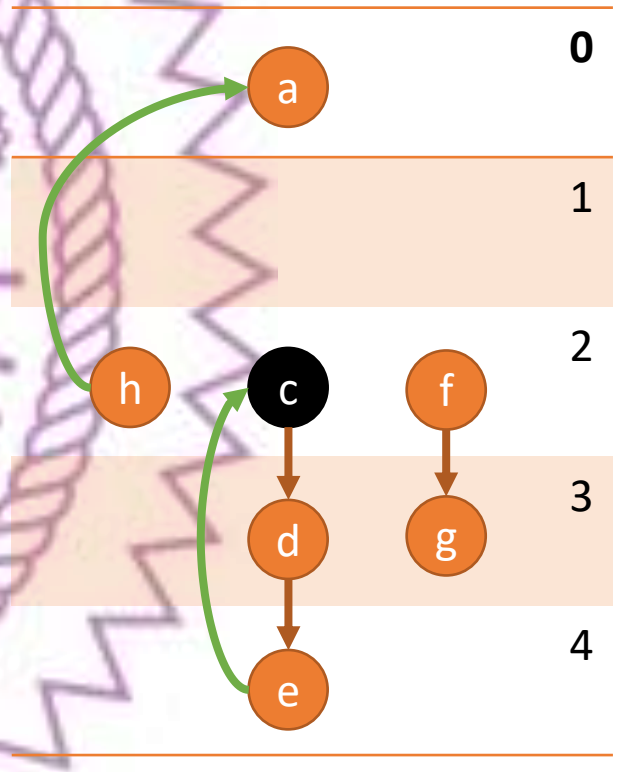
- 設 v 是 u 的小孩
- u 是割點或是 Root $\Leftrightarrow low[v] \geq deep[u]$
- 以 $v = c, u = b$ 為例
  - $low[c] \geq deep[b]$



LOW

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	1	1	2	2	3	0

- 設 v 是 u 的小孩
- u 是割點或是 Root $\Leftrightarrow low[v] \geq deep[u]$
- 以 $v = c, u = b$ 為例
  - $low[c] \geq deep[b]$
  - 把 b 刪掉後 c 的子樹就會被切割出來



DFS 就能計算

```
void DFS(Vertex u, Vertex pa, Number d){
  visit[u] := true;
  deep[u] := d;
  low[u] := d;
  for ( Vertex v : neighbors[u] ){
    if ( not visit[v] ){
      DFS(v, u, d + 1);
      low[u] := min(low[u], low[v]);
    } else if ( deep[v]<deep[u] and v ≠ pa ){
      low[u] := min(low[u], deep[v]);
    }
  }
}
```

DFS 就能計算

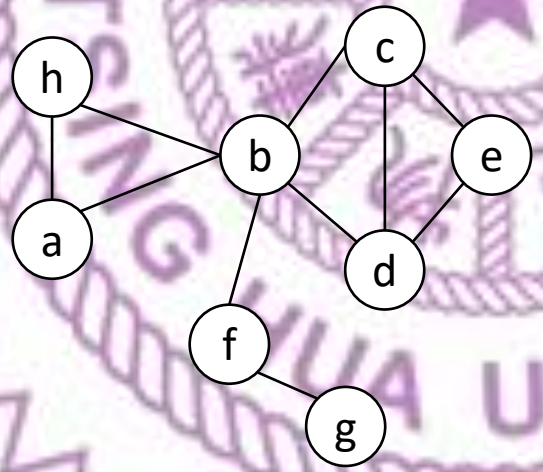
- 只要多紀錄一個 stack
- 就能順便找出所有 block

```
void DFS(Vertex u, Vertex pa, Number d) {
  visit[u] := true;
  deep[u] := d;
  low[u] := d;
  stk.push(u);
  for ( Vertex v : neighbors[u] ){
    if ( not visit[v] ){
      DFS(v, u, d + 1);
      low[u] := min(low[u], low[v]);
      if ( d ≤ low[v] ){
        B := ∅;
        while ( stk.top() ≠ u ){
          B := B ∪ stk.top();
          stk.pop();
        }
        B = B ∪ u;
        Blocks := Blocks ∪ B;
      }
    } else if ( deep[v]<deep[u] and v ≠ pa ) {
      low[u] := min(low[u], deep[v]);
    }
  }
}
```

DFS

	a	b	c	d	e	f	g	h
$deep$	x	x	x	x	x	x	x	x
low	x	x	x	x	x	x	x	x

stk



0

1

2

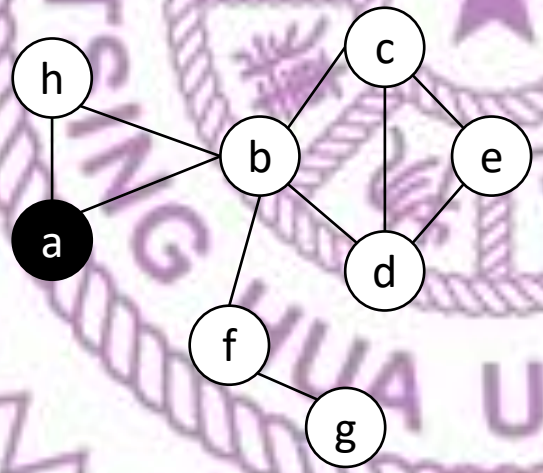
3

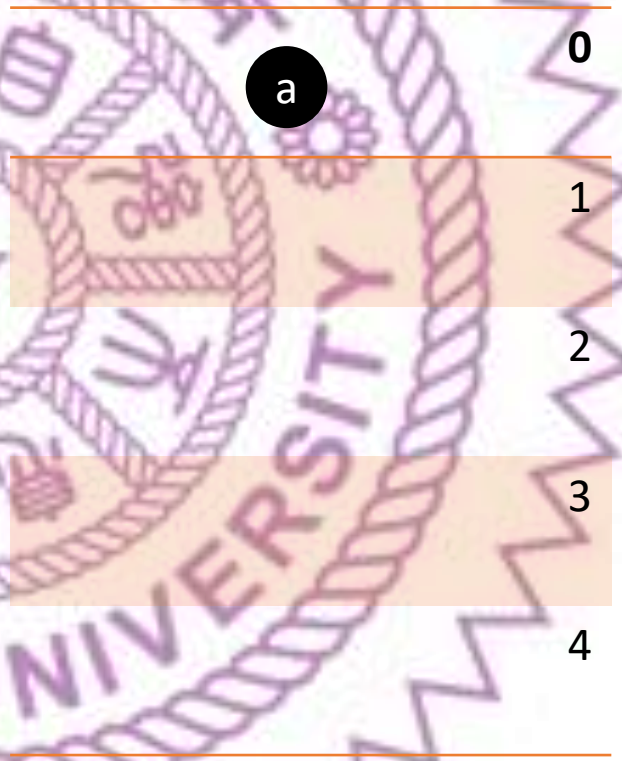
4

DFS

	a	b	c	d	e	f	g	h
$deep$	0	x	x	x	x	x	x	x
low	0	x	x	x	x	x	x	x

stk | a |

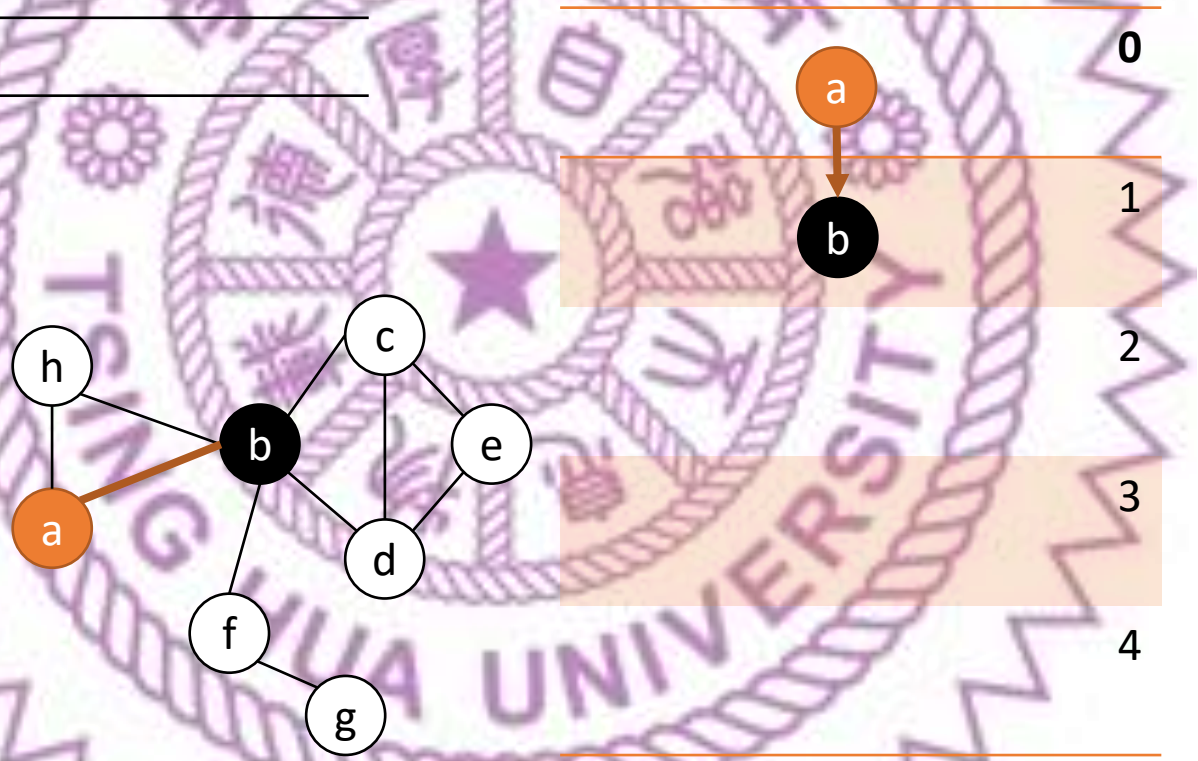




DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	x	x	x	x	x	x
low	0	1	x	x	x	x	x	x

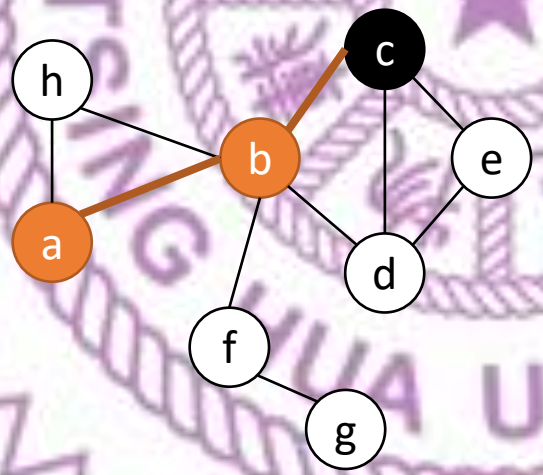
stk | a | b |

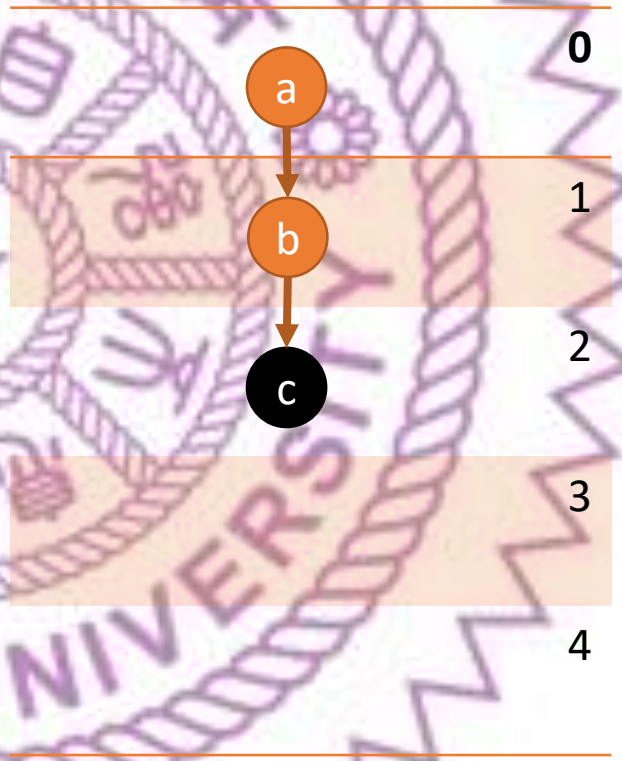


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	x	x	x	x	x
low	0	1	2	x	x	x	x	x

stk | a | b | c |

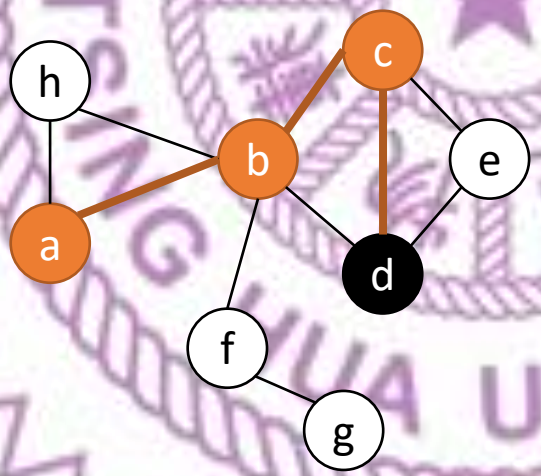


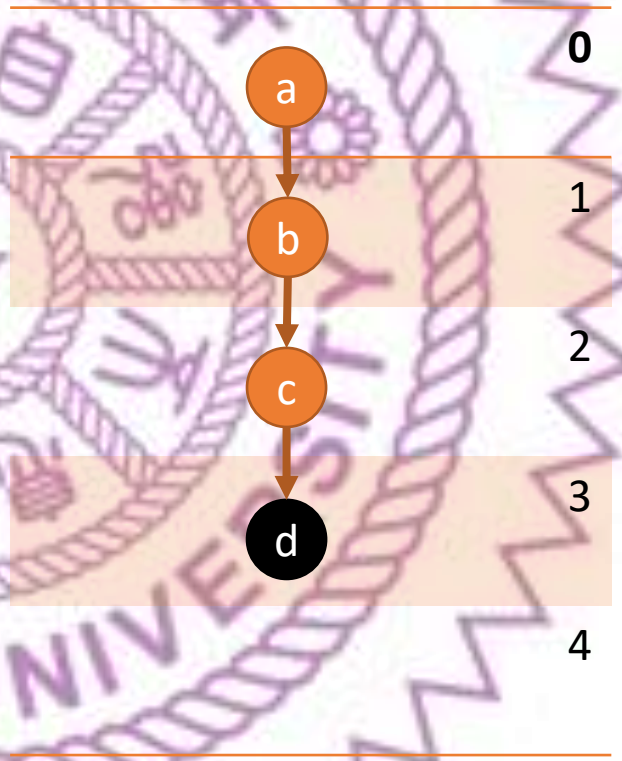


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	x	x	x	x
low	0	1	2	3	x	x	x	x

stk | a | b | c | d |

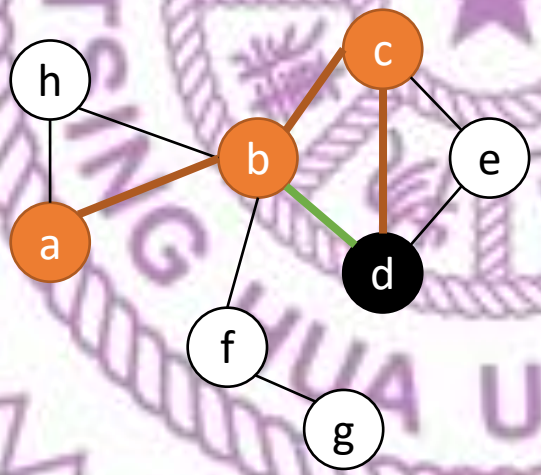


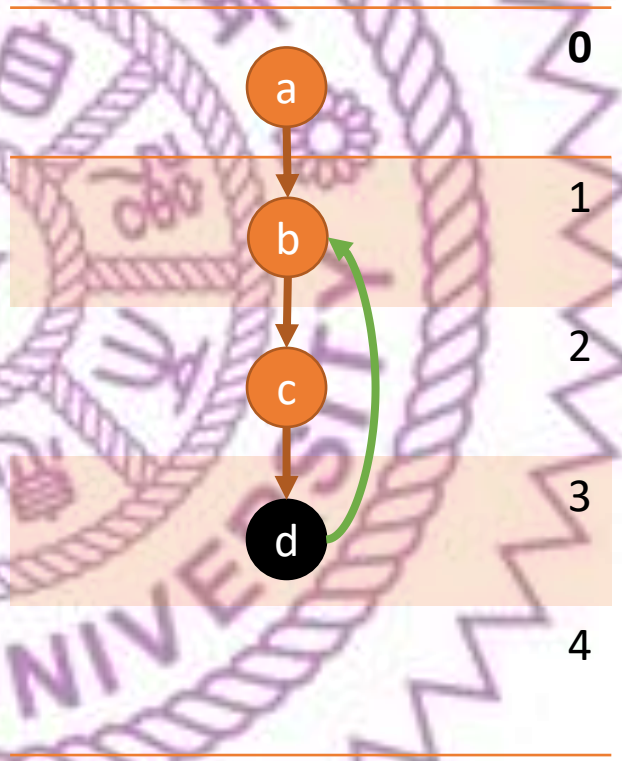


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	x	x	x	x
low	0	1	2	1	x	x	x	x

stk | a | b | c | d |

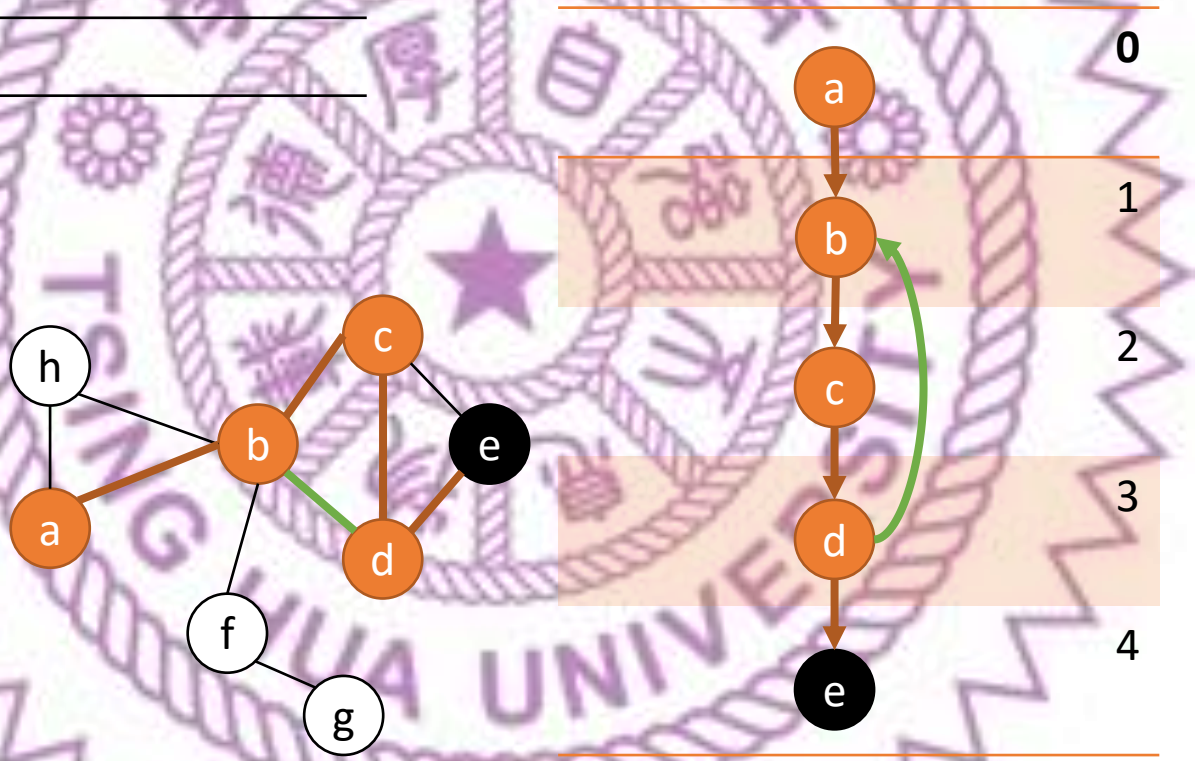




DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	x	x	x
low	0	1	2	1	4	x	x	x

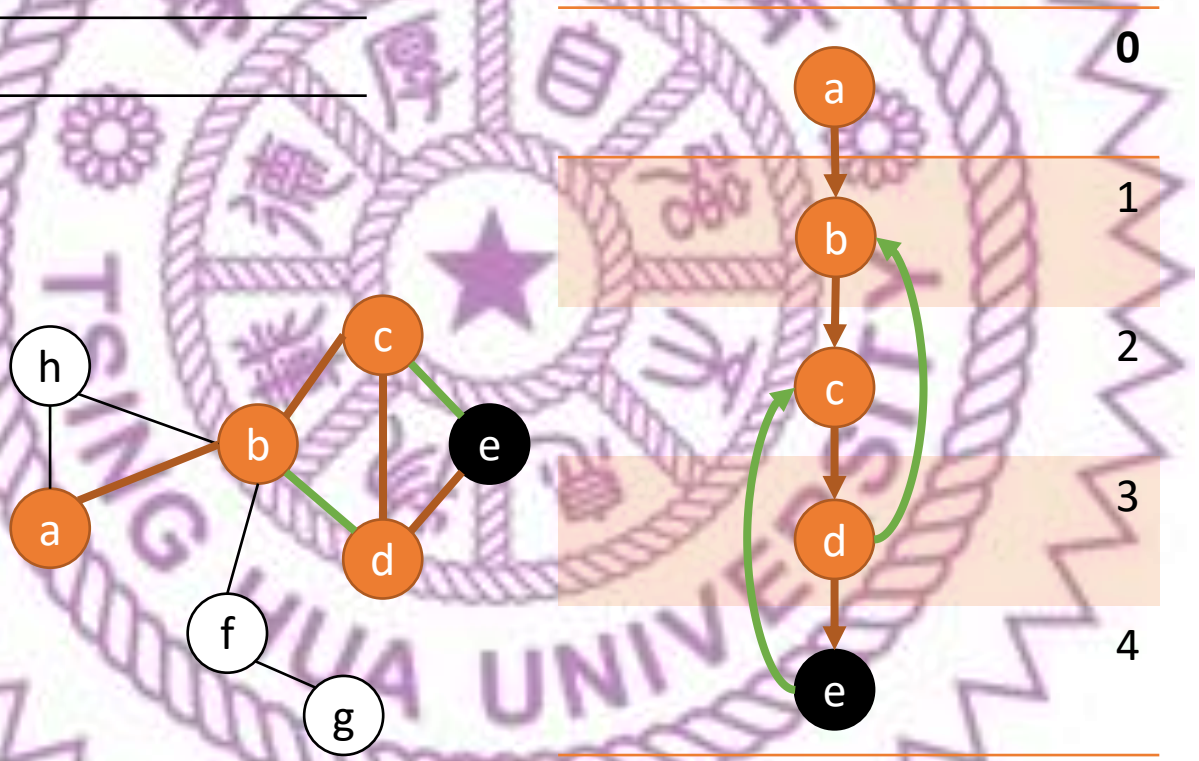
stk | a | b | c | d | e |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	x	x	x
low	0	1	2	1	2	x	x	x

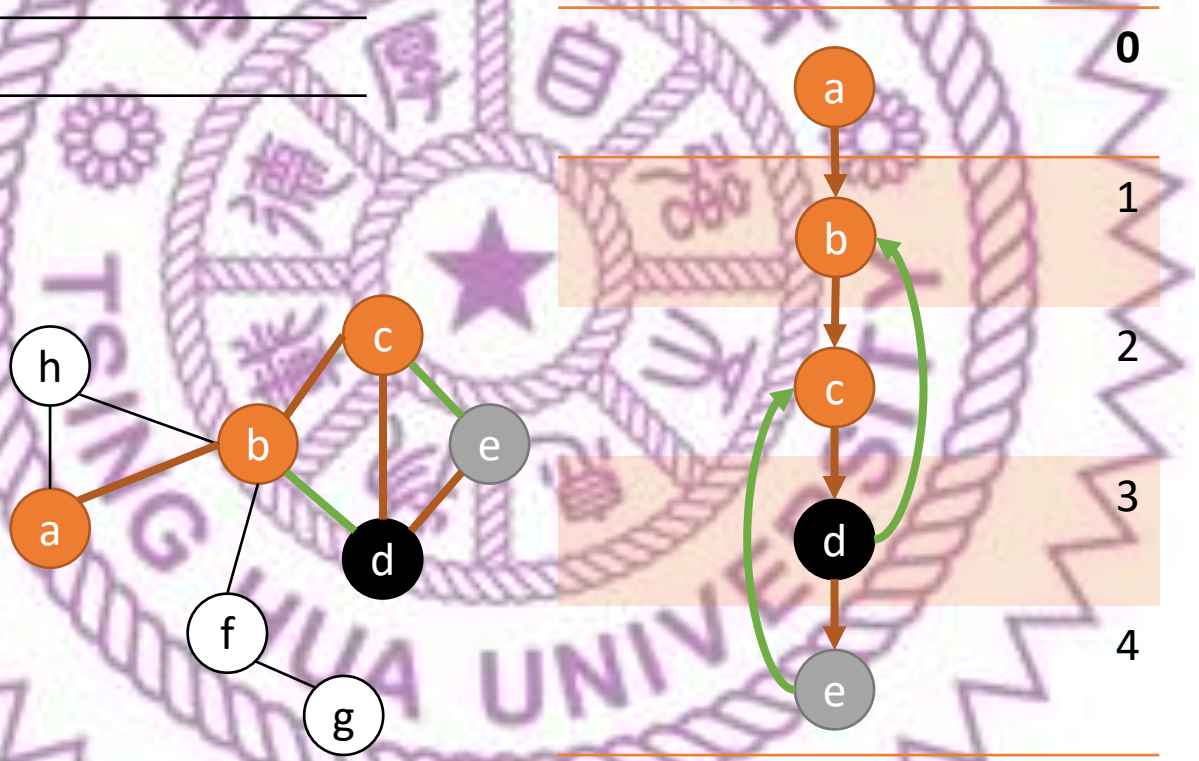
stk | a | b | c | d | e |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	x	x	x
low	0	1	2	1	2	x	x	x

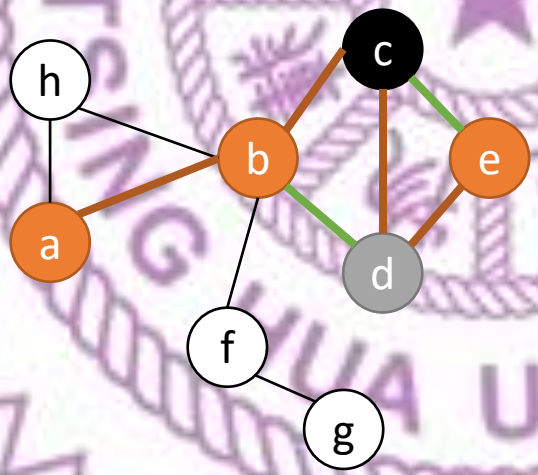
stk | a | b | c | d | e |

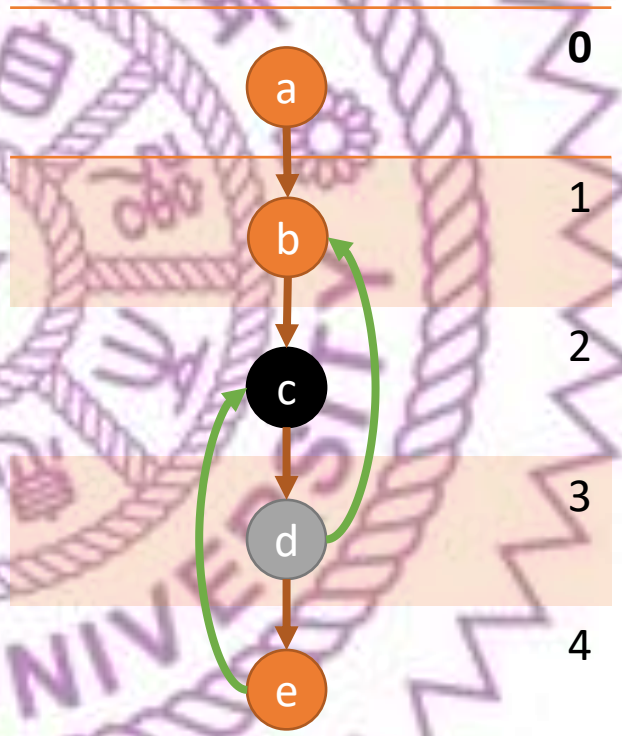


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	x	x	x
low	0	1	1	1	2	x	x	x

stk | a | b | c | d | e |

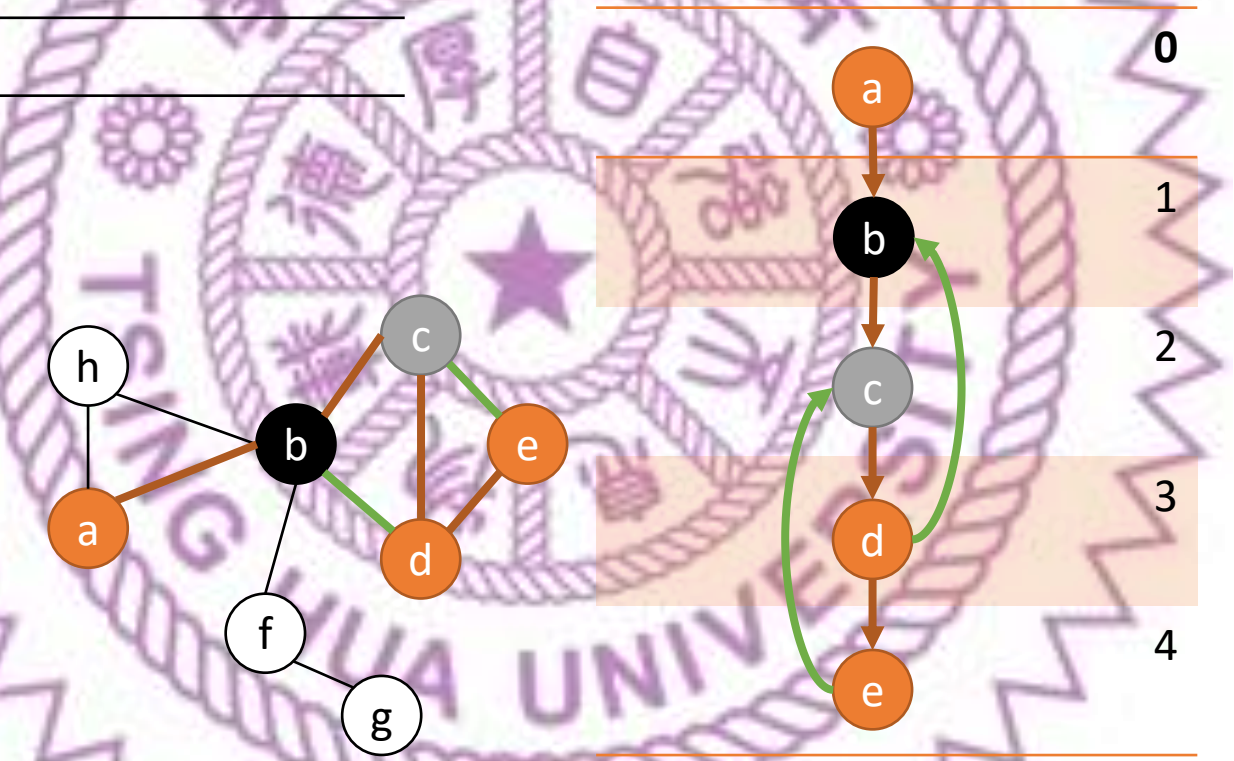




DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	x	x	x
low	0	1	1	1	2	x	x	x

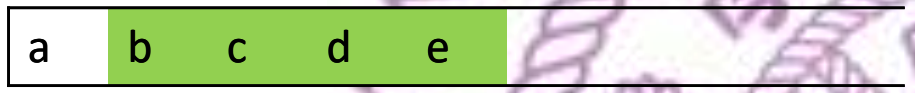
stk | a | b | c | d | e |



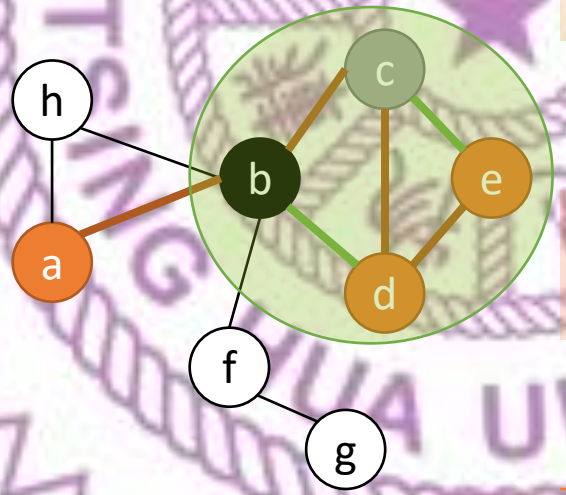
DFS

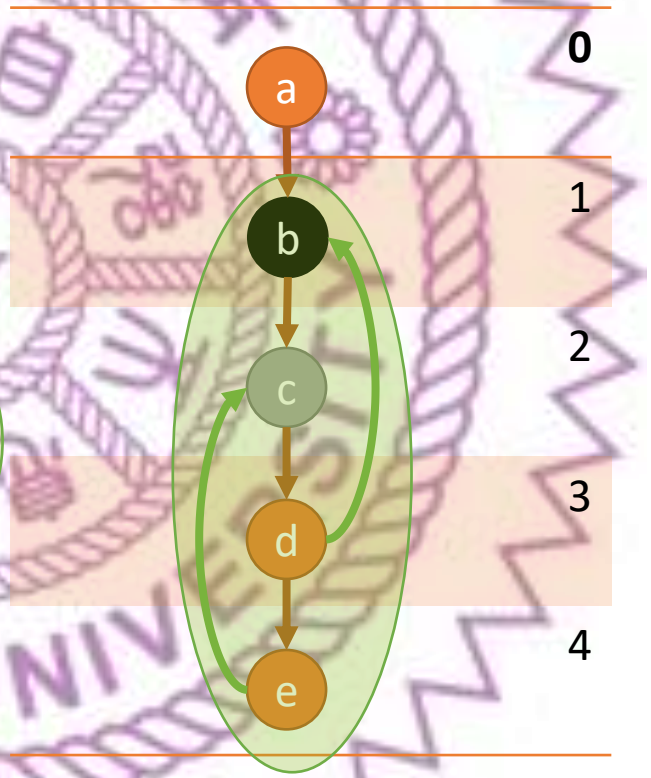
	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	x	x	x
low	0	1	1	1	2	x	x	x

stk



low[c] ≥ deep[b]
形成block



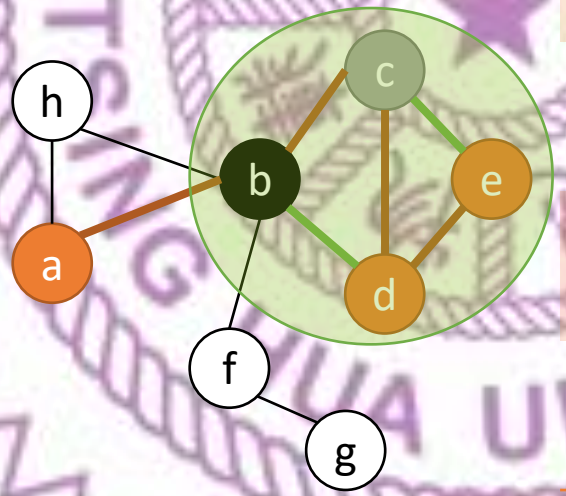


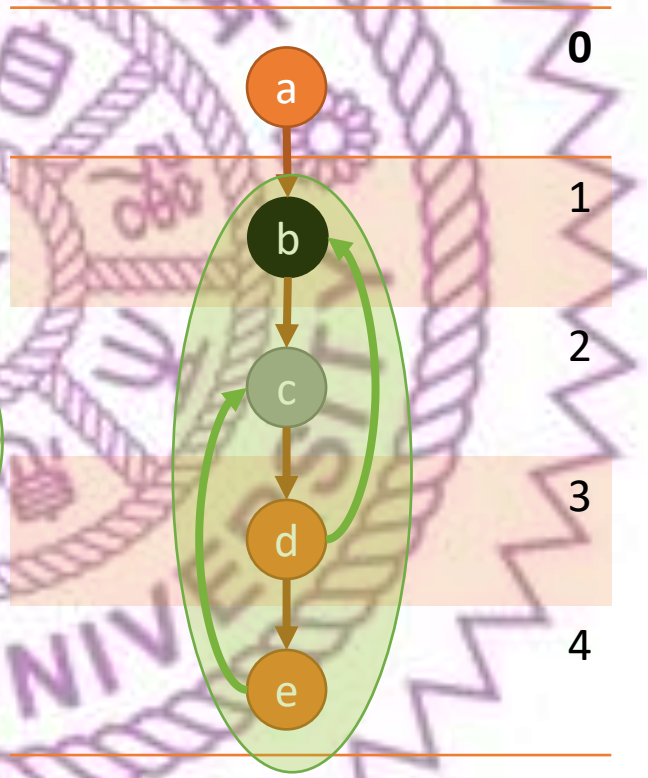
DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	x	x	x
low	0	1	1	1	2	x	x	x

stk | a b |

b有可能被加入其他block中所以留著

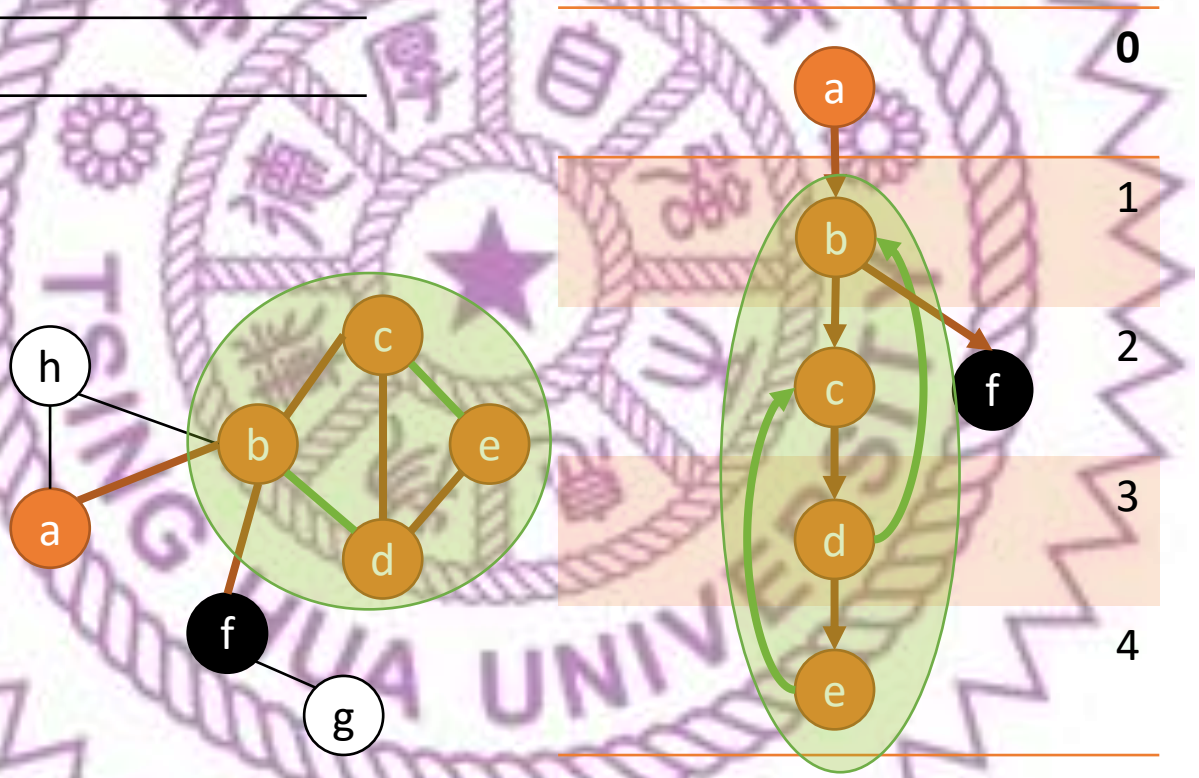




DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	x	x
low	0	1	1	1	2	2	x	x

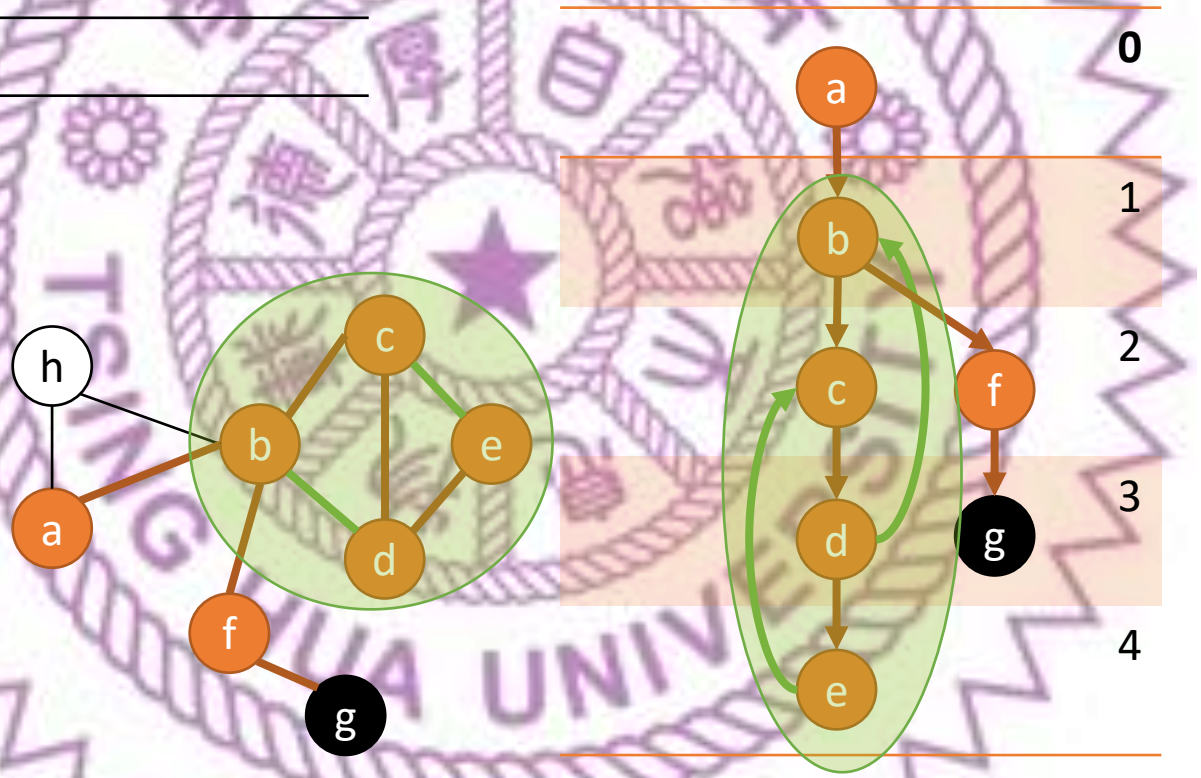
stk | a | b | f |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	x
low	0	1	1	1	2	2	3	x

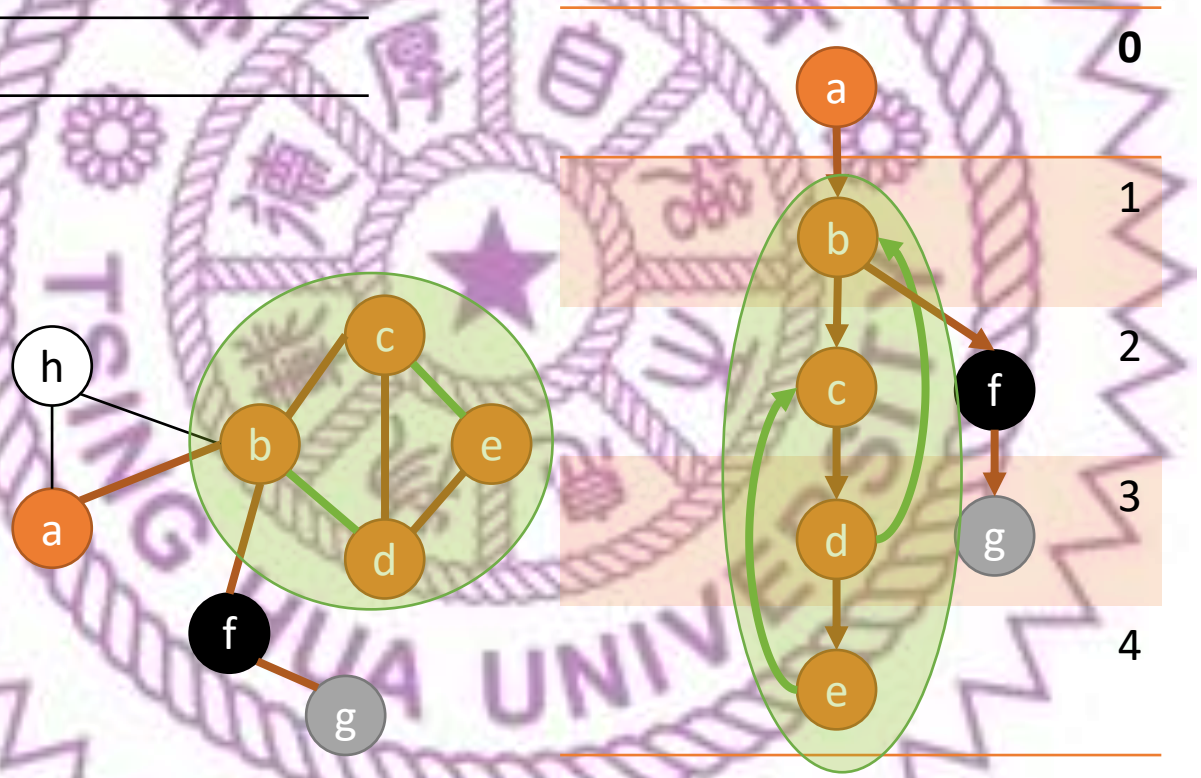
stk | a | b | f | g |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	x
low	0	1	1	1	2	2	3	x

stk | a | b | f | g |

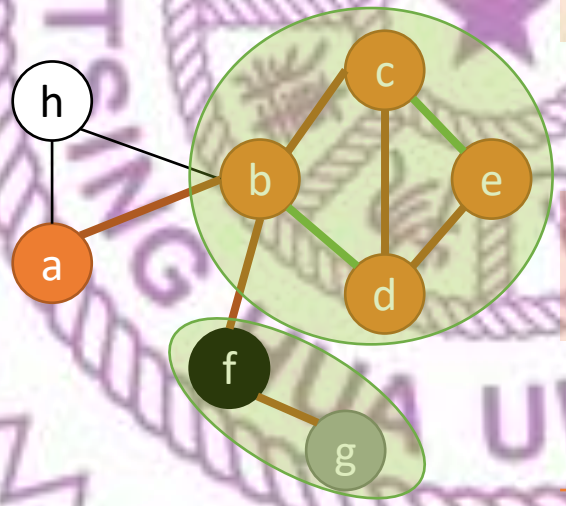


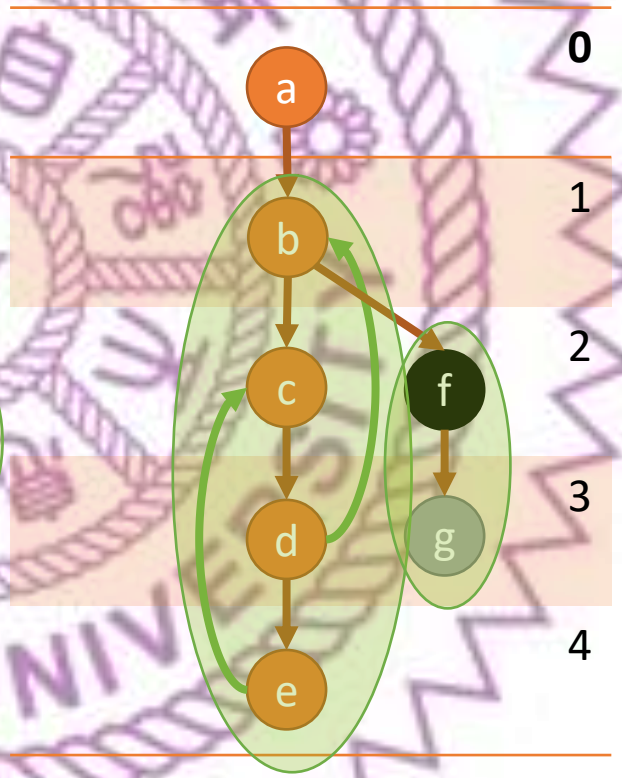
DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	x
low	0	1	1	1	2	2	3	x

stk

low[g] ≥ deep[f]
形成block

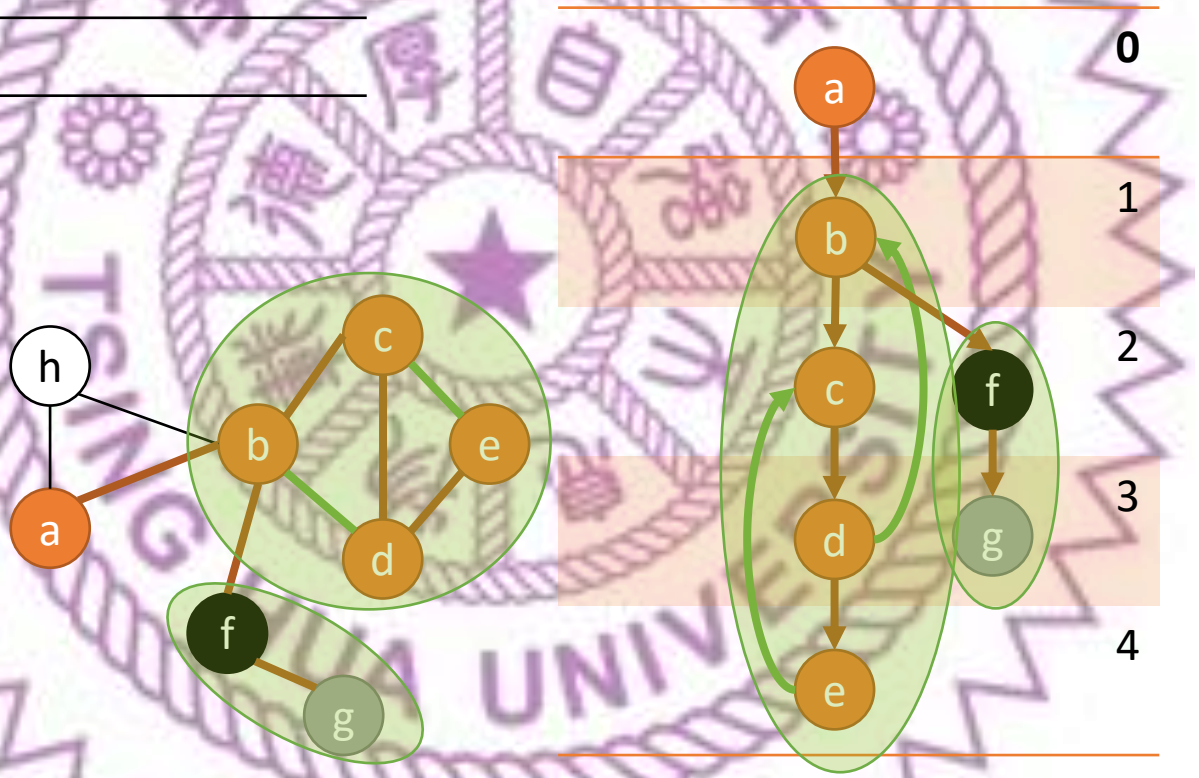




DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	x
low	0	1	1	1	2	2	3	x

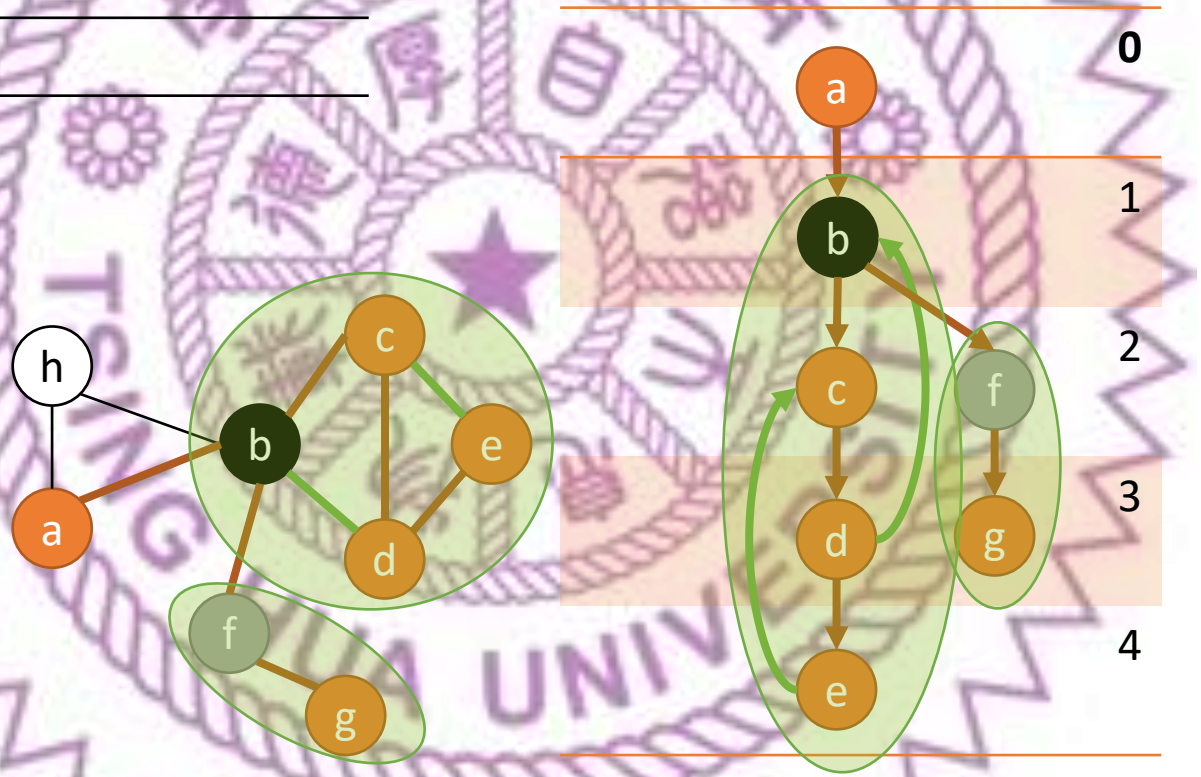
stk | a | b | f |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	x
low	0	1	1	1	2	2	3	x

stk | a | b | f |

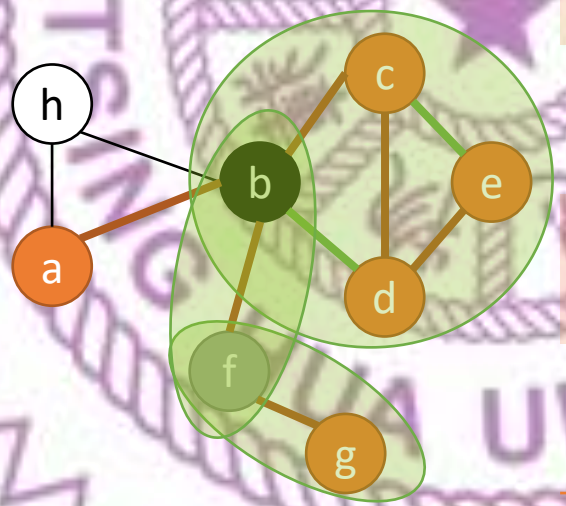


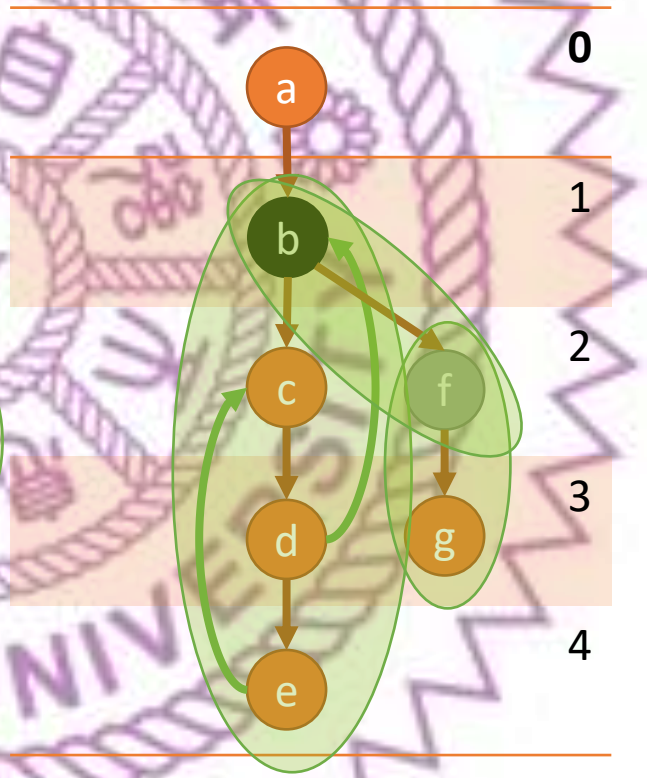
DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	x
low	0	1	1	1	2	2	3	x

stk

low[f] ≥ deep[b]
形成block

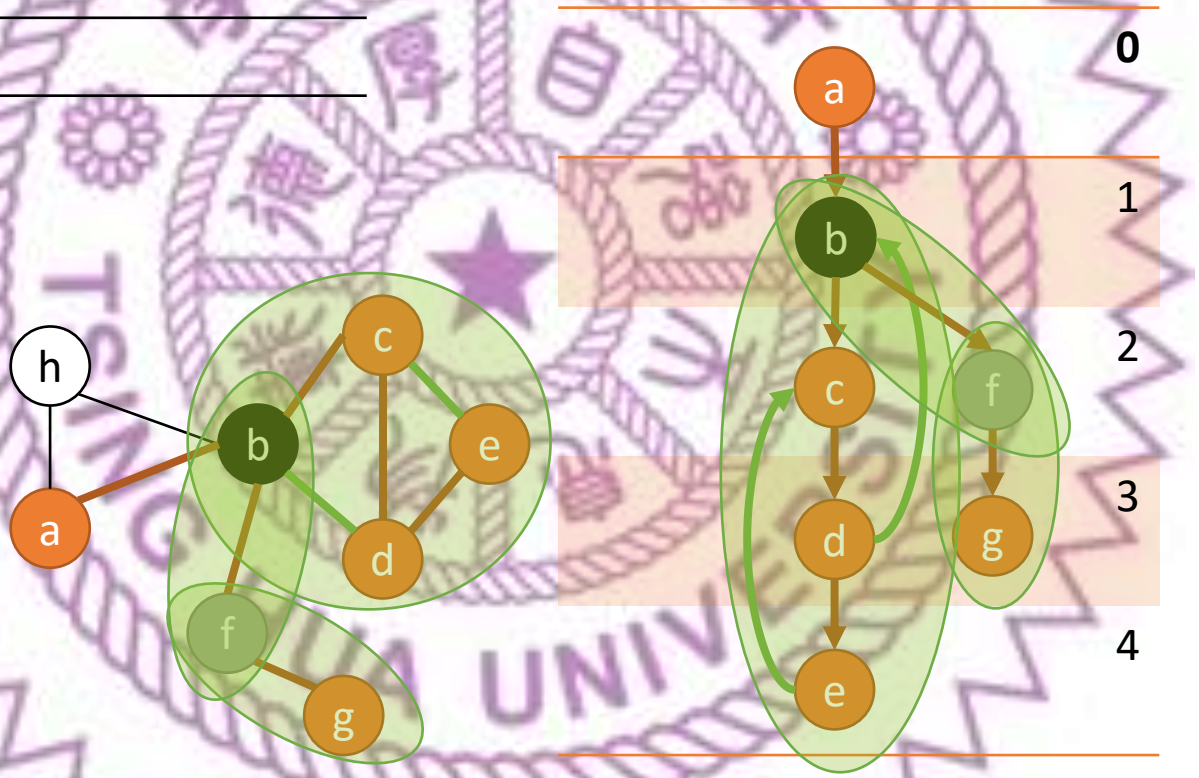




DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	x
low	0	1	1	1	2	2	3	x

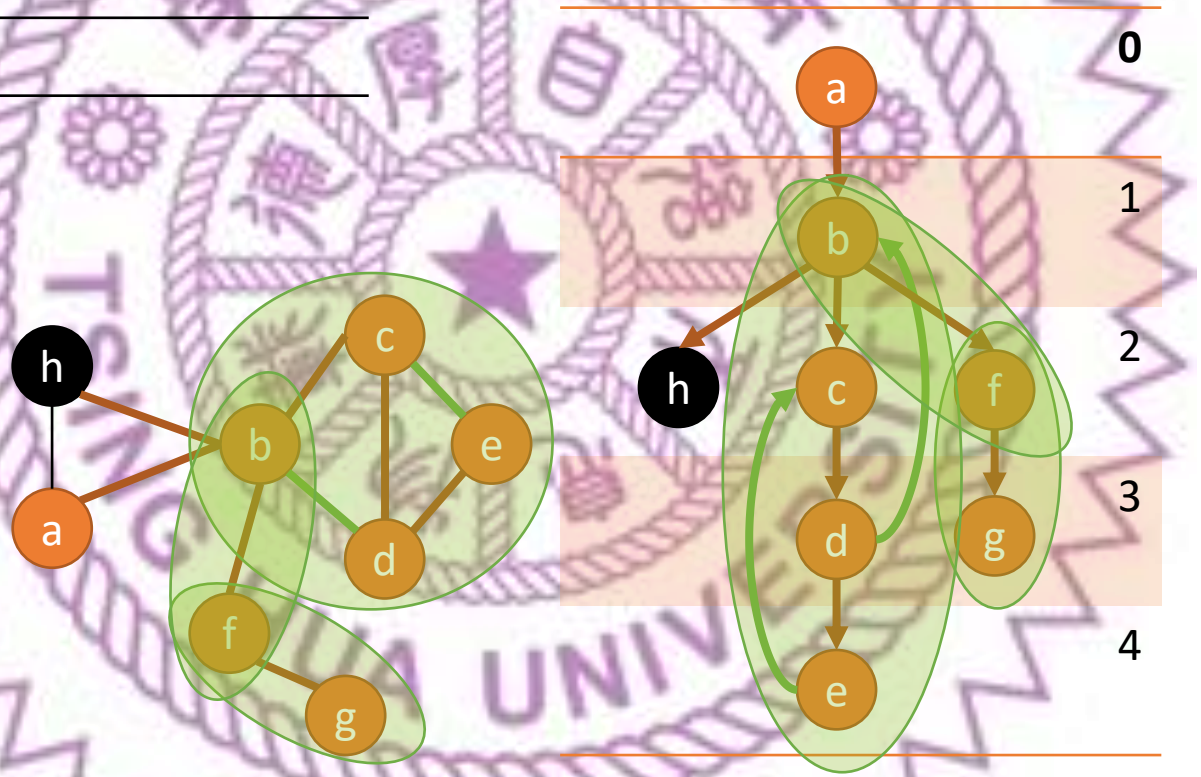
stk | a b |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	1	1	1	2	2	3	2

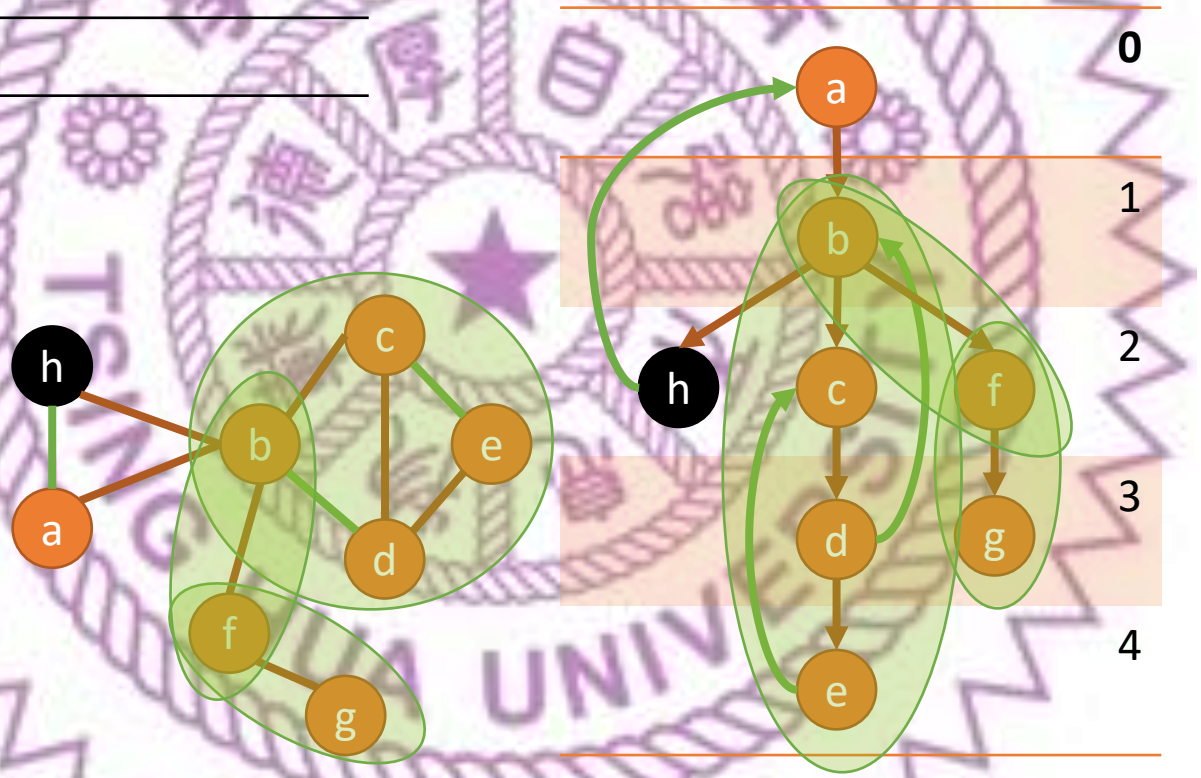
stk | a | b | h |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	1	1	1	2	2	3	0

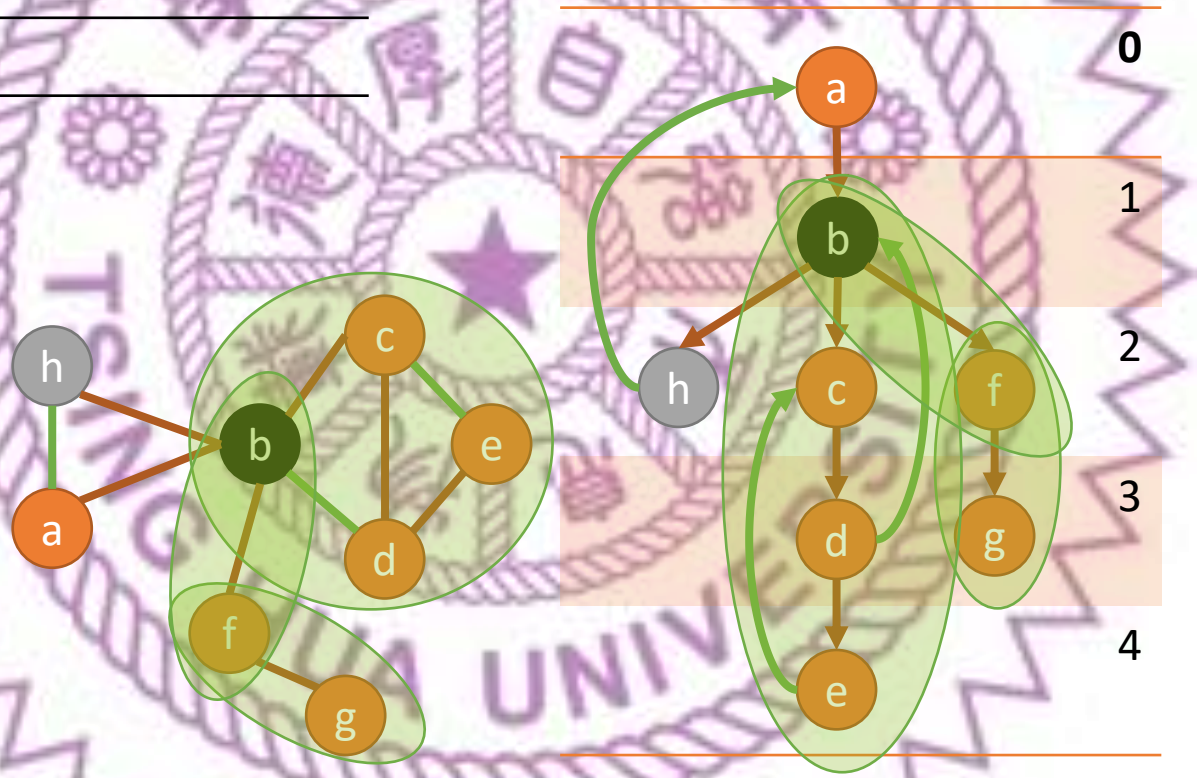
stk | a | b | h |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	1	1	2	2	3	0

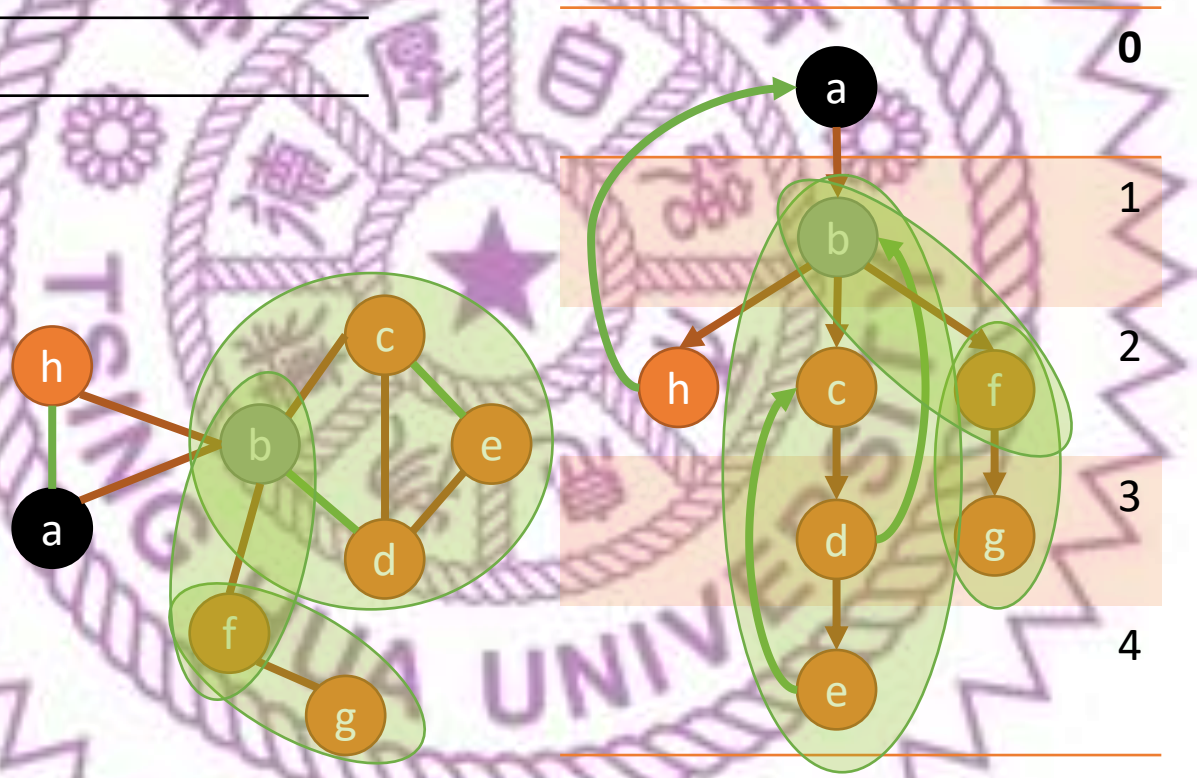
stk | a | b | h |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	1	1	2	2	3	0

stk | a | b | h |

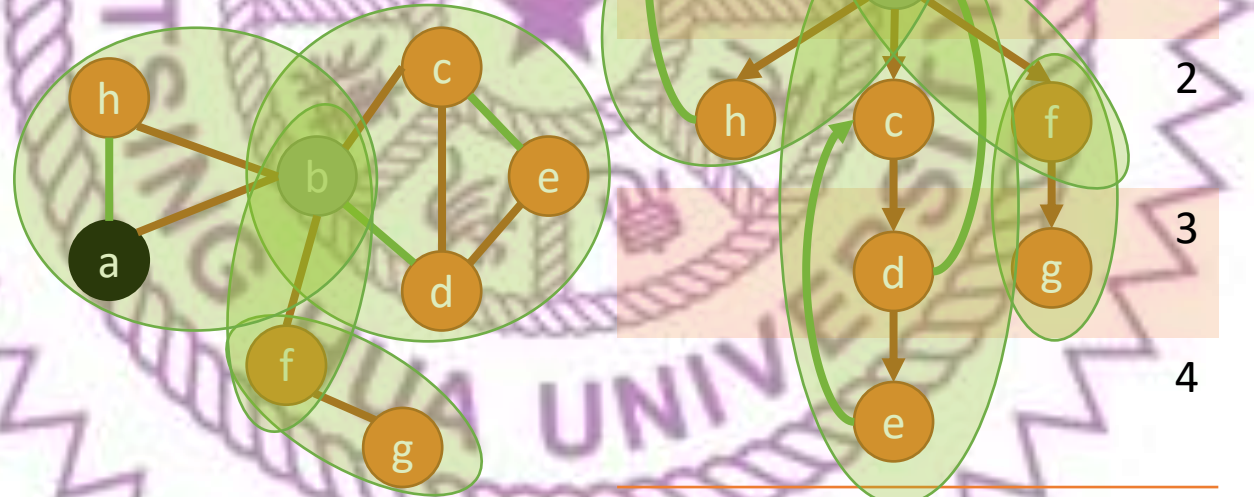


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	1	1	2	2	3	0

stk

low[b] ≥ deep[a]
形成block

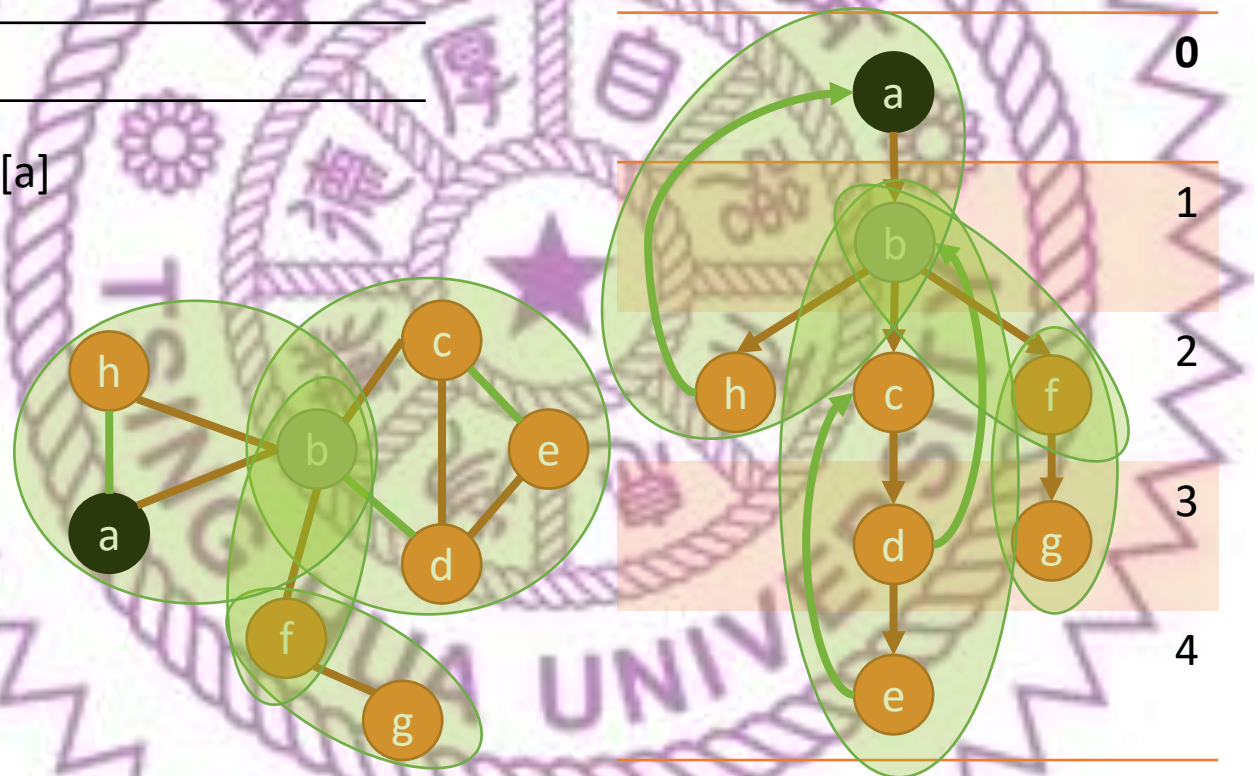


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	1	1	2	2	3	0

stk | a |

low[b] ≥ deep[a]
形成block



DFS 順便找出割點

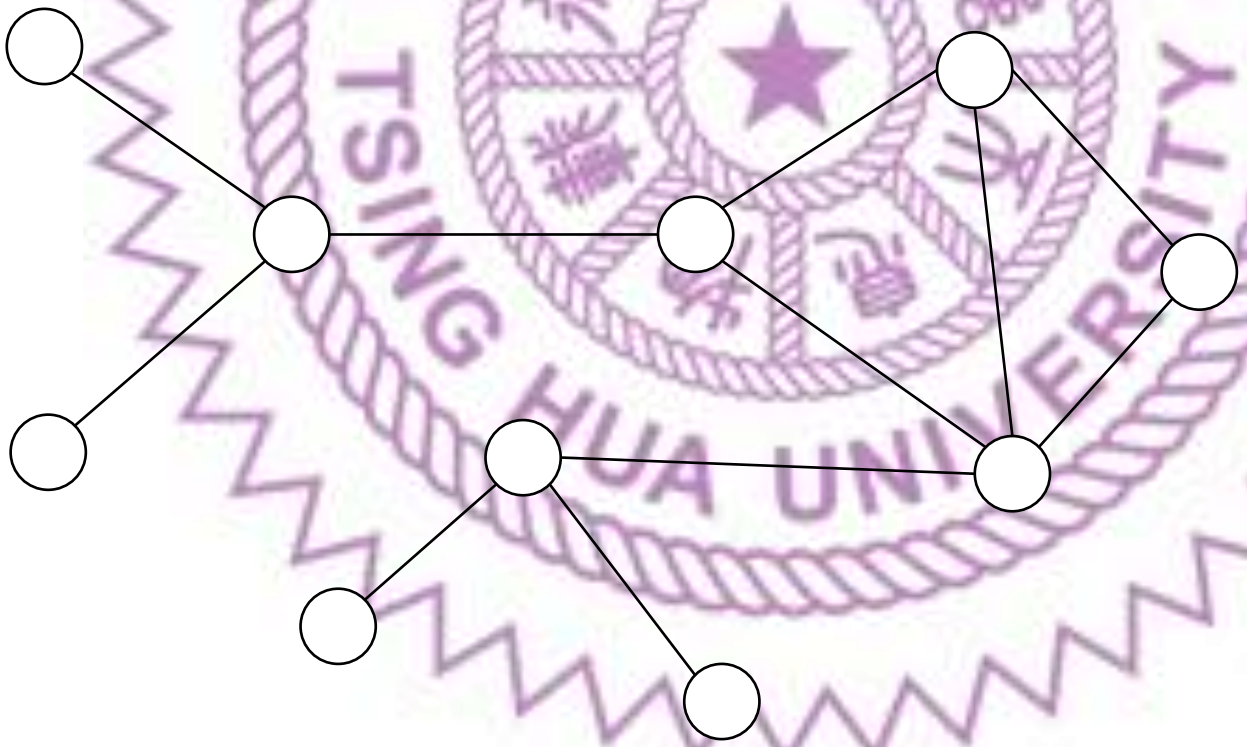
- 在 DFS 的過程中可以順便把割點找出來
- 判斷根結點是不是割點會比較困難
  可以自己想一想

橋雙連通分量

(Bridge) BCC

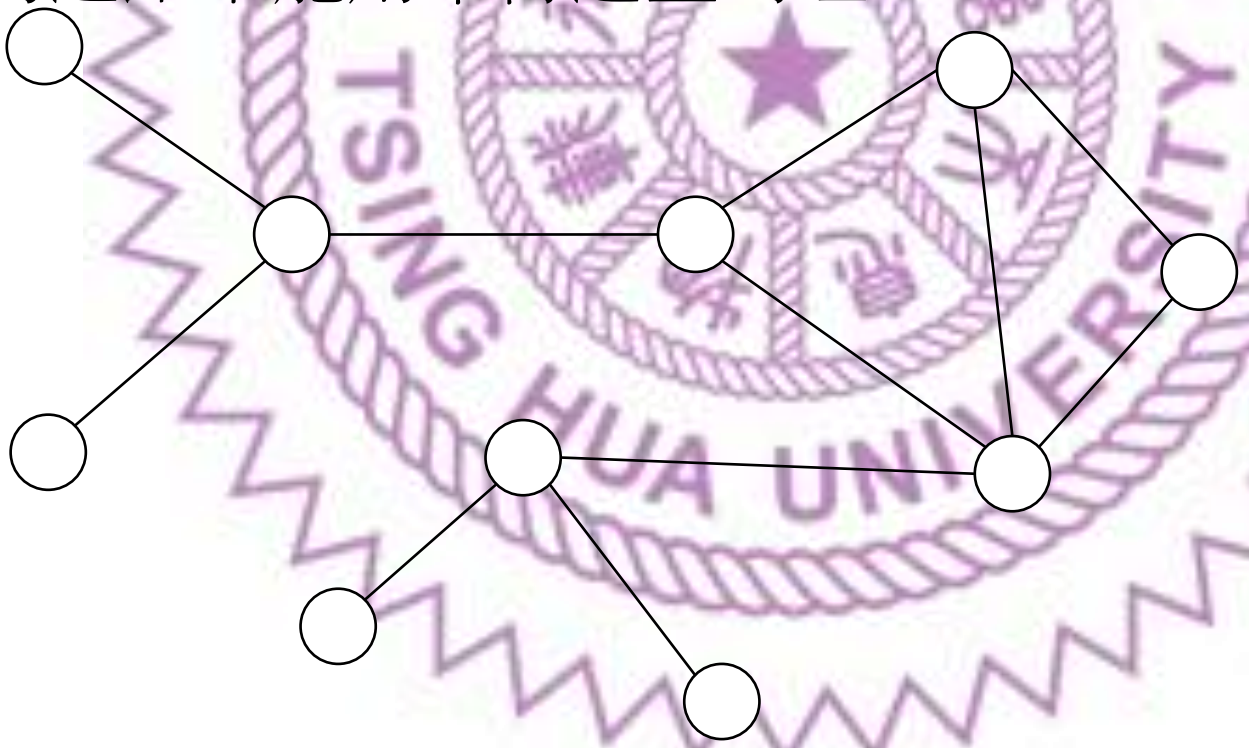
金坷垃運輸問題2

- 身為聖地亞戈的負責人
  你的工作就是讓每座城市之間可以互相傳遞金坷垃



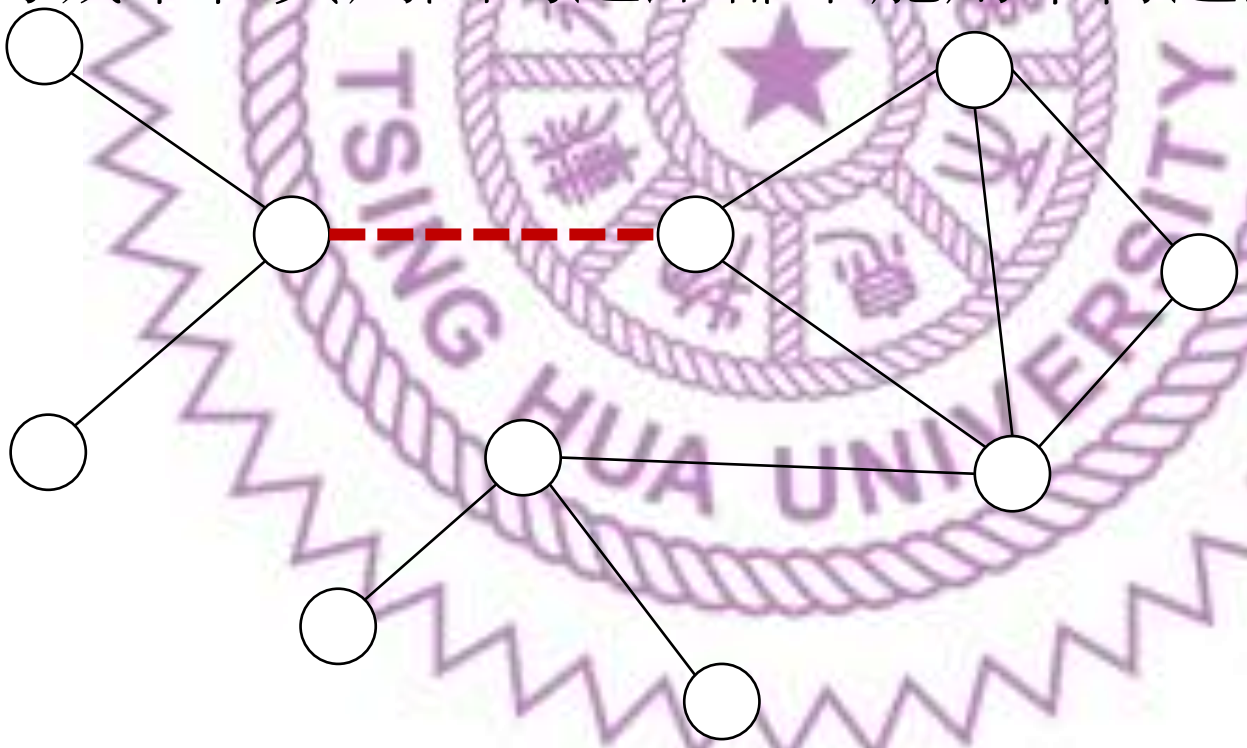
金坷垃運輸問題2

- 現在你有一個危機
  有一個小日本鬼子打算佔領你的任意**一條道路**
- 被佔領的道路不能用來傳遞金坷垃



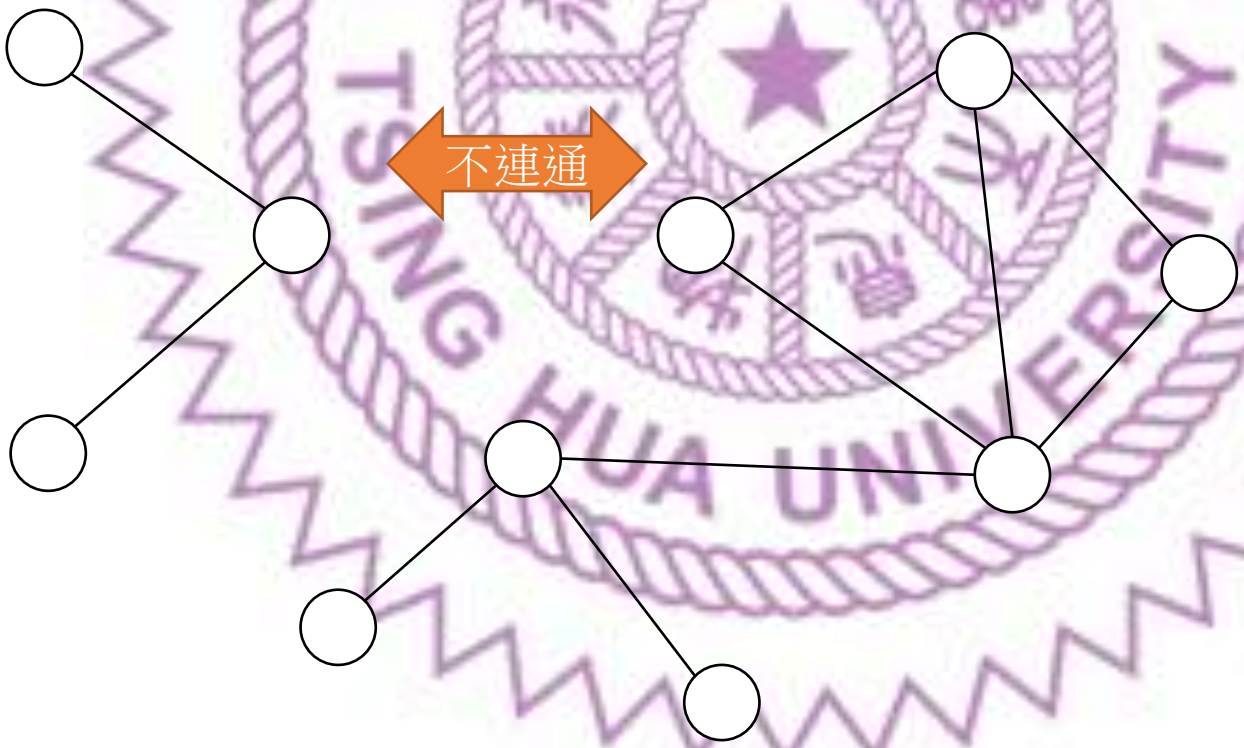
金坷垃運輸問題2

- 現在你有一個危機
  有一個小日本鬼子打算佔領你的任意**一條道路**
- 被佔領的城市和其周圍的道路都不能用來傳遞金坷垃



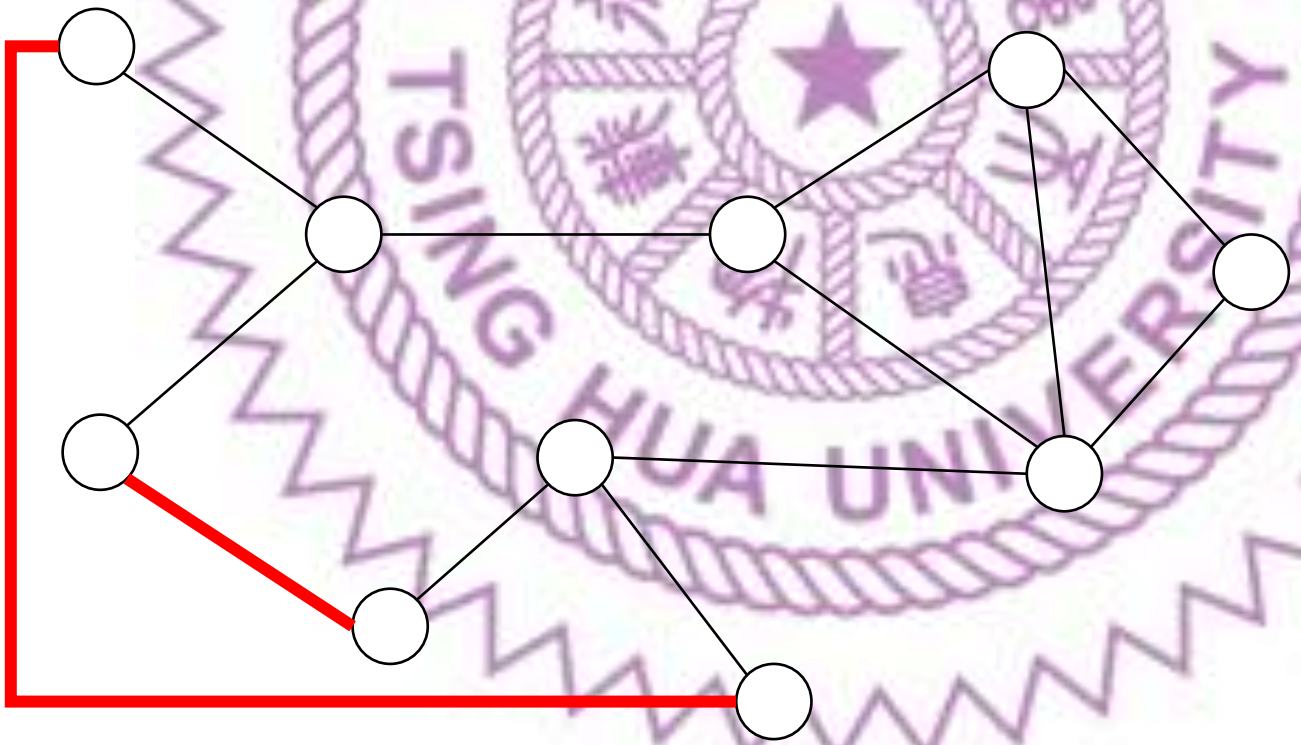
金坷垃運輸問題2

- 這樣可能會導致有些城市之間無法互相傳遞金坷垃平衡就會崩壞



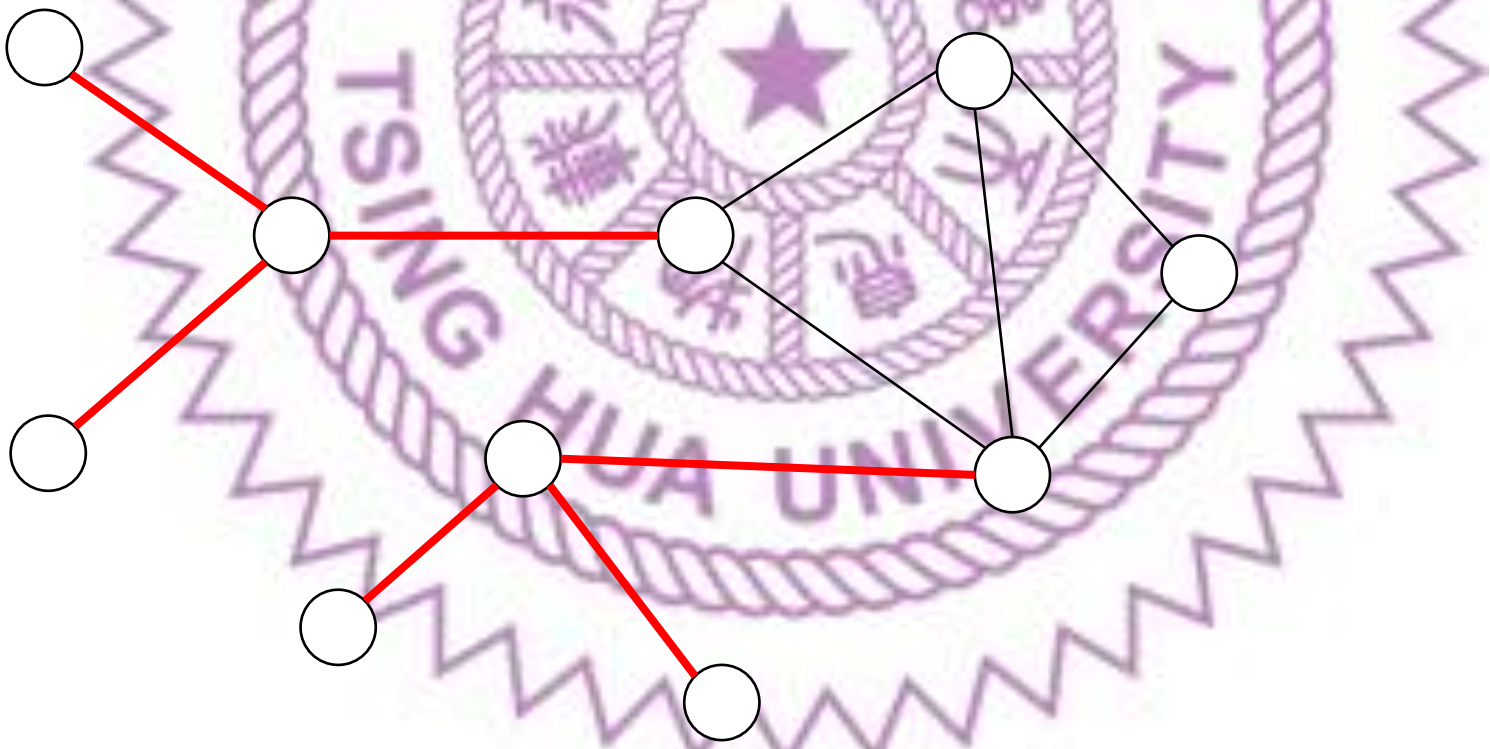
金坷垃運輸問題2

- 請問你最少需要增加幾條路
  才能讓剩下的城市之間能繼續互相傳遞金坷垃？



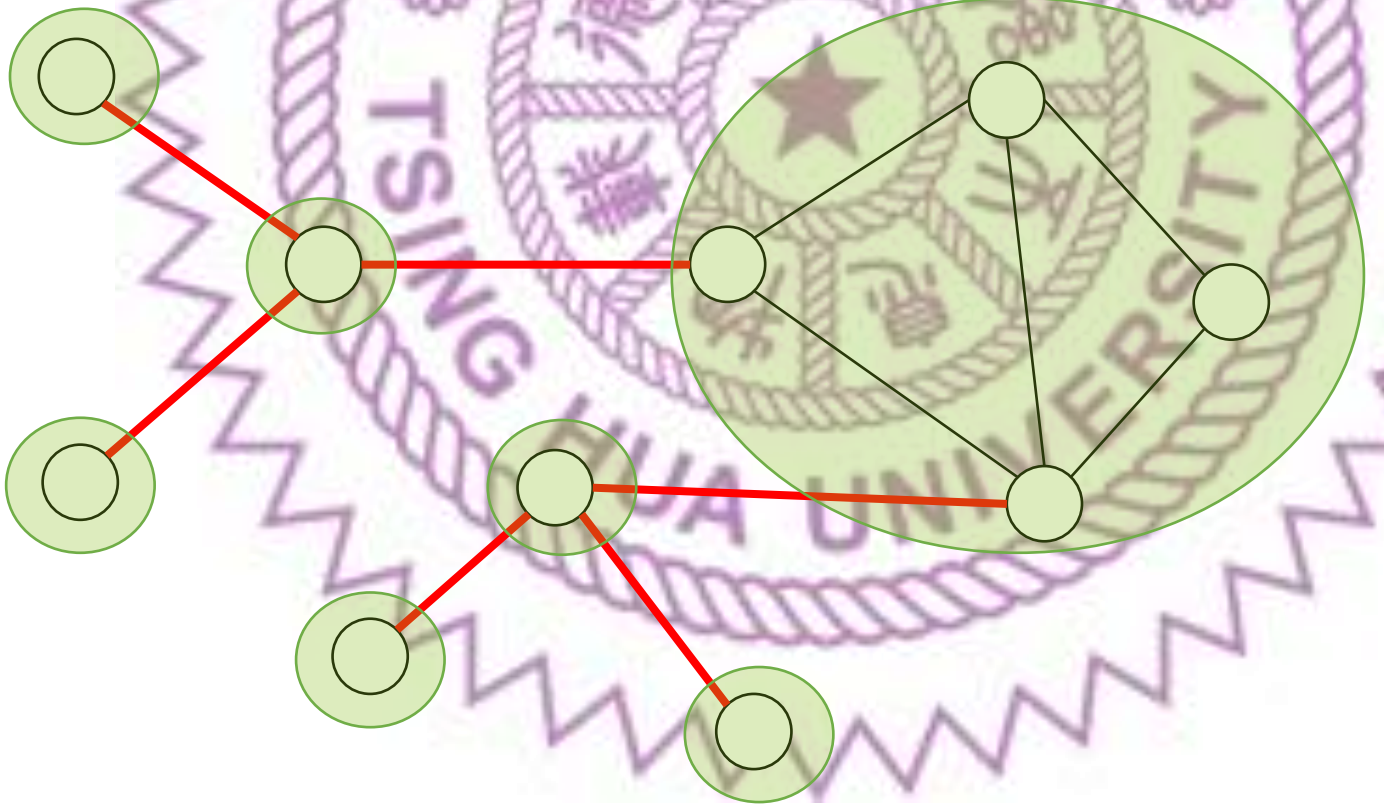
想法

- 類似於割點，可以把橋 (cut-edge) 找出來



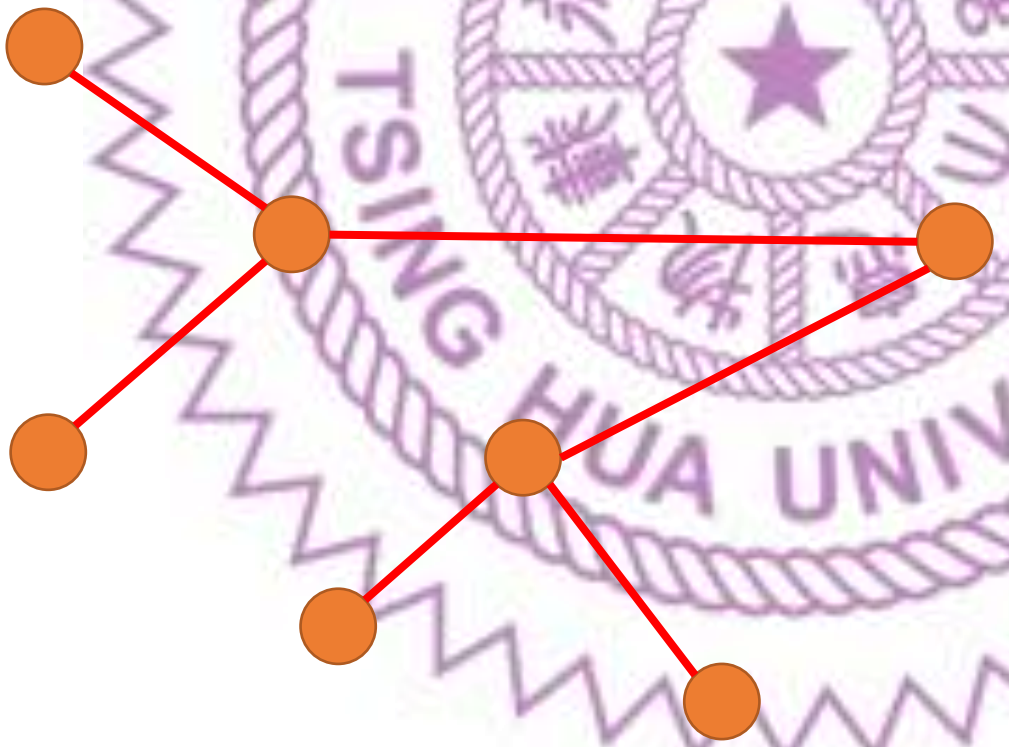
想法

- 也可以用相似的方式定義出 橋連通分量 (bridge) BCC



橋連通樹

- 將橋連通分量縮點後可以得到 橋連通樹

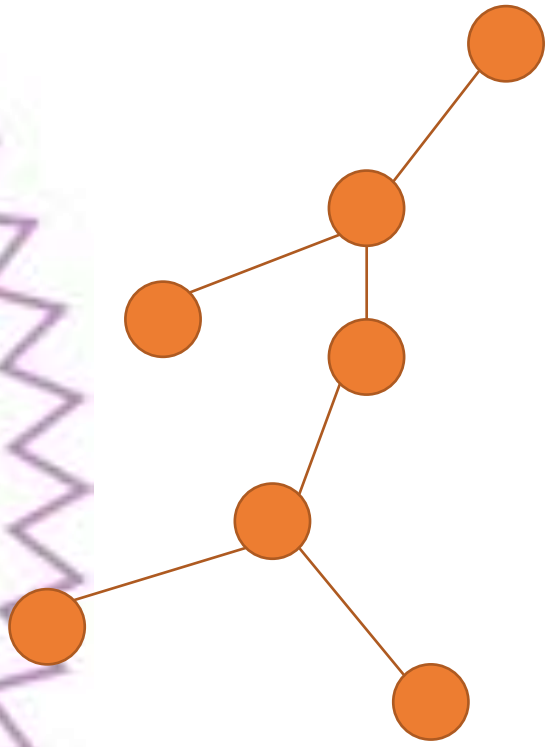


金坷垃運輸問題2

- 可以輕鬆求出這地問題的答案：

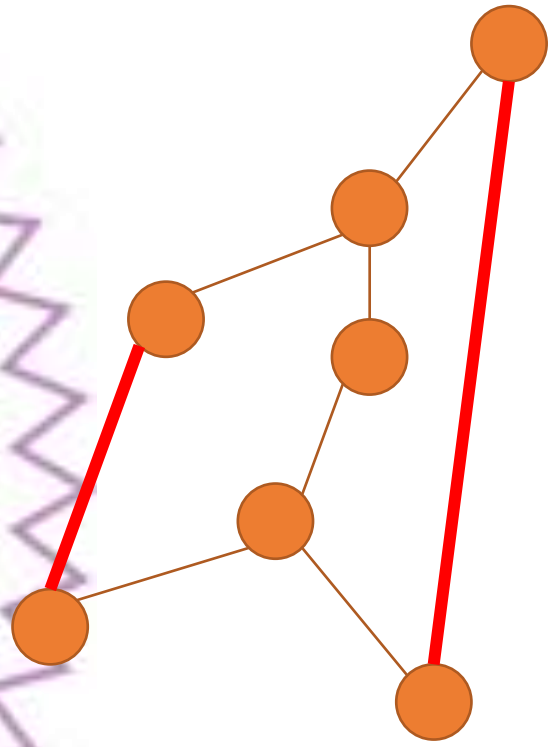
  1.計算該 graph 的橋連通樹 H

  2.問題 reduce 成增加最少的邊使得 H 中沒有橋



金坷垃運輸問題2

- 可以輕鬆求出這地問題的答案：
  1.計算該 graph 的橋連通樹 H
  2.問題 reduce 成增加最少的邊使得 H 中沒有橋
  3.答案是 $\left\lceil \frac{\text{葉節點的數量}}{2} \right\rceil$



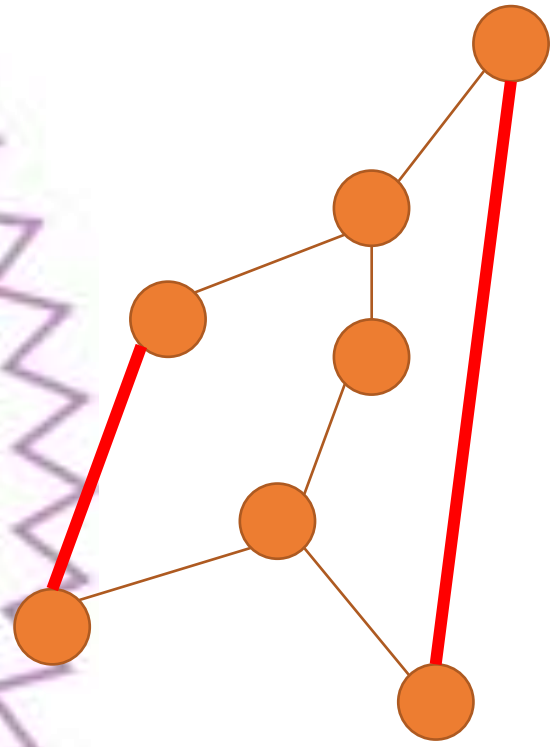
金坷垃運輸問題2

- 可以輕鬆求出這地問題的答案：

  1.計算該 graph 的橋連通樹 H

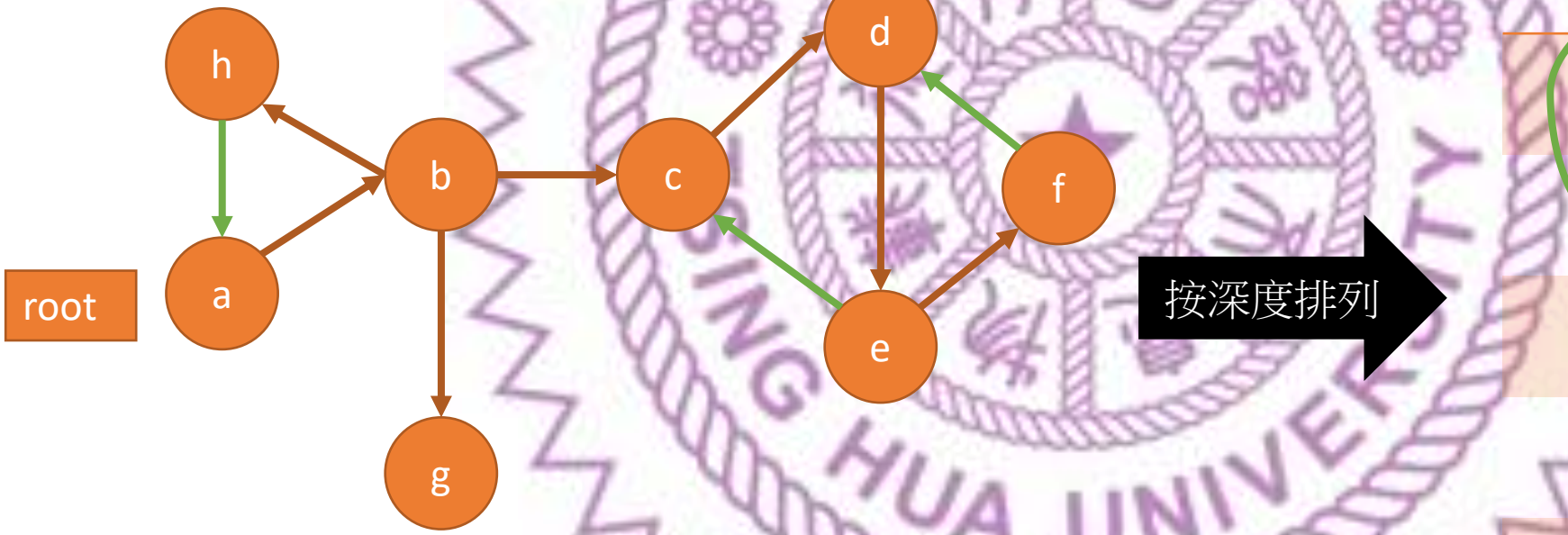
  2.問題 reduce 成增加最少的邊使得 H 中沒有橋

  3.答案是 $\left\lceil \frac{\text{葉節點的數量}}{2} \right\rceil$

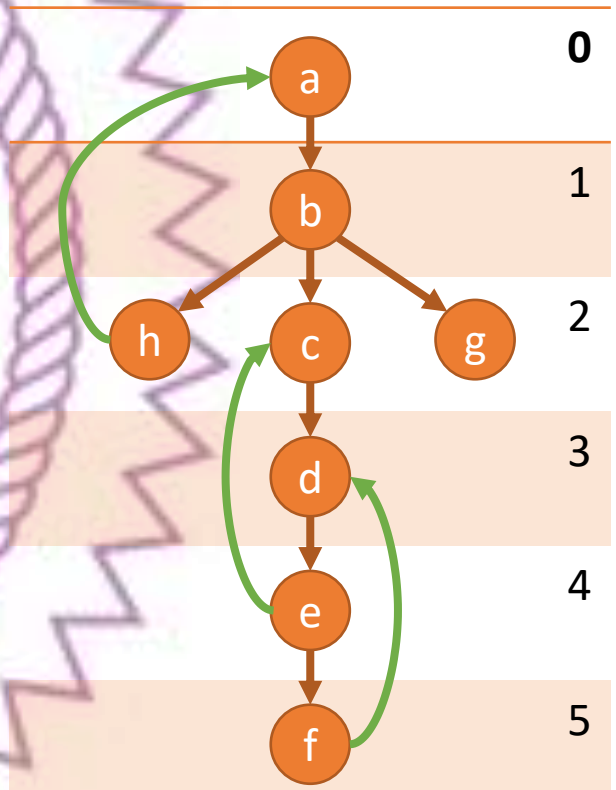


- 如果能把所有**橋**找出來，那就能求出**橋連通樹**了！

重新檢視 DFS tree



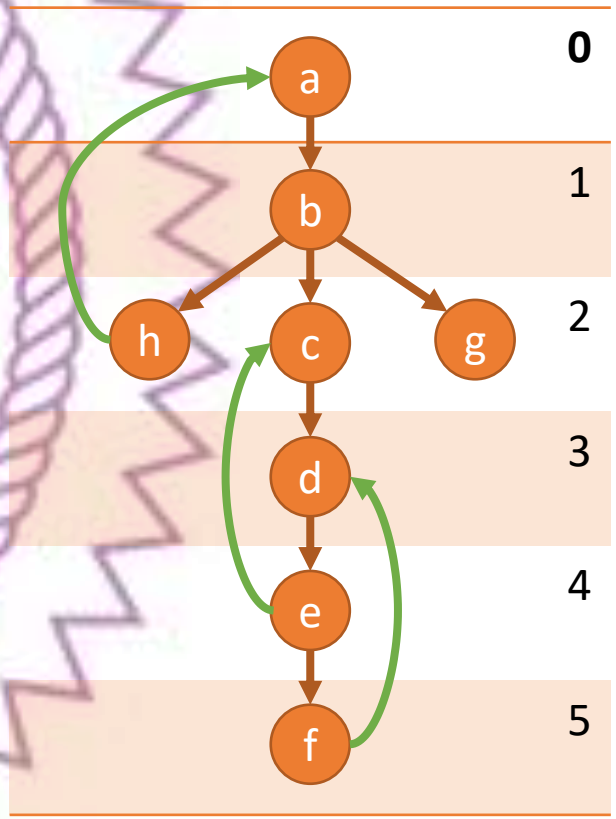
按深度排列



LOW

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	2	2	2	2	3	0

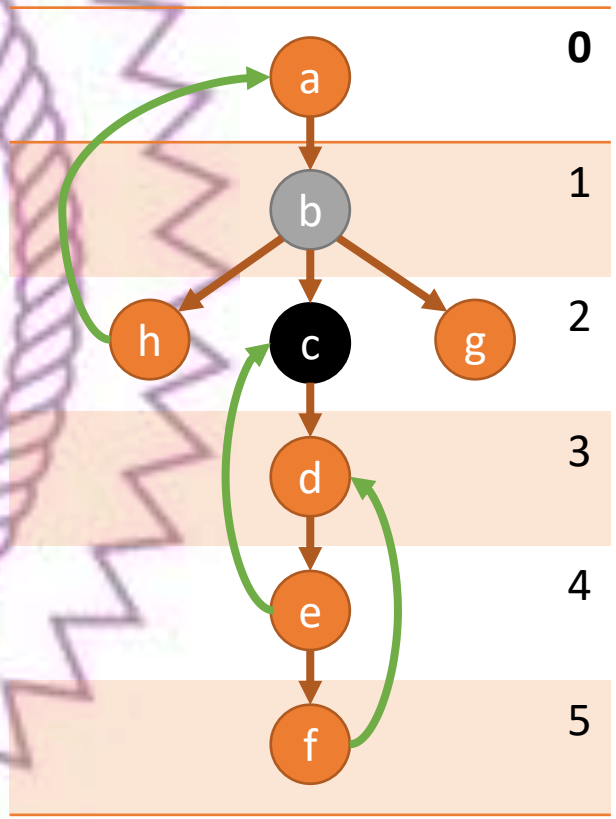
- 紀錄與點雙連通分量相同的 low



LOW

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	2	2	2	2	3	0

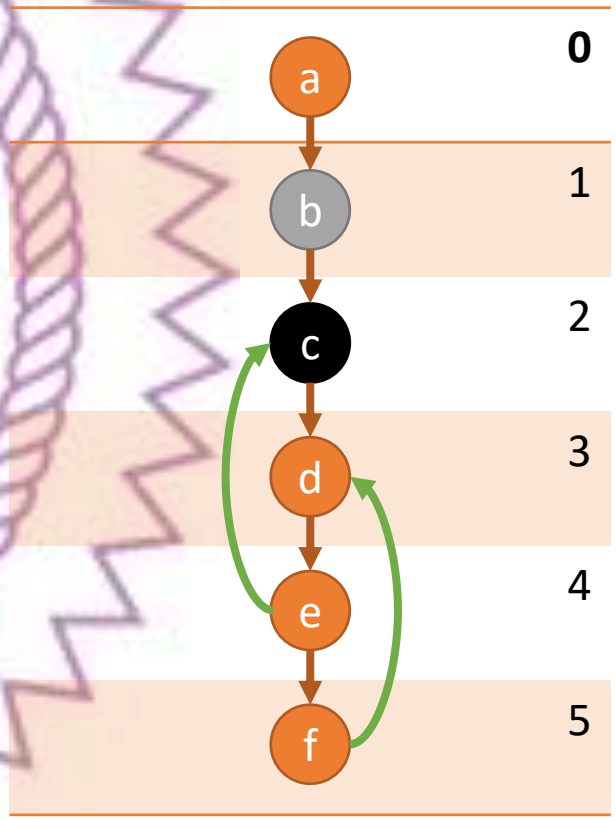
- 設 v 是 u 小孩
- (u, v) 是橋 $\Leftrightarrow low[v] = deep[v]$
- 例如右圖
  - $u = b$
  - $v = c$



LOW

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	2	2	2	2	3	0

- 由於所有綠色邊都只會指向自己的祖先
- 根據 low 的定義可以知道從 c 出發
  沒辦法透過任何一條邊到達 b

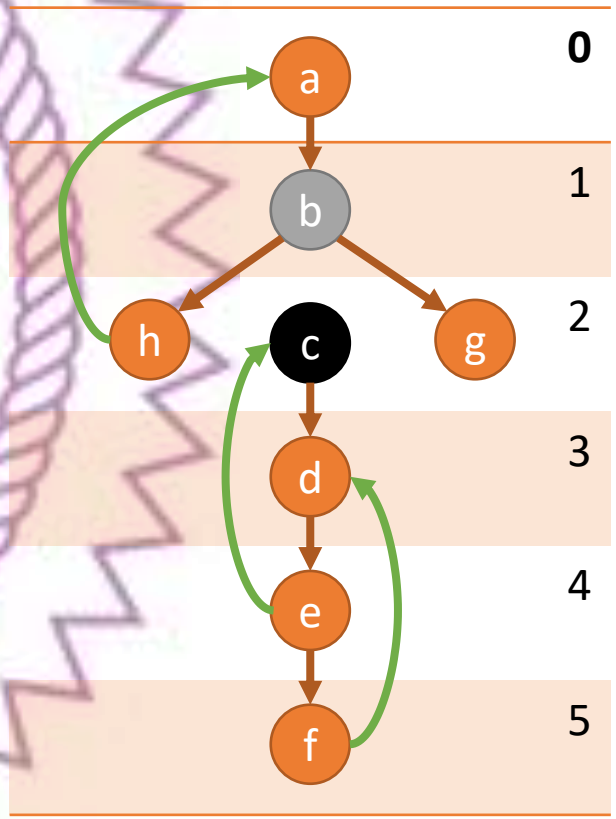


LOW

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	2	3	2
low	0	0	2	2	2	2	3	0

- 這樣把 $b \rightarrow c$ 的邊砍掉
  c 的聯通塊就會和原來的圖完全分開

- 可以確定 $b \rightarrow c$ 的邊是**橋**



Pseudocode c++ style

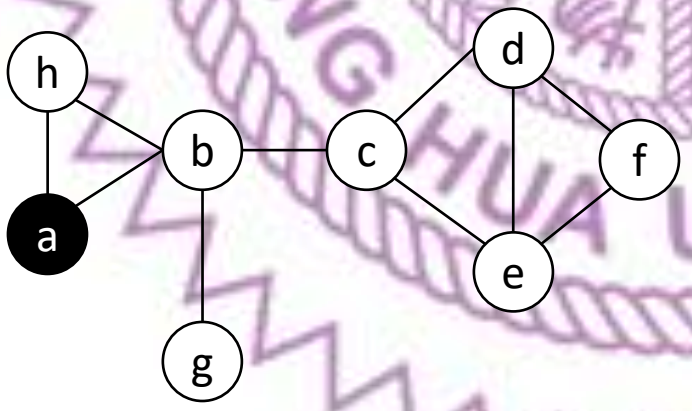
- 與雙連通分量的差異在於 stack pop 的地方位於 for 外面

```
void DFS(Vertex u, Vertex pa, Number d) {
  visit[u] := true;
  deep[u] := d;
  low[u] := d;
  stk.push(u);
  for (Vertex v : neighbors[u]) {
    if (not visit[v]) {
      DFS(v, u, d + 1);
      low[u] := min(low[u], low[v]);
    } else if (deep[v] < deep[u] and v ≠ pa) {
      low[u] := min(low[u], deep[v]);
    }
  }
  if (d == low[u]) {
    B := ∅;
    Vertex x;
    do {
      x := stk.top();
      B := B ∪ x;
      stk.pop();
    } while (x ≠ u);
    BCC := BCC ∪ B;
  }
}
```

DFS

	a	b	c	d	e	f	g	h
$deep$	0	x	x	x	x	x	x	x
low	0	x	x	x	x	x	x	x

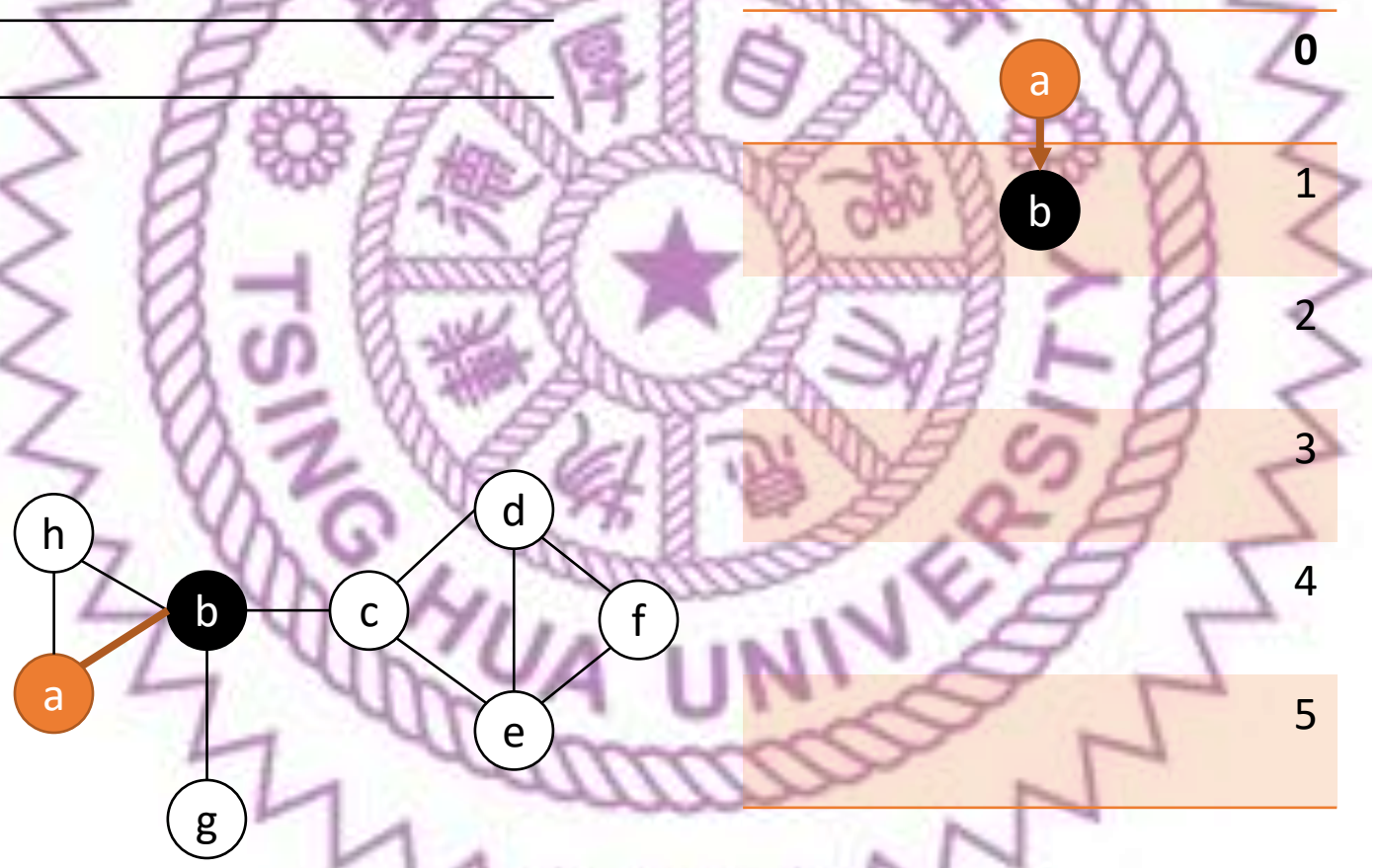
stk | a |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	x	x	x	x	x	x
low	0	1	x	x	x	x	x	x

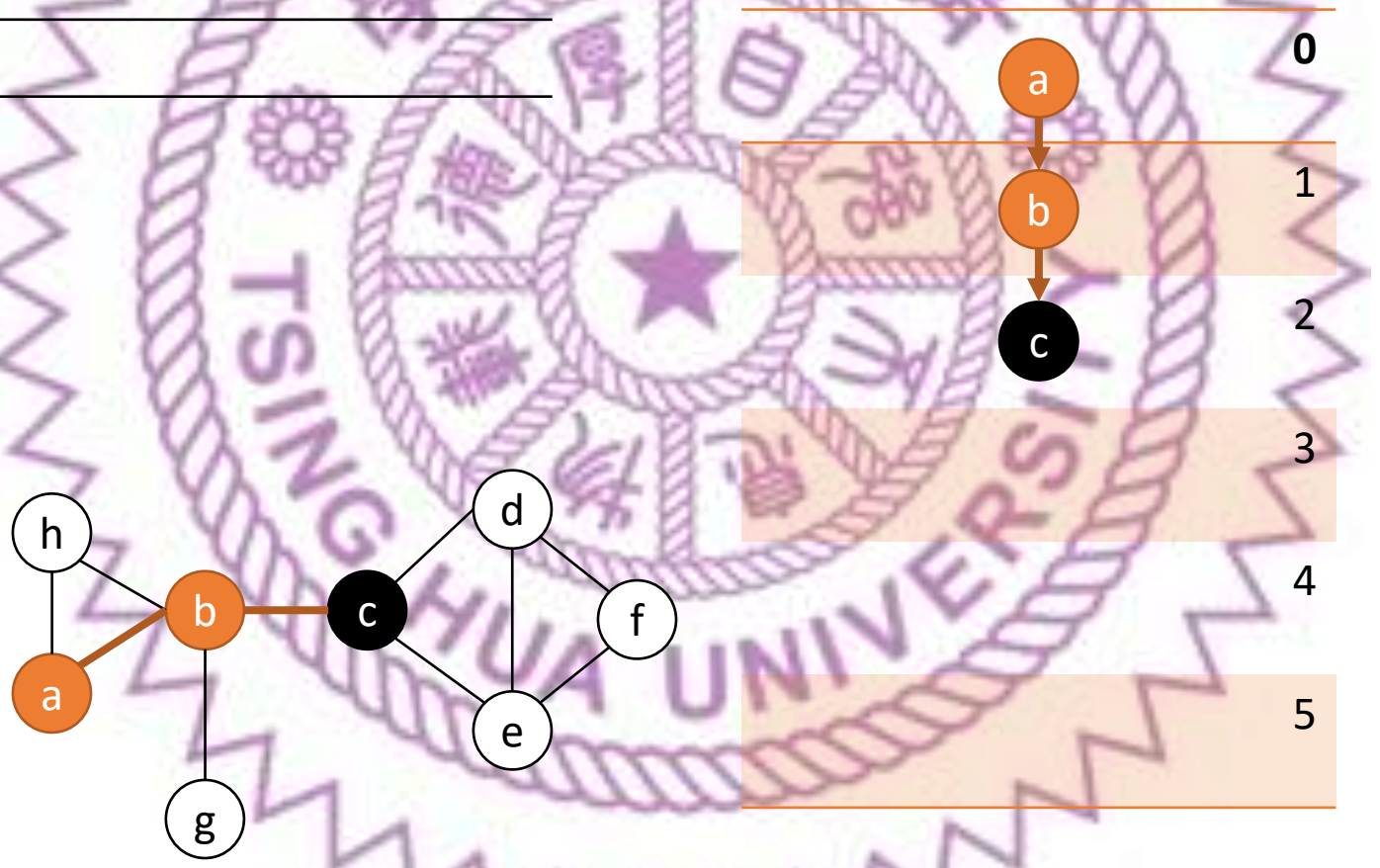
stk | a | b |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	x	x	x	x	x
low	0	1	2	x	x	x	x	x

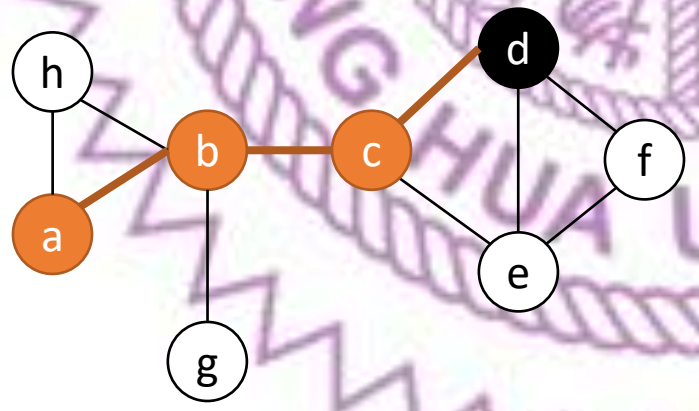
stk | a | b | c |

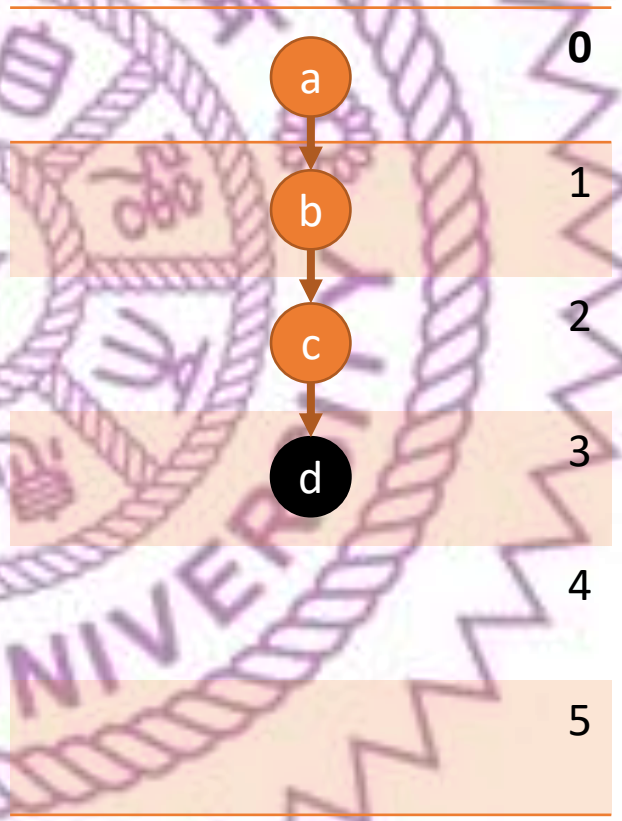


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	x	x	x	x
low	0	1	2	3	x	x	x	x

stk | a | b | c | d |

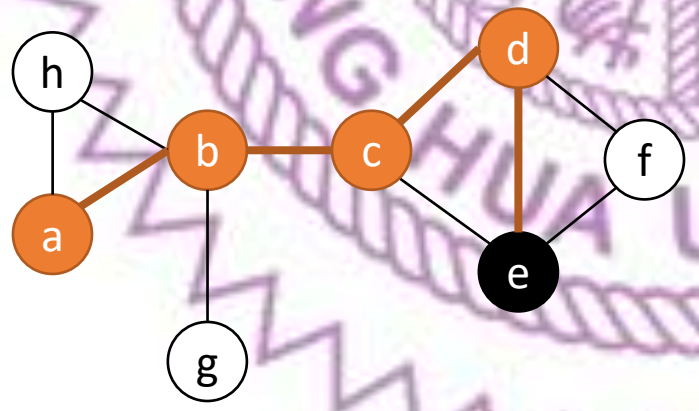


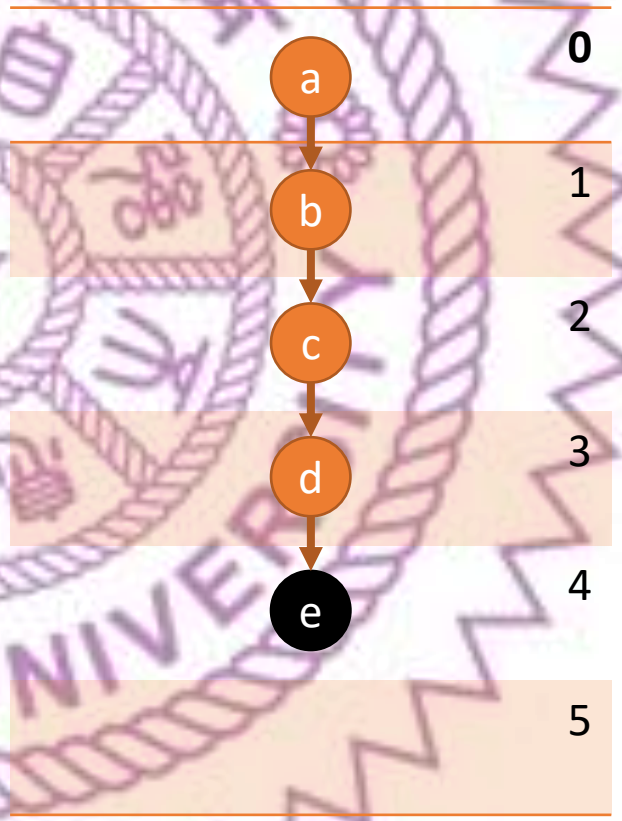


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	x	x	x
low	0	1	2	3	4	x	x	x

stk | a | b | c | d | e |

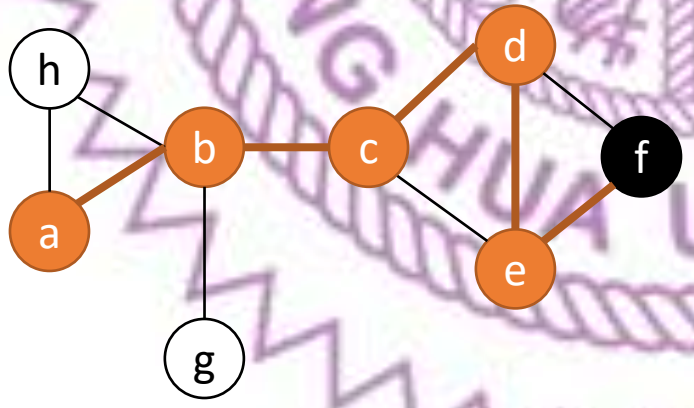


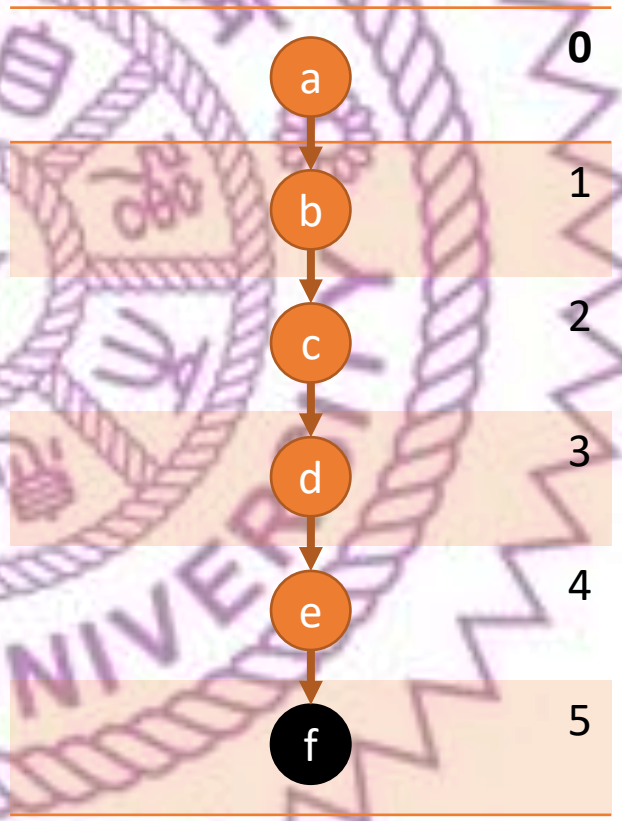


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	x	x
low	0	1	2	3	4	5	x	x

stk | a | b | c | d | e | f |

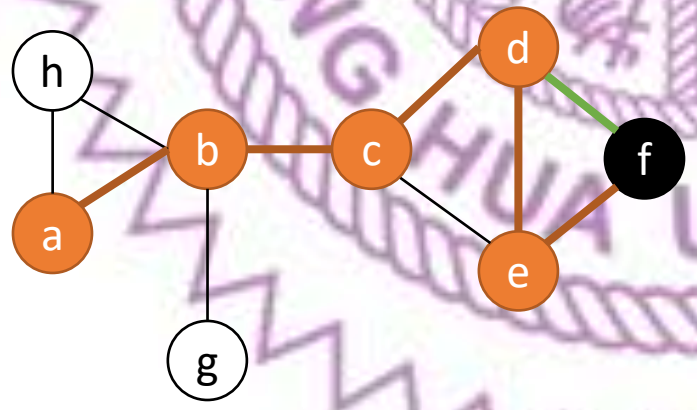


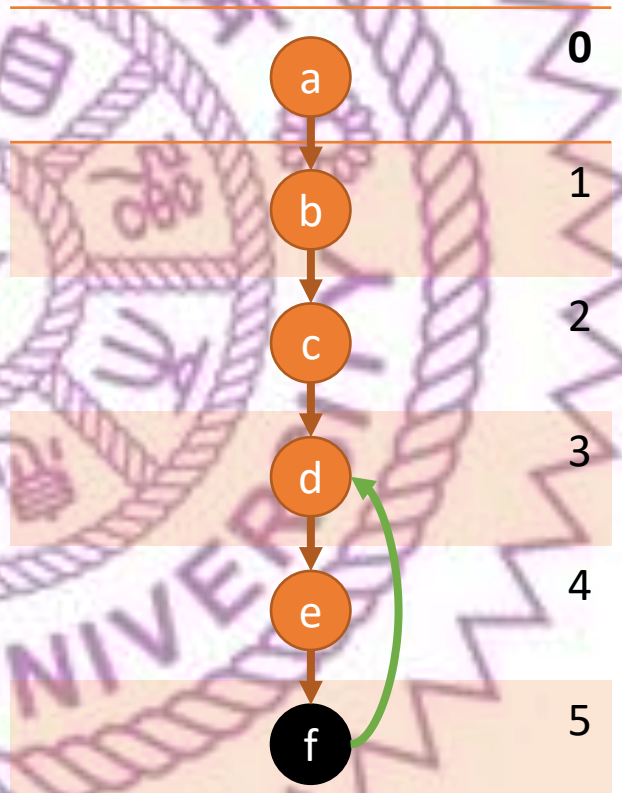


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	x	x
low	0	1	2	3	4	3	x	x

stk | a | b | c | d | e | f |

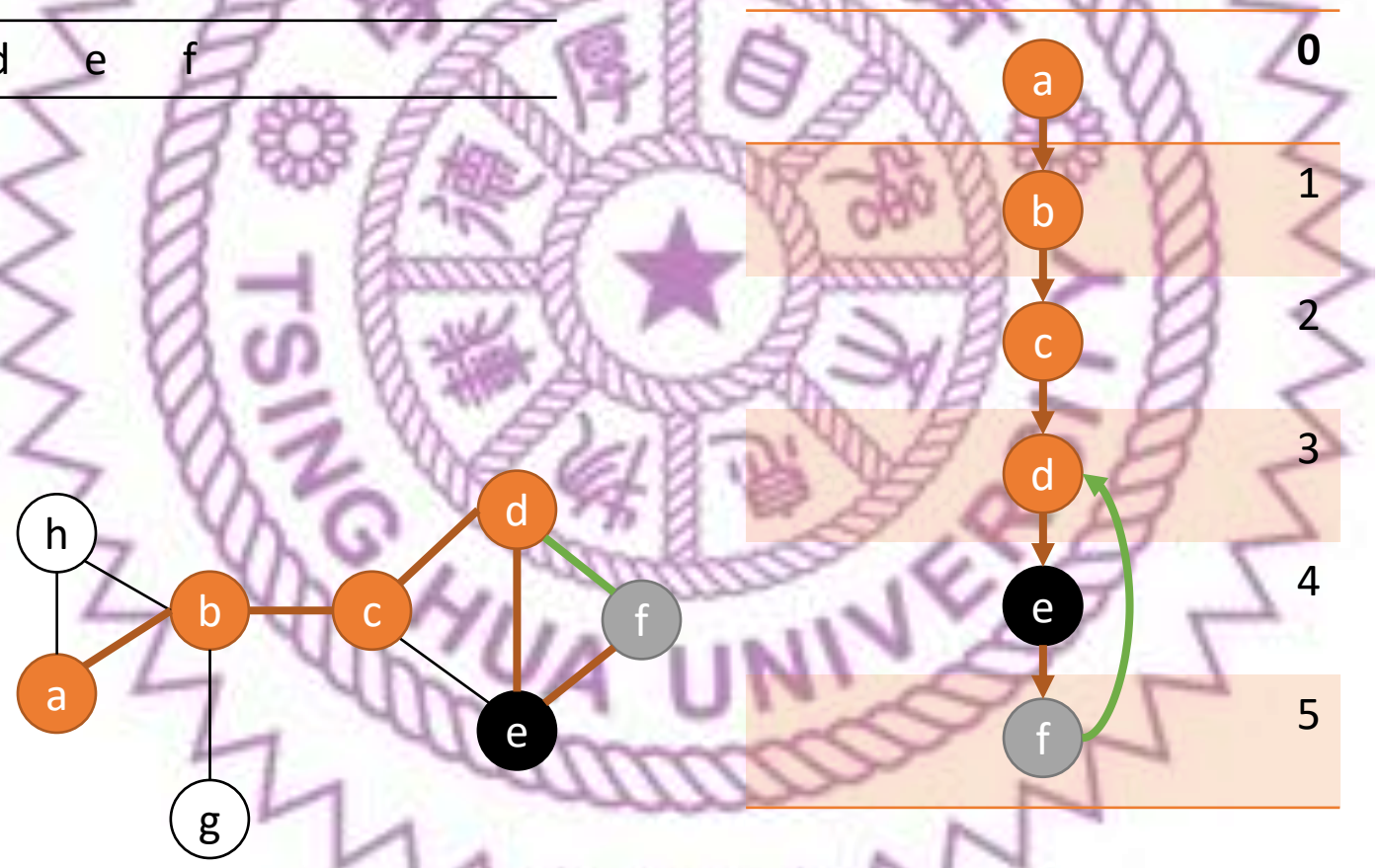




DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	x	x
low	0	1	2	3	3	3	x	x

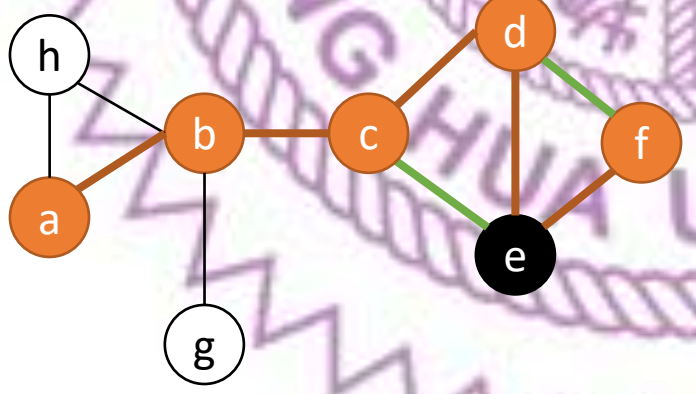
stk | a | b | c | d | e | f |

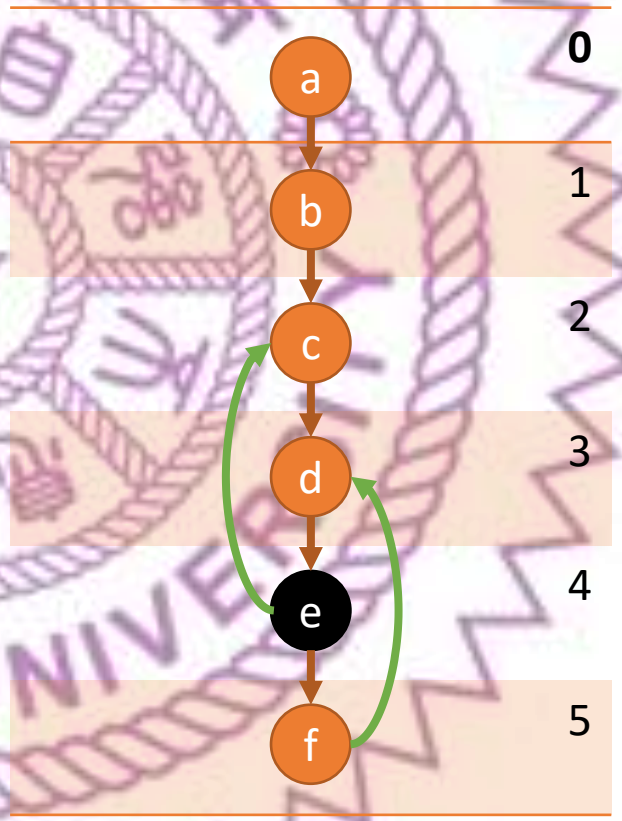


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	x	x
low	0	1	2	3	2	3	x	x

stk | a | b | c | d | e | f |

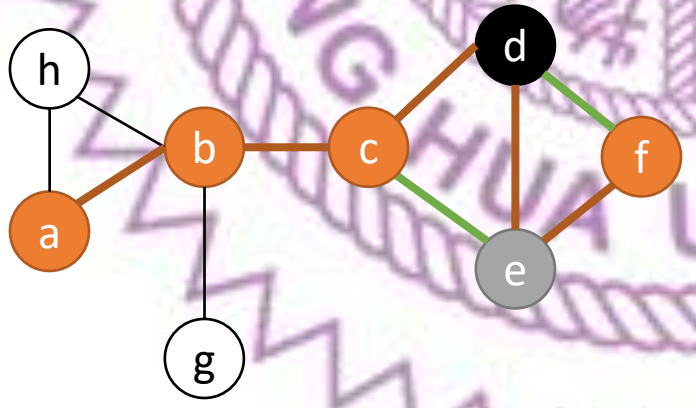


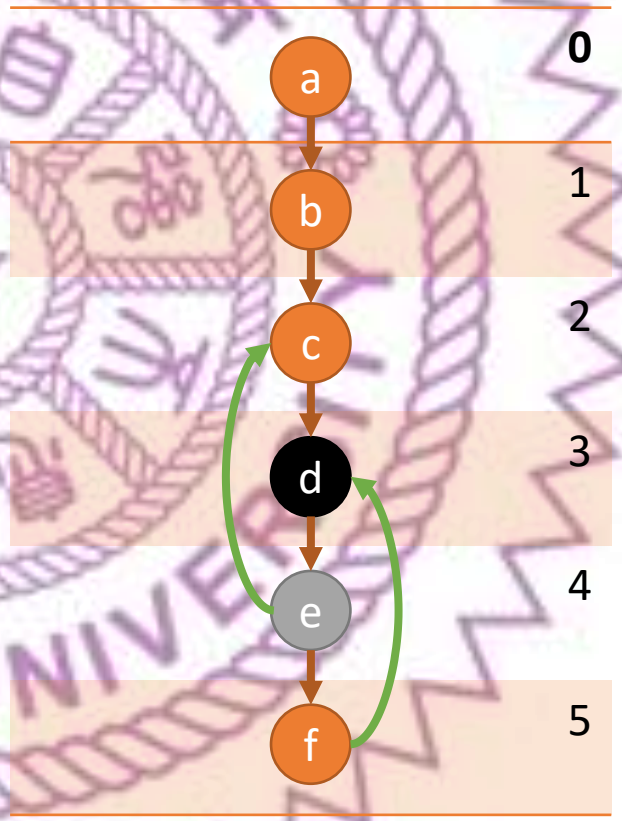


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	x	x
low	0	1	2	2	2	3	x	x

stk | a | b | c | d | e | f |

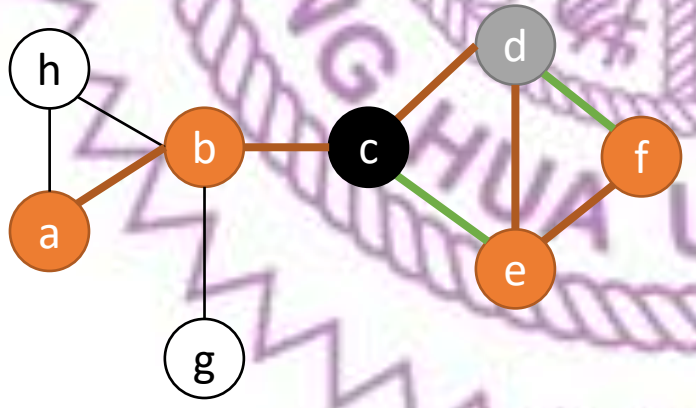


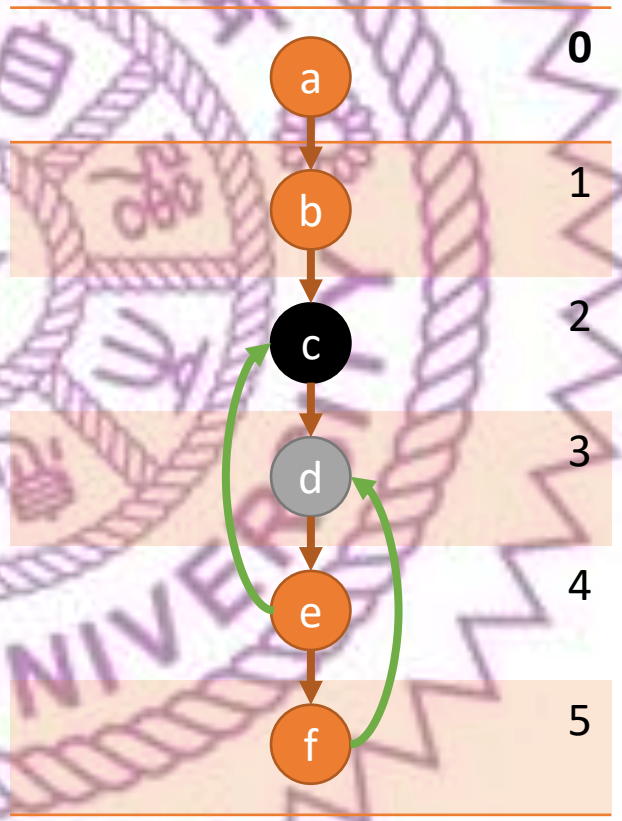


DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	x	x
low	0	1	2	2	2	3	x	x

stk | a | b | c | d | e | f |



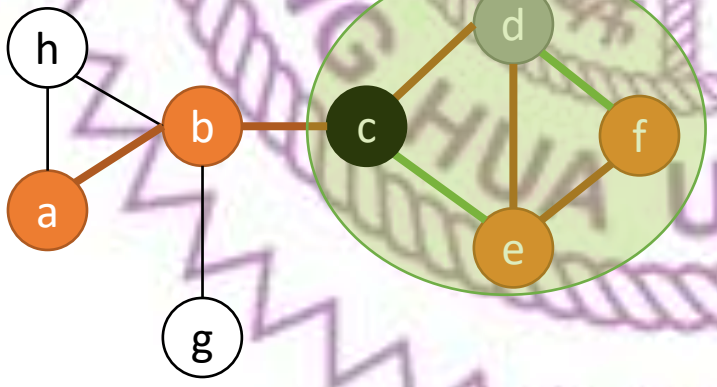


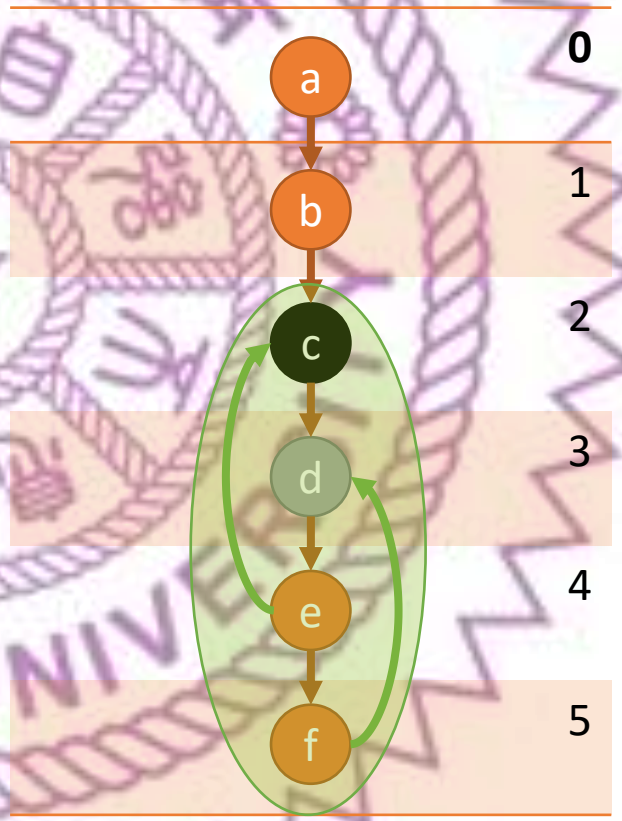
DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	x	x
low	0	1	2	2	2	3	x	x

stk | a | b | c | d | e | f |

low[c] = deep[c]
(b, c)是橋
形成橋連通分量

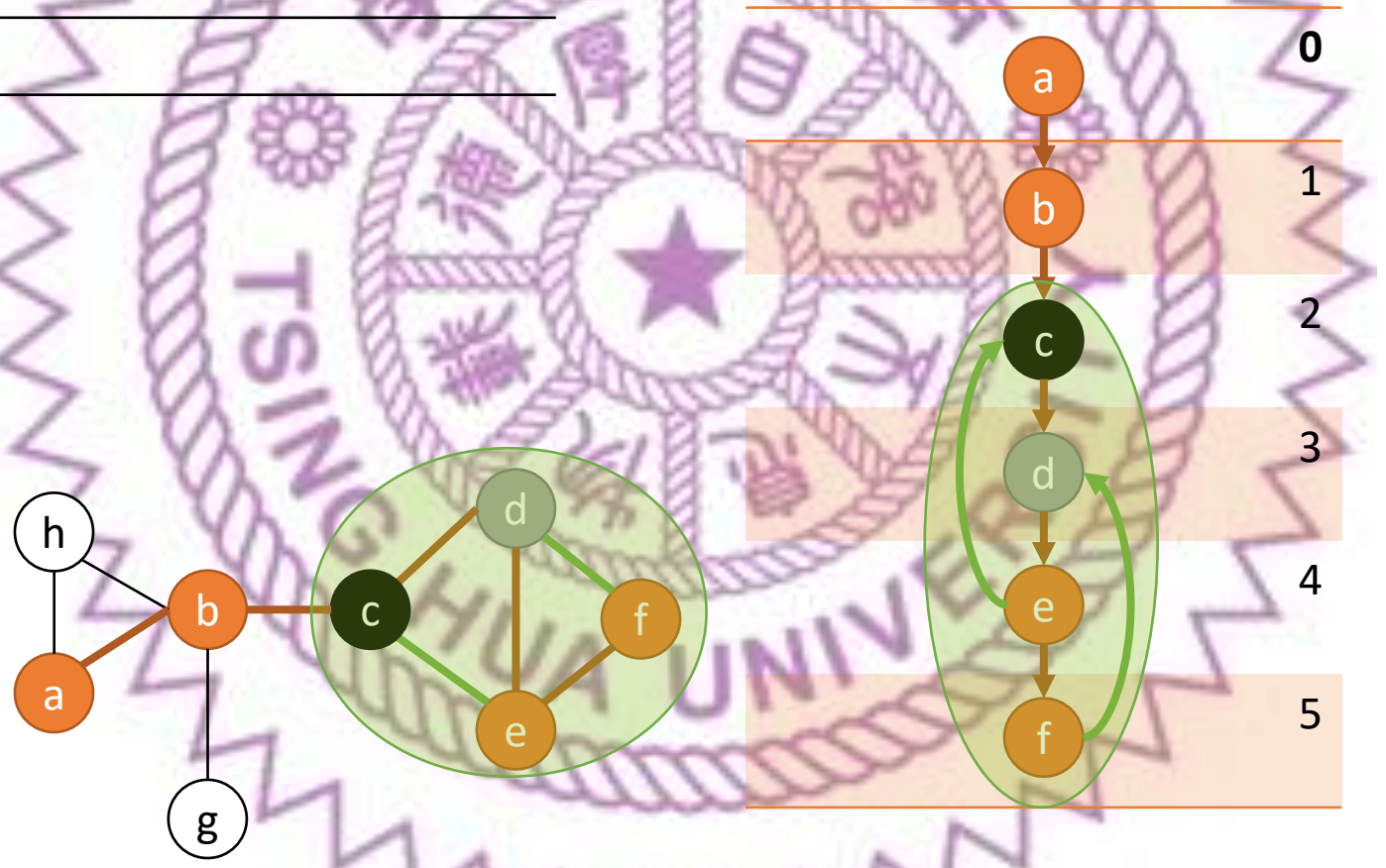




DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	x	x
low	0	1	2	2	2	3	x	x

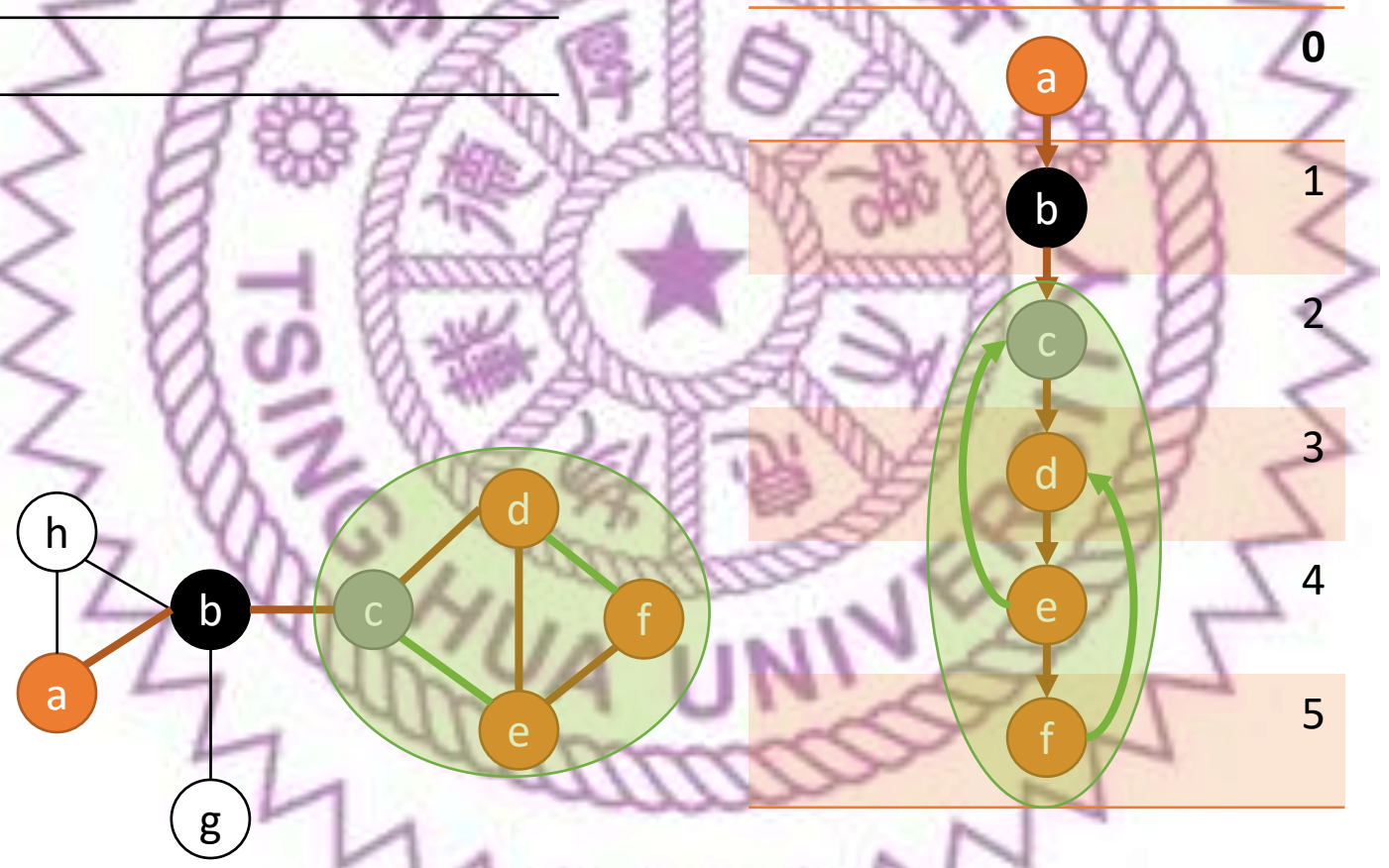
stk | a | b |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	x	x
low	0	1	2	2	2	3	x	x

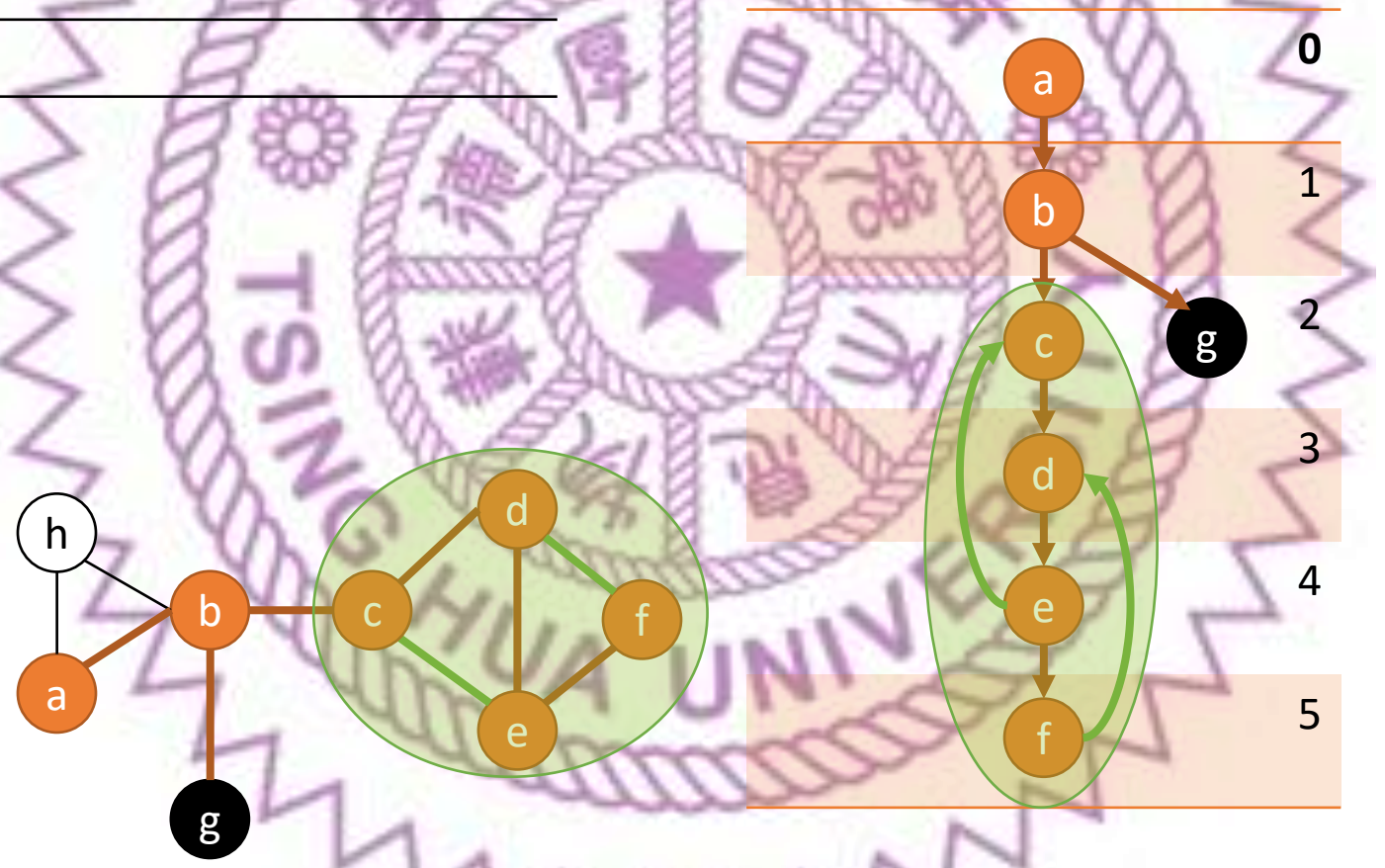
stk | a | b |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	2	x
low	0	1	2	2	2	3	2	x

stk | a | b | g |

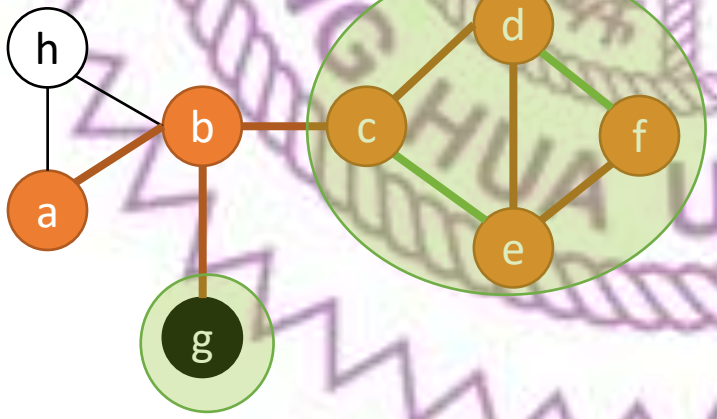


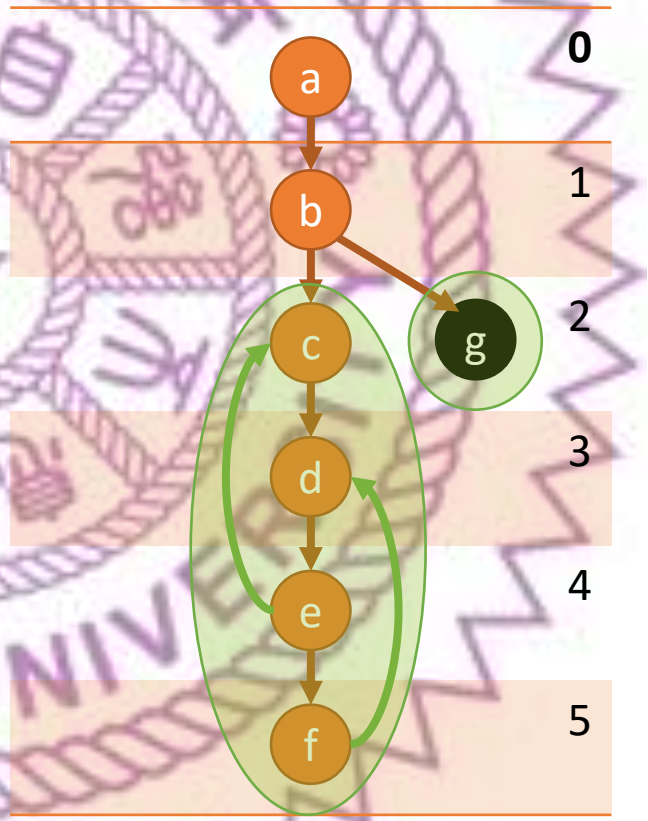
DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	2	x
low	0	1	2	2	2	3	2	x

stk | a | b | g |

low[g] = deep[g]
(b, g)是橋
形成橋連通分量

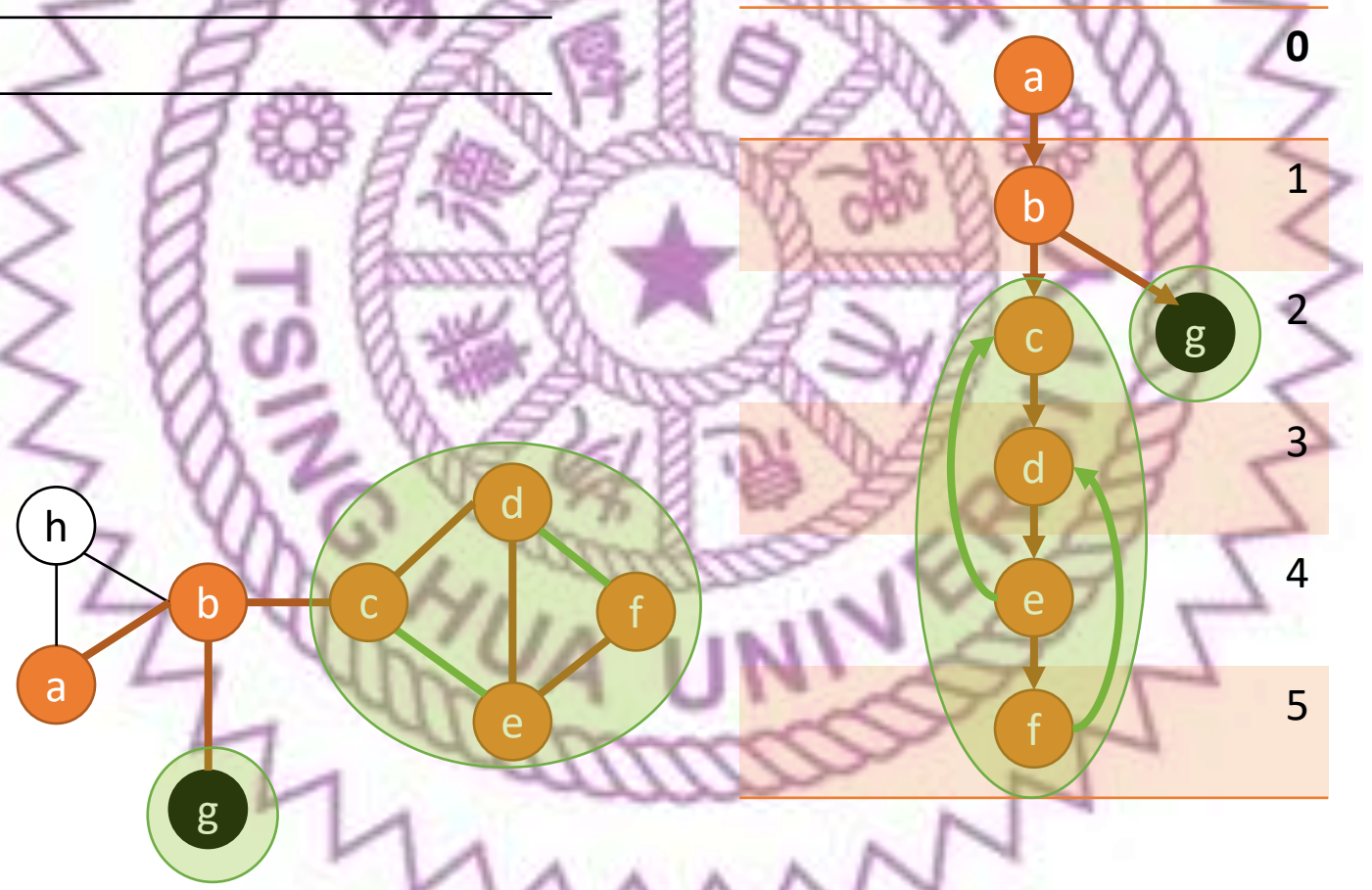




DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	2	x
low	0	1	2	2	2	3	2	x

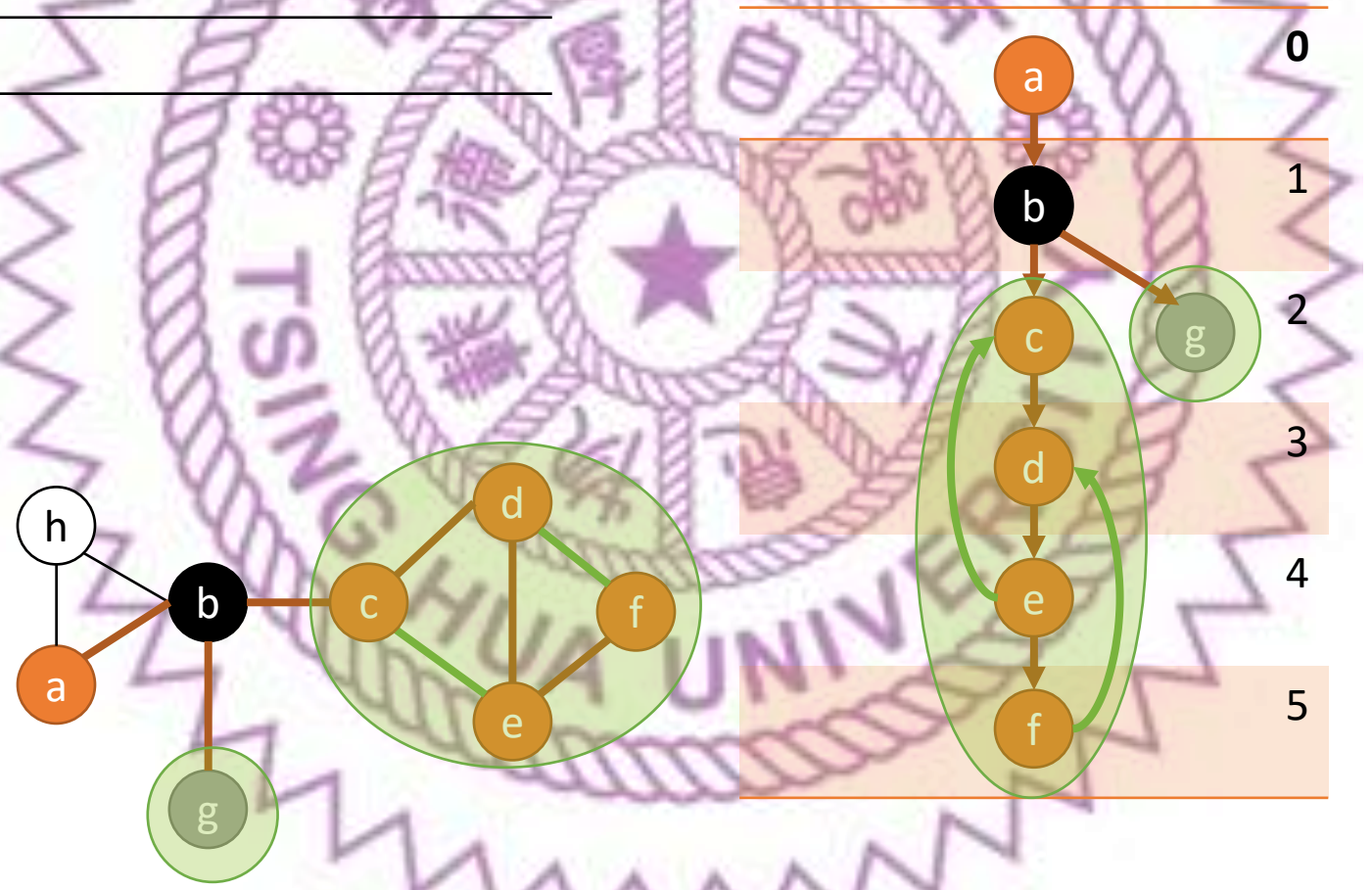
stk | a | b |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	2	x
low	0	1	2	2	2	3	2	x

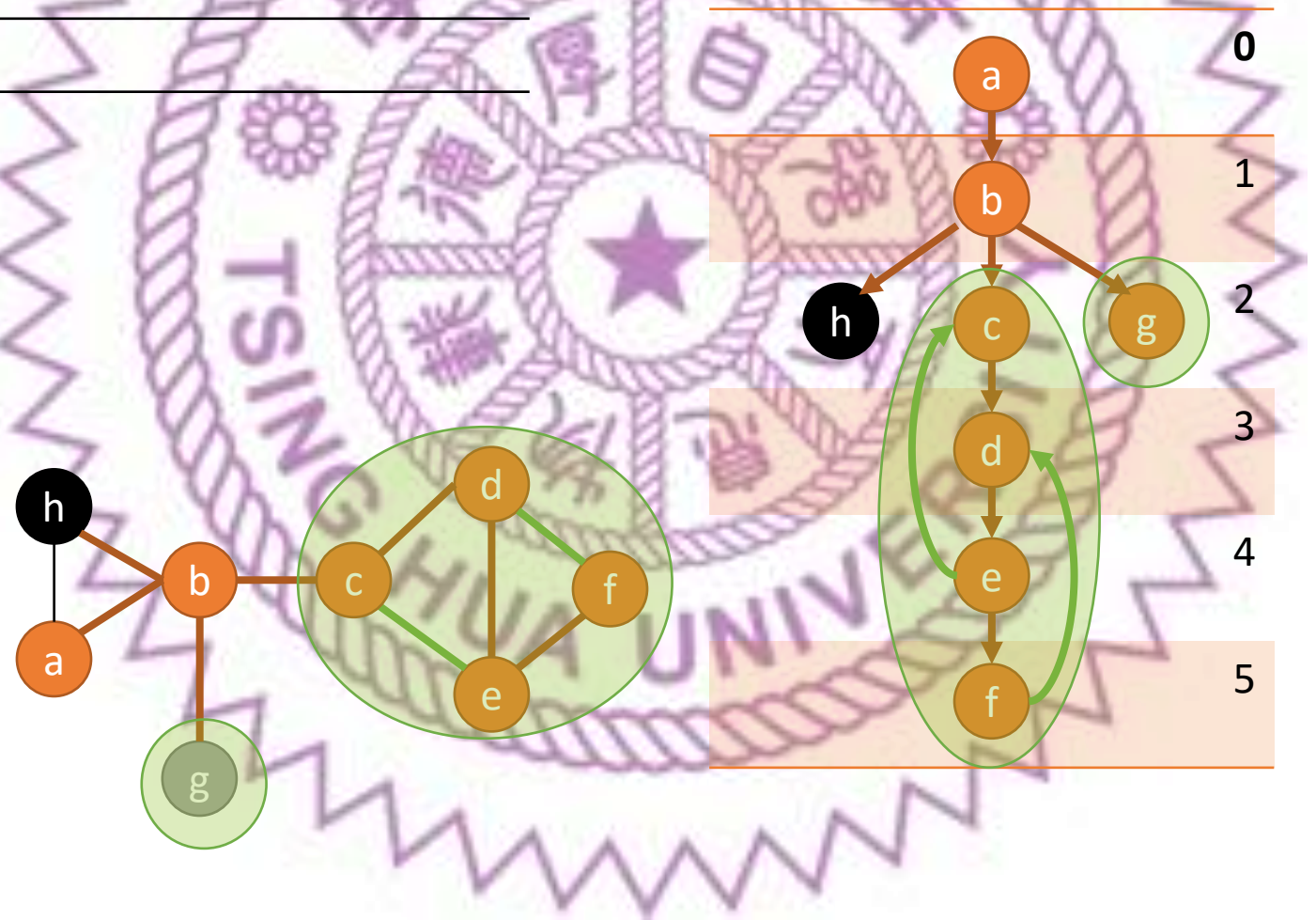
stk | a | b |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	2	2
low	0	1	2	2	2	3	2	2

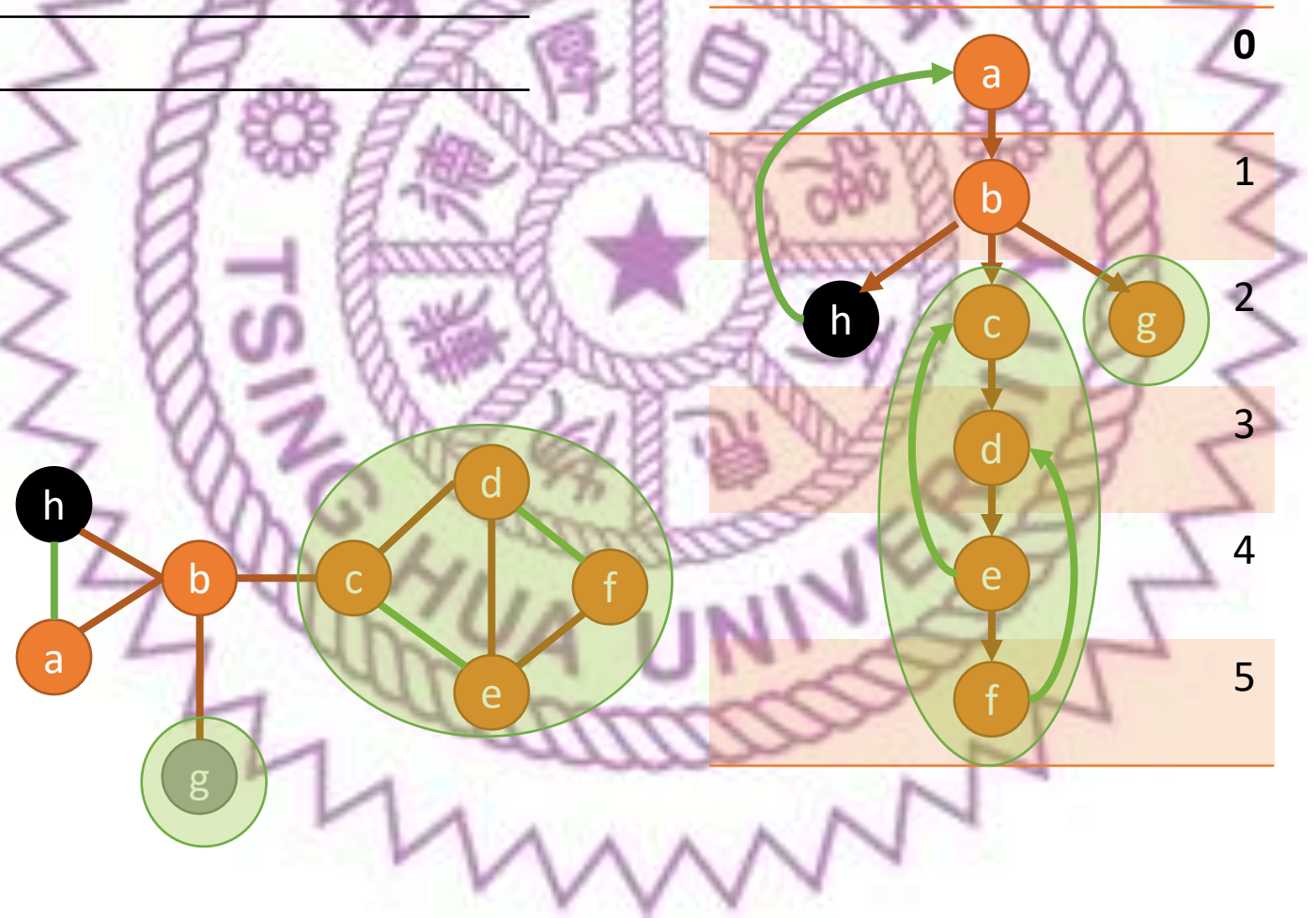
stk | a | b | h |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	2	2
low	0	1	2	2	2	3	2	0

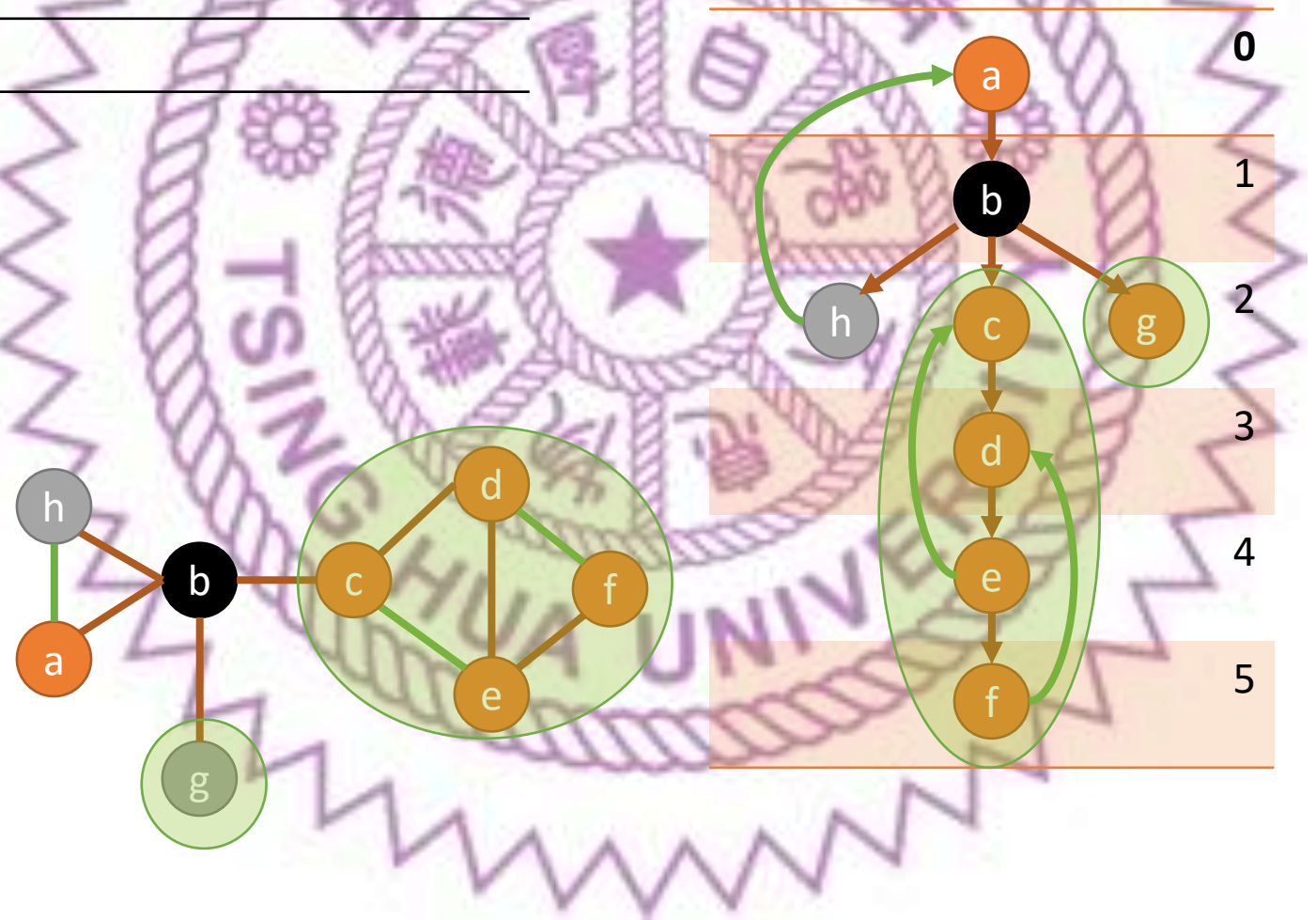
stk | a | b | h |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	2	2
low	0	1	2	2	2	3	2	0

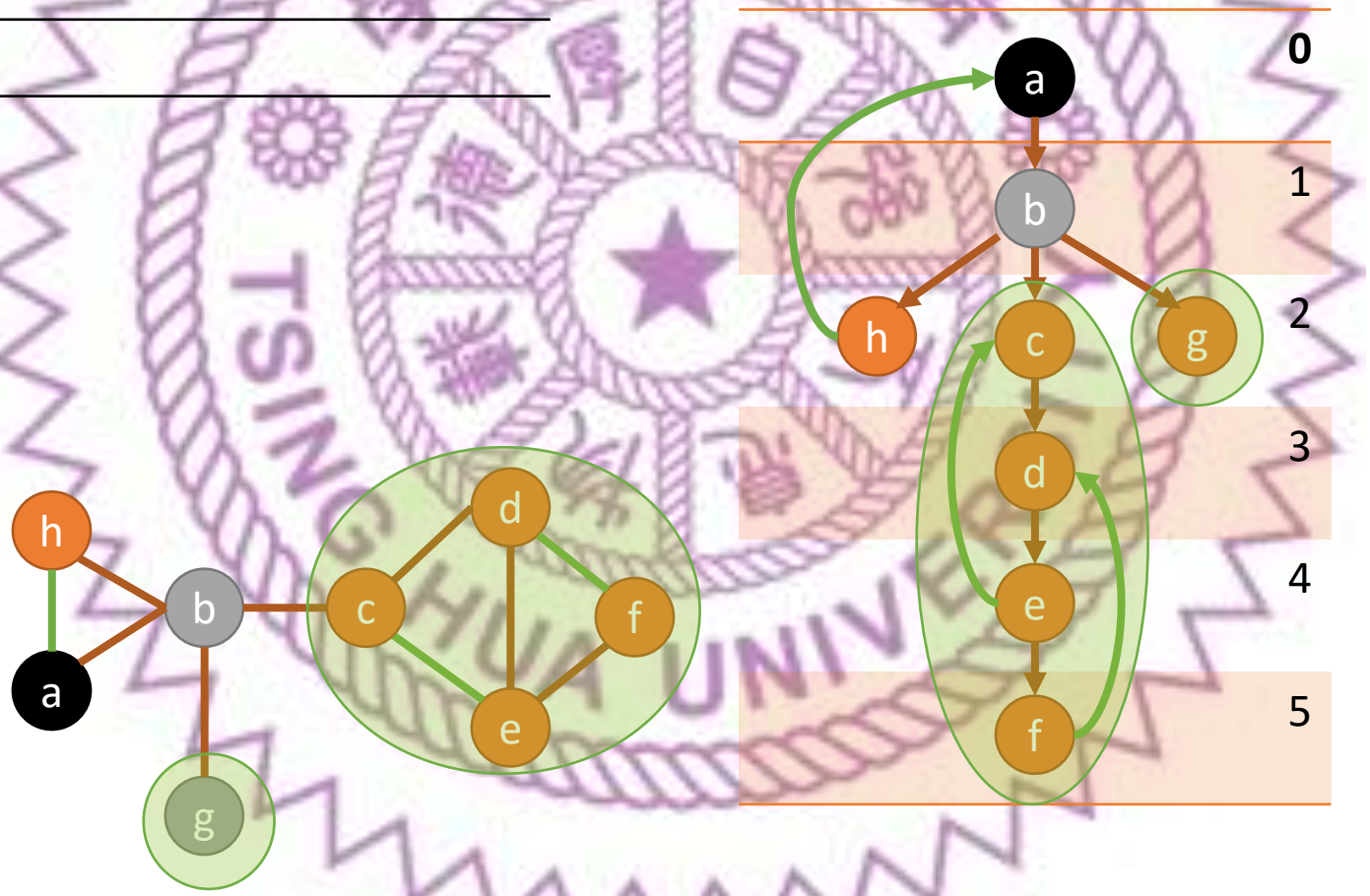
stk | a | b | h |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	2	2
low	0	1	2	2	2	3	2	0

stk | a | b | h |



DFS

	a	b	c	d	e	f	g	h
$deep$	0	1	2	3	4	5	2	2
low	0	1	2	2	2	3	2	0

stk

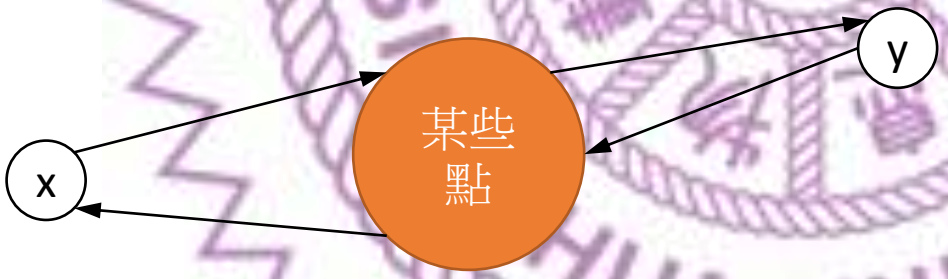
low[a] = deep[a]
形成橋連通分量

強連通分量

Strongly Connected Components, SCC

強連通分量

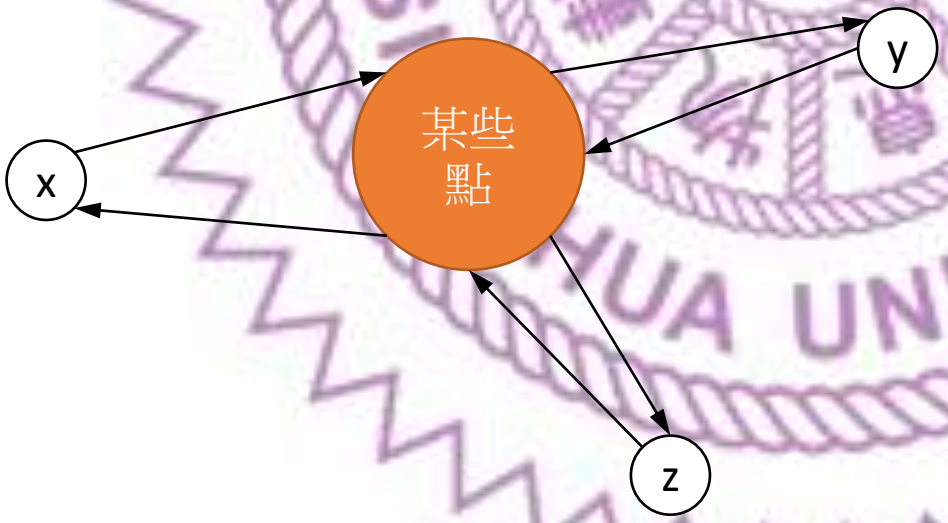
- 在一張有向圖中
  如果兩個節點 x 和 y 間同時存在路徑 $path(x, y)$ 和 $path(y, x)$
  則我們說 x 和 y 位於同一個強連通分量



強連通分量

- 如果 x, y 在同一個強連通分量
  y, z 在同一個強連通分量

- 則 x, z 也會在同一個強連通分量中

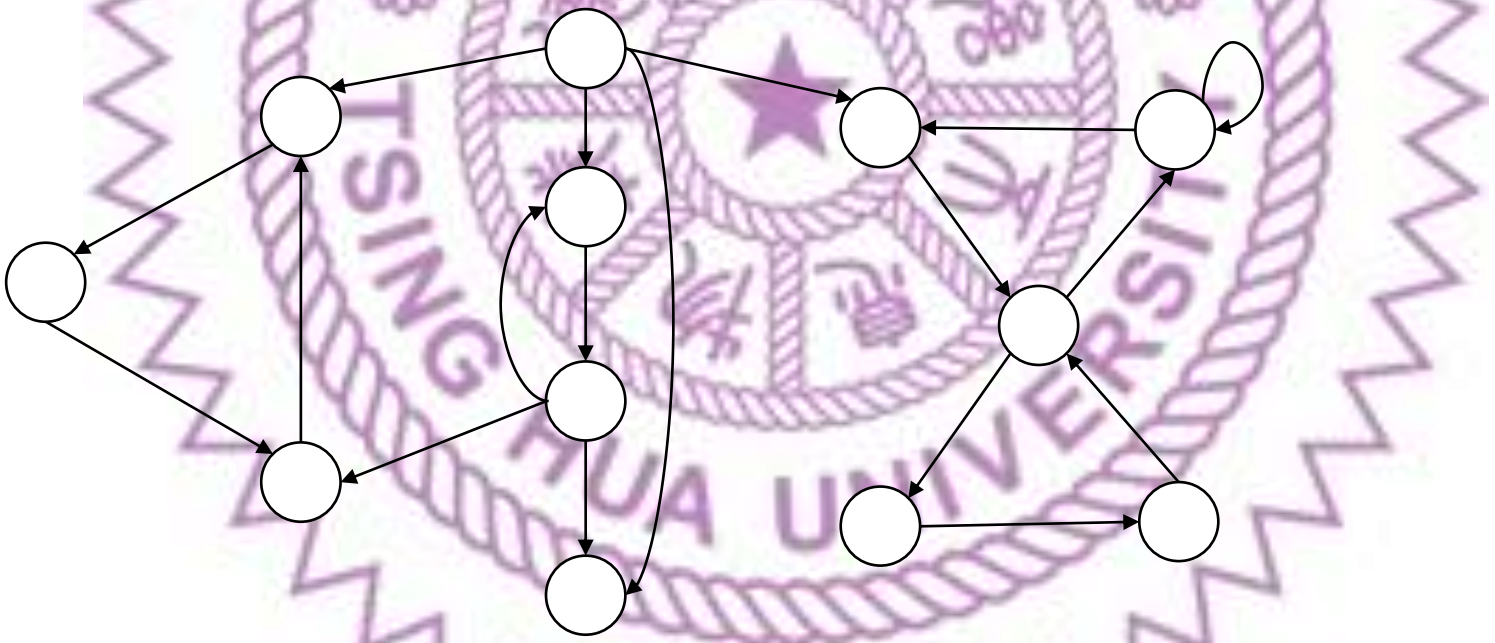


強連通分量

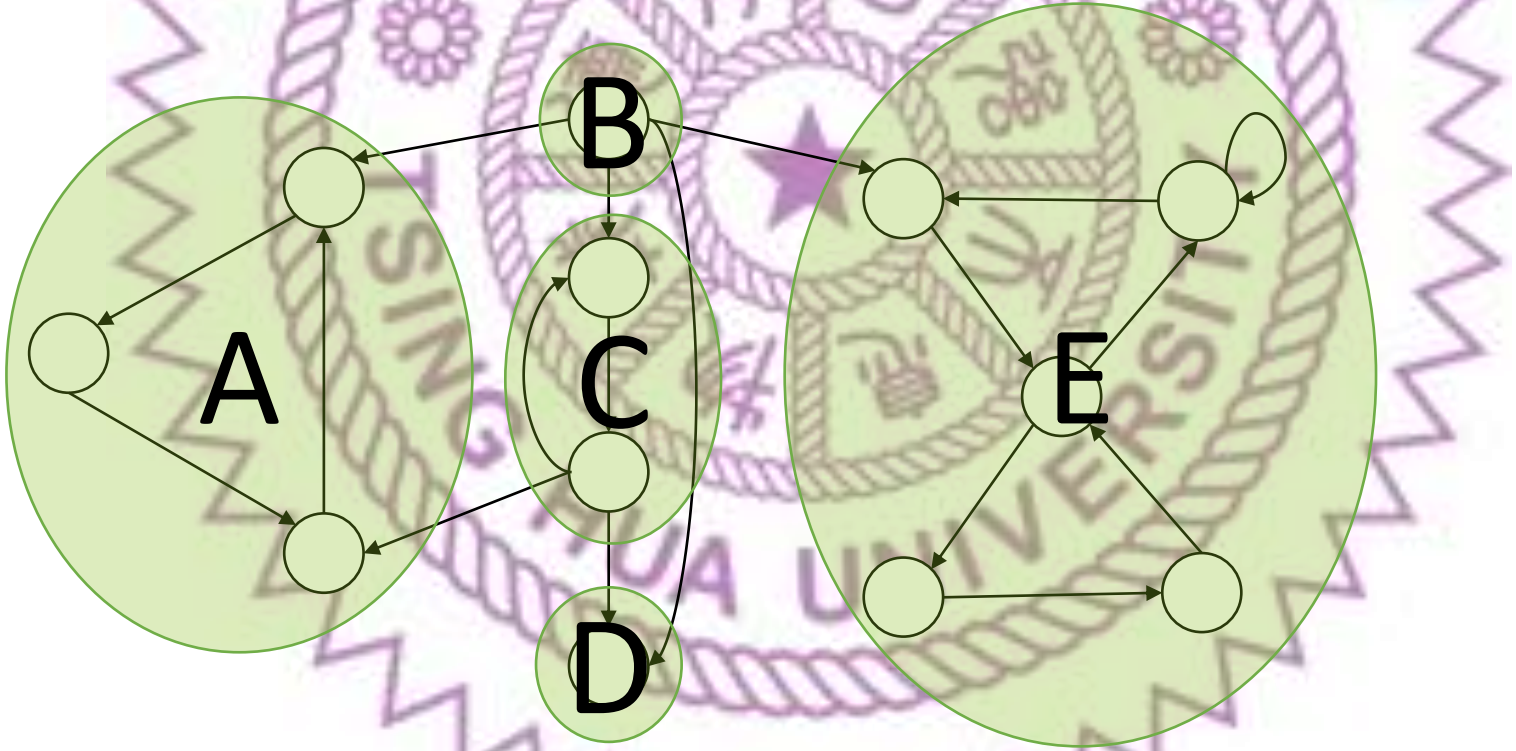
- 定義：

- 有向圖 G 的一個強連通分量
  是 G 的一個極大子圖滿足該子圖中任兩點之件都存在來回的路徑

練習：找出所有強連通分量

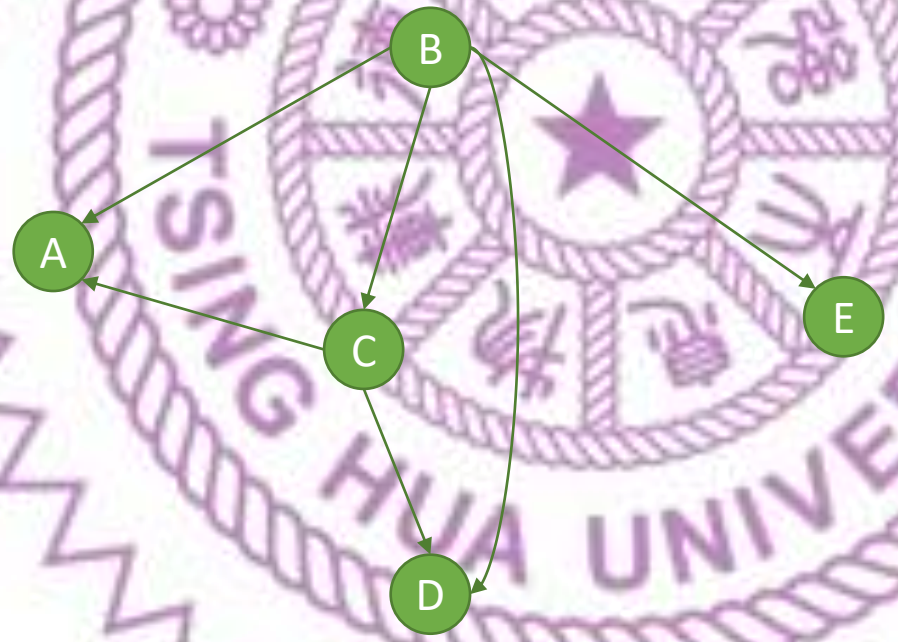


解答



縮點

- 將每個強連通分量收縮成一個點
- 縮點之後的圖會是有向無環圖 (DAG)



兩個演算法

- 通常計算SCC常用的演算法有以下兩種
- Tarjan’s SCC
- Kosaraju (很簡單自己學習)

Tarjan’s SCC

- 類似於之前無項圖連通分量的作法，維護一個 low 陣列
- DFS 過程中維護一個 stack 就能找出所有的SCC
- 但這個 low 陣列紀錄的東西是根據當前深度以及 stack 中的節點決定的

Pseudocode c++ style


```
Number cnt := 0;
void DFS(Vertex u) {
  cnt := cnt + 1;
  deep[u] := cnt, low[u] := cnt;
  stk.push(u), inStk[u] = true;
  for (Vertex v : neighbors[u]) {
    if (deep[v] == 0) {
      DFS(v);
      low[u] := min(low[u], low[v]);
    } else if (inStk[v]) {
      low[u] := min(low[u], deep[v]);
    }
  }
  if (d == low[u]) {
    S := ∅;
    Vertex x;
    do {
      x := stk.top();
      S := S ∪ x;
      stk.pop(), inStk[x] = false;
    } while (x ≠ u);
    SCC := SCC ∪ S;
  }
}
```

比較

```
void DFS(Vertex u, Vertex pa, Number d) {
  visit[u] := true;
  deep[u] := d;
  low[u] := d;
  stk.push(u);
  for (Vertex v : neighbors[u]) {
    if (not visit[v]) {
      DFS(v, u, d + 1);
      low[u] := min(low[u], low[v]);
    } else if (deep[v] < deep[u] and v ≠ pa) {
      low[u] := min(low[u], deep[v]);
    }
  }
  if (d == low[u]) {
    B := ∅;
    Vertex x;
    do {
      x := stk.top();
      B := B ∪ x;
      stk.pop();
    } while (x ≠ u);
    BCC := BCC ∪ B;
  }
}
```

(Bridge) BCC

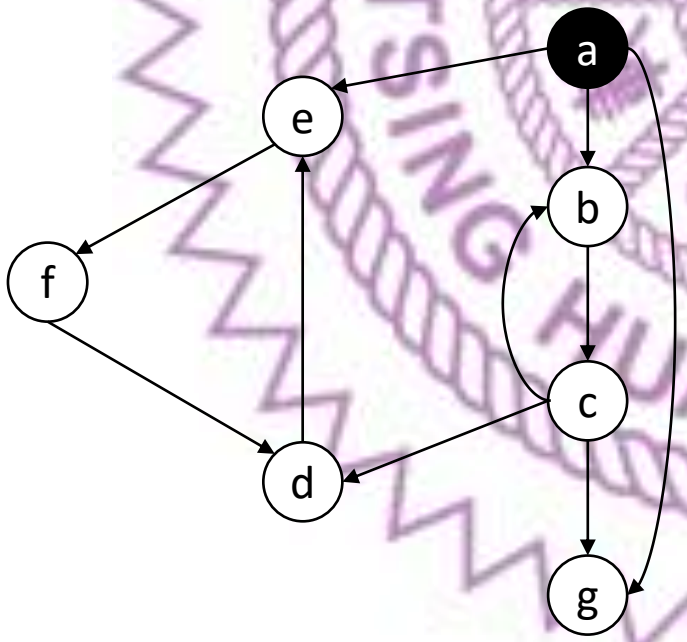
```
Number cnt := 0;
void DFS(Vertex u) {
  cnt := cnt + 1;
  deep[u] := cnt, low[u] := cnt;
  stk.push(u), inStk[u] = true;
  for (Vertex v : neighbors[u]) {
    if (deep[v] == 0) {
      DFS(v);
      low[u] := min(low[u], low[v]);
    } else if (inStk[v]) {
      low[u] := min(low[u], deep[v]);
    }
  }
  if (d == low[u]) {
    S := ∅;
    Vertex x;
    do {
      x := stk.top();
      S := S ∪ x;
      stk.pop(), inStk[x] = false;
    } while (x ≠ u);
    SCC := SCC ∪ S;
  }
}
```

DFS

cnt = 1

	a	b	c	d	e	f	g
$deep$	1	x	x	x	x	x	x
low	1	x	x	x	x	x	x

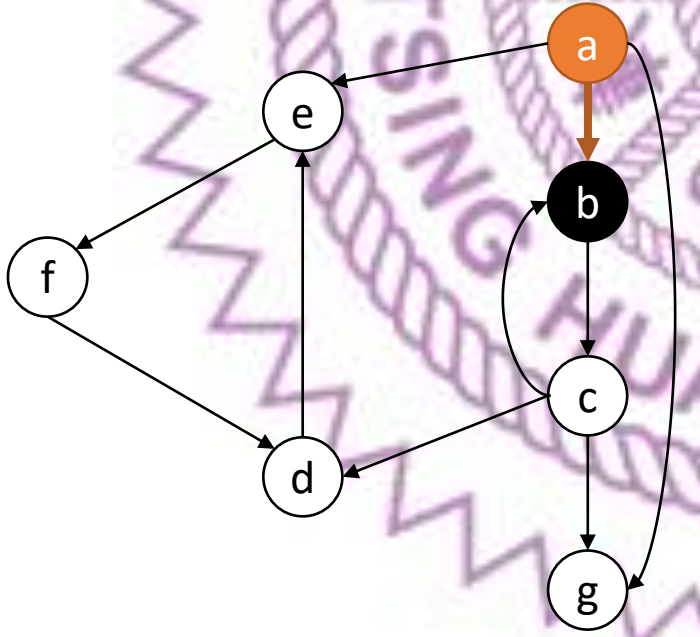
stk | a

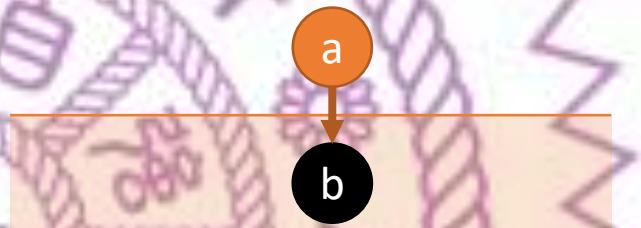


DFS

	a	b	c	d	e	f	g
$deep$	1	2	x	x	x	x	x
low	1	2	x	x	x	x	x

stk | a | b |

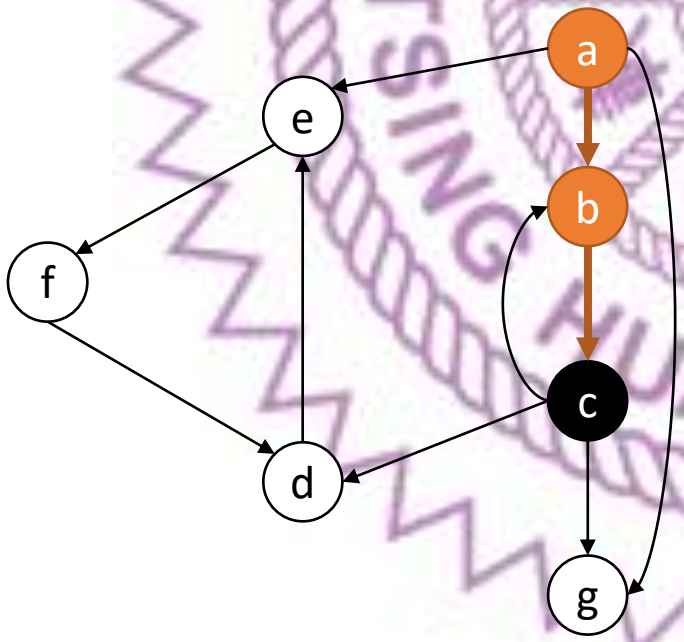


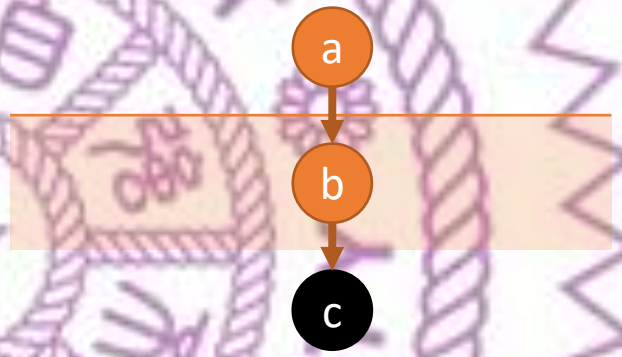


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	x	x	x	x
low	1	2	3	x	x	x	x

stk | a | b | c |

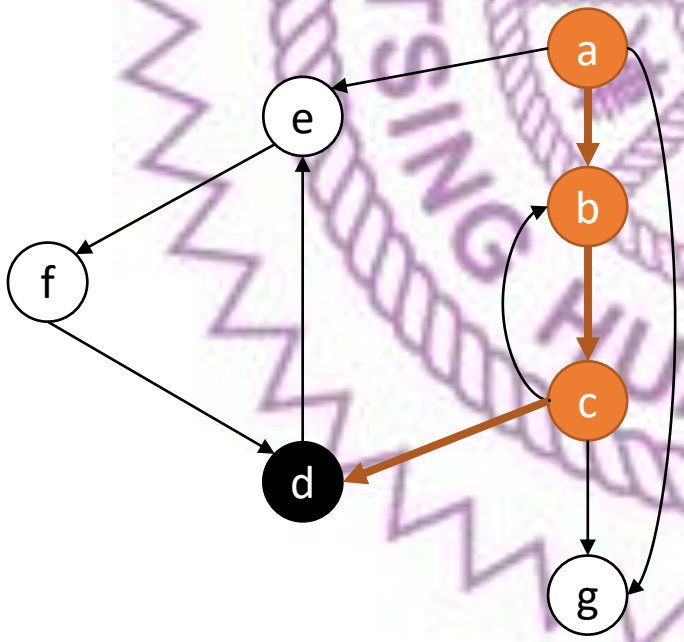


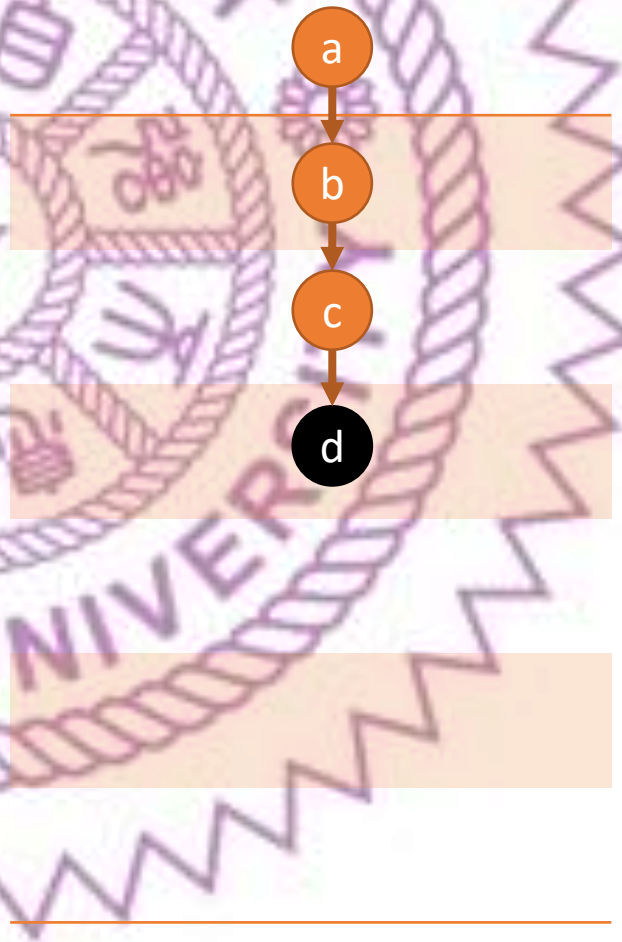


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	x	x	x
low	1	2	3	4	x	x	x

stk | a | b | c | d |

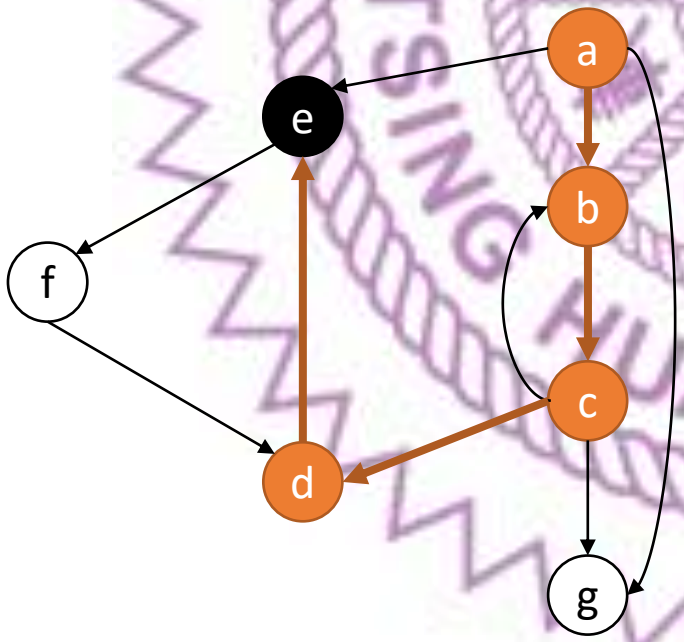


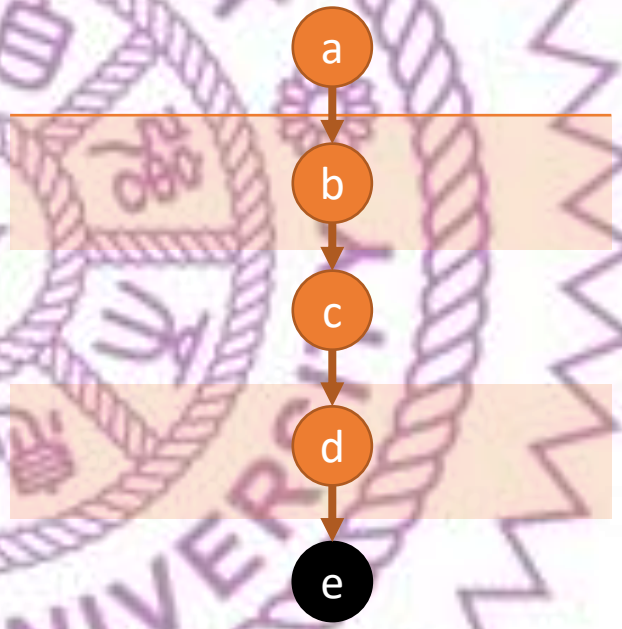


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	x	x
low	1	2	3	4	5	x	x

stk | a | b | c | d | e |

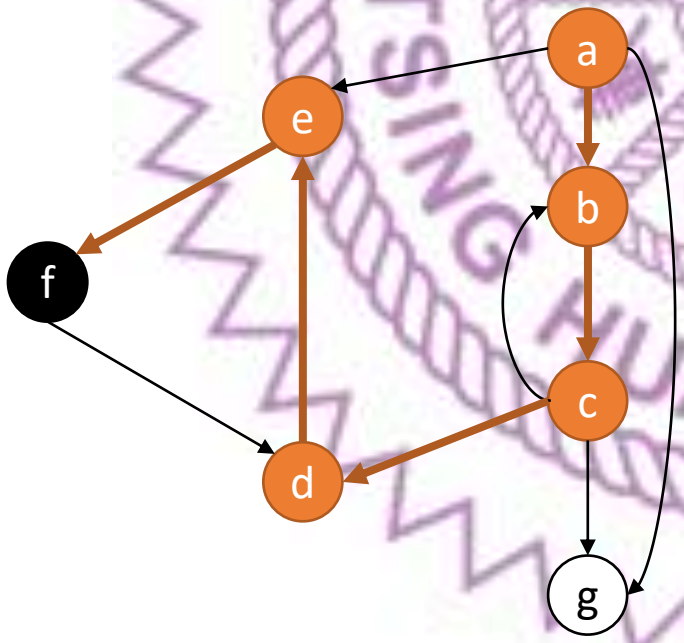


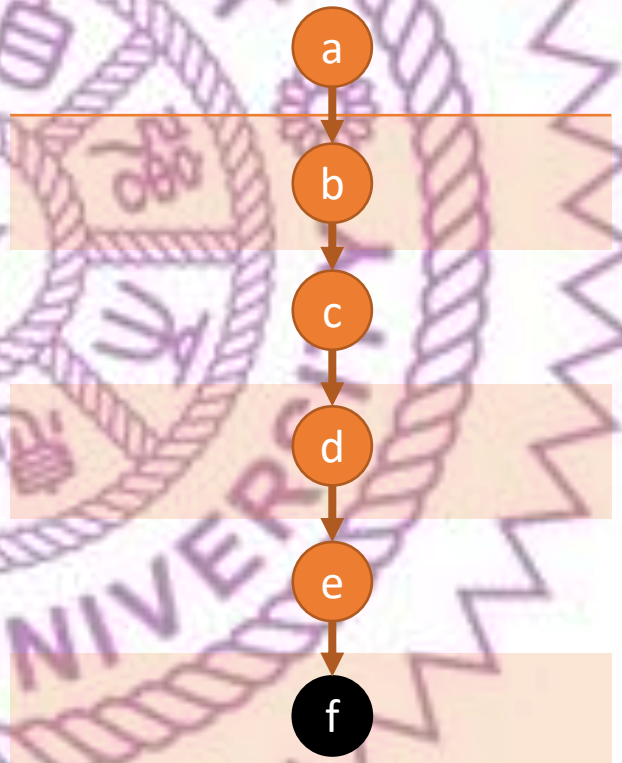


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	x
low	1	2	3	4	5	6	x

stk | a | b | c | d | e | f |

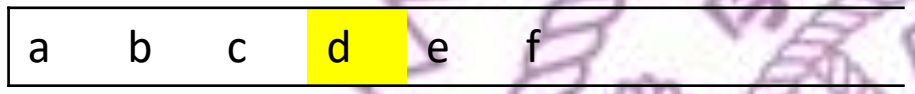


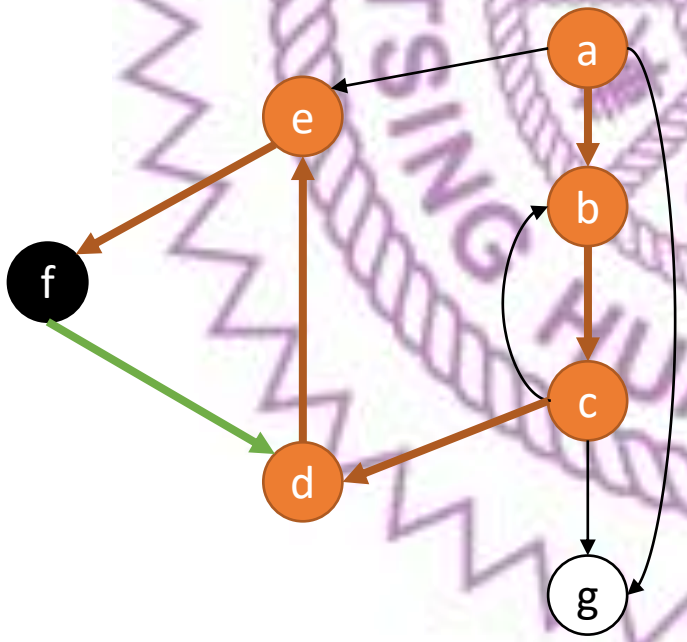


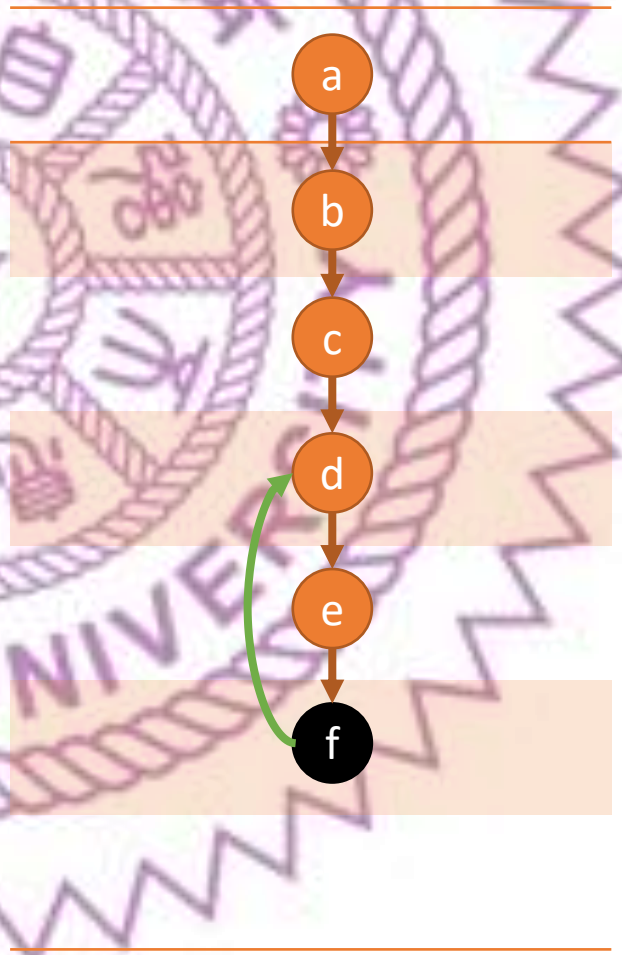
DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	x
low	1	2	3	4	5	4	x

stk



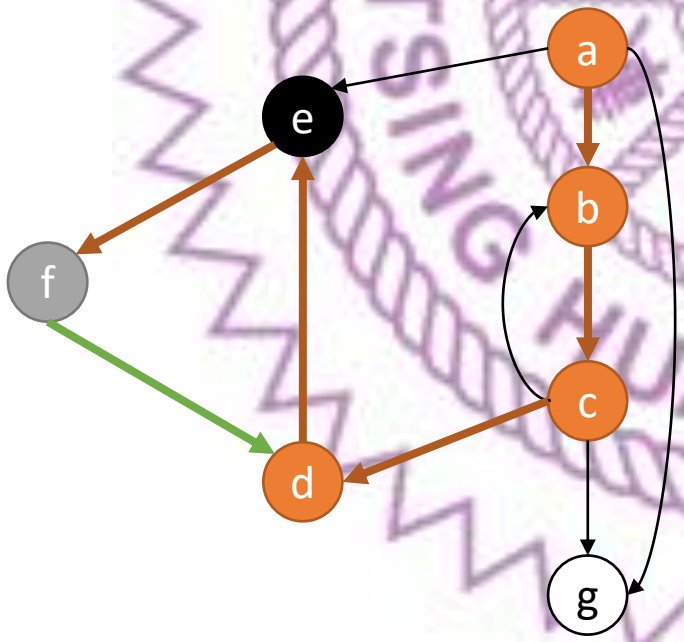


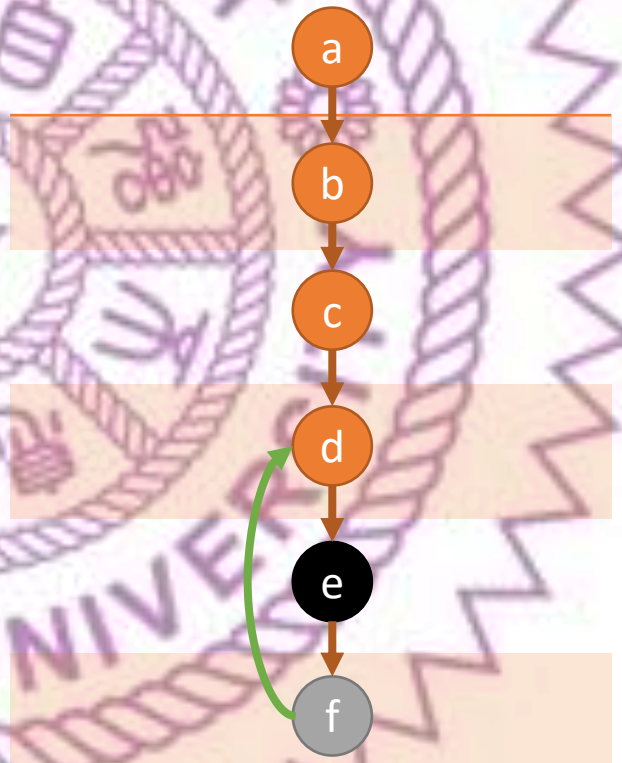


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	x
low	1	2	3	4	4	4	x

stk | a | b | c | d | e | f |



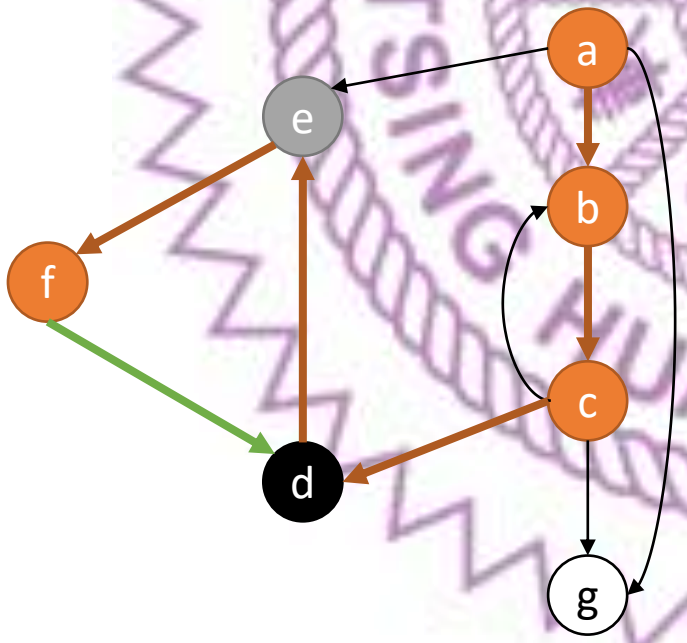


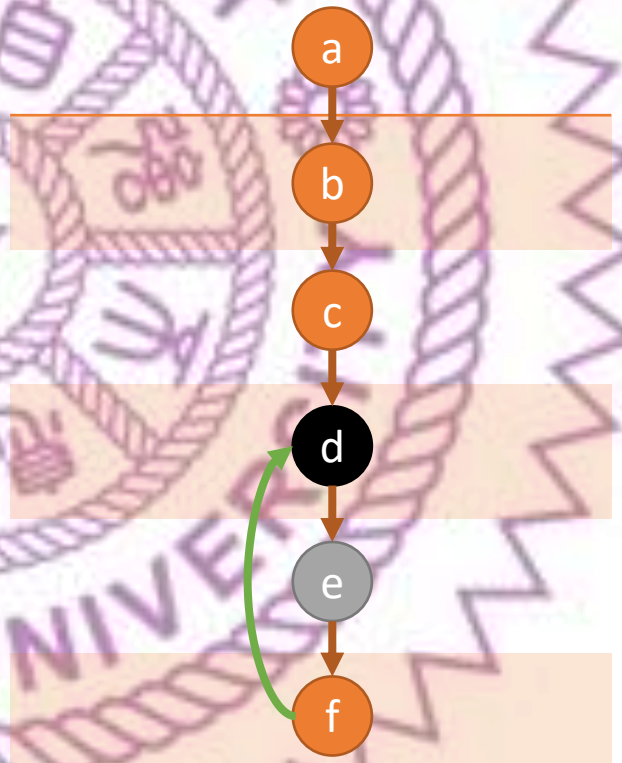
DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	x
low	1	2	3	4	4	4	x

stk | a | b | c | d | e | f |

Low[d]=deep[d]
形成SCC

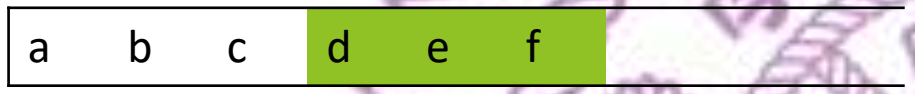




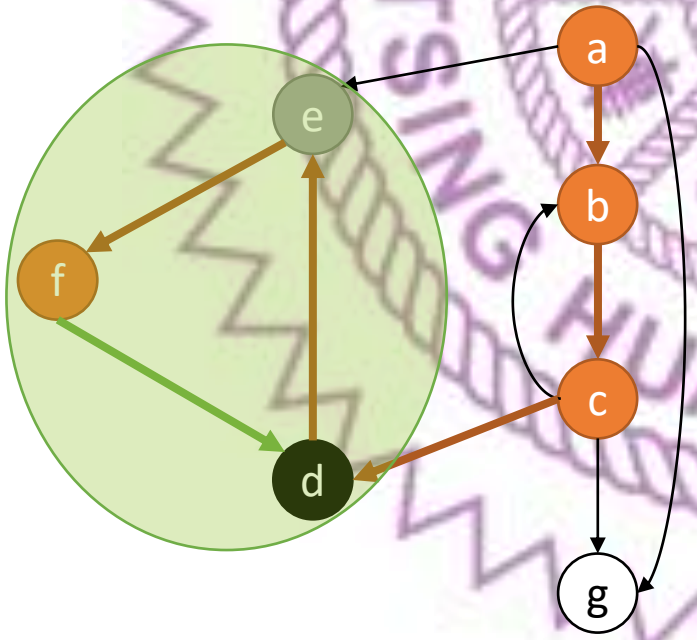
DFS

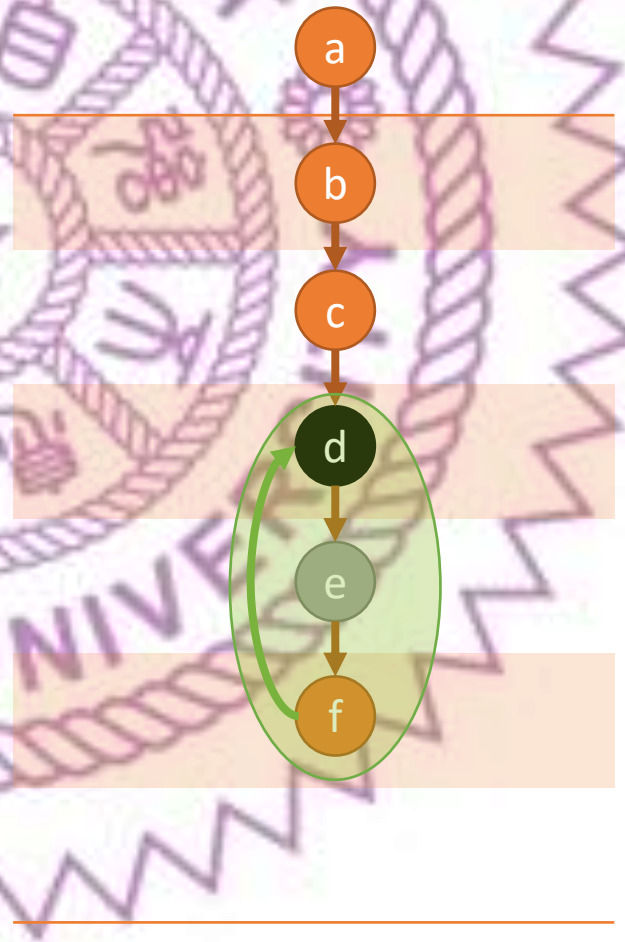
	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	x
low	1	2	3	4	4	4	x

stk



Low[d]=deep[d]
形成SCC



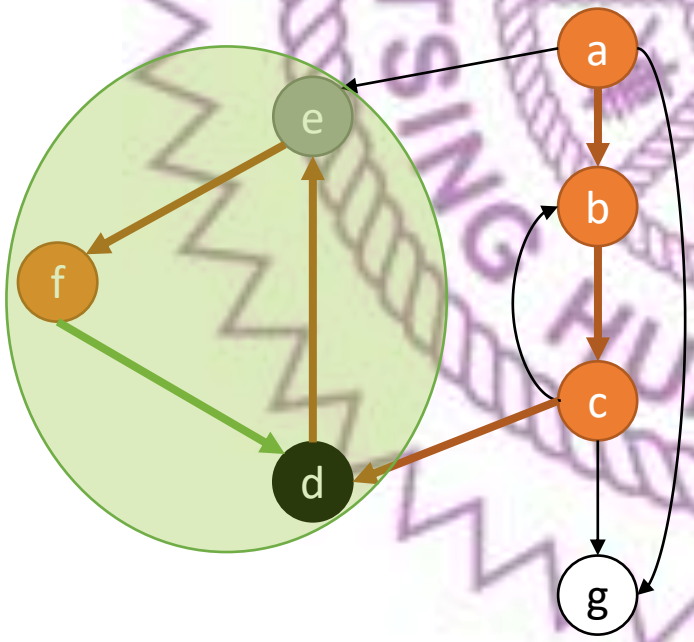


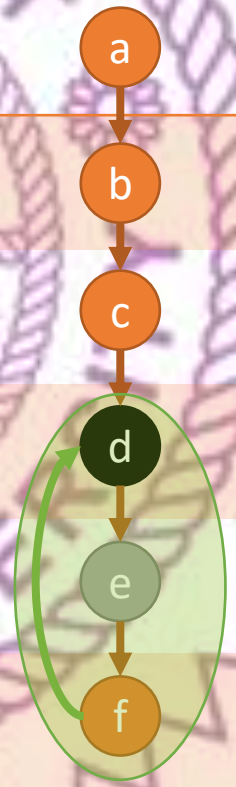
DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	x
low	1	2	3	4	4	4	x

stk | a b c |

Low[d]=deep[d]
形成SCC

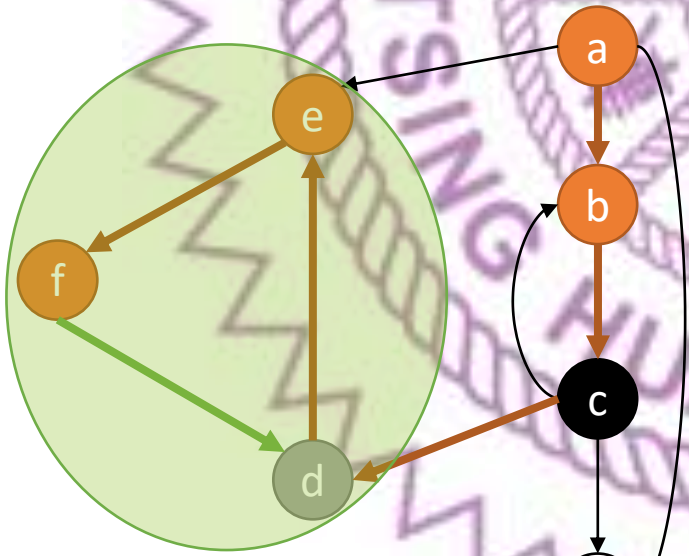


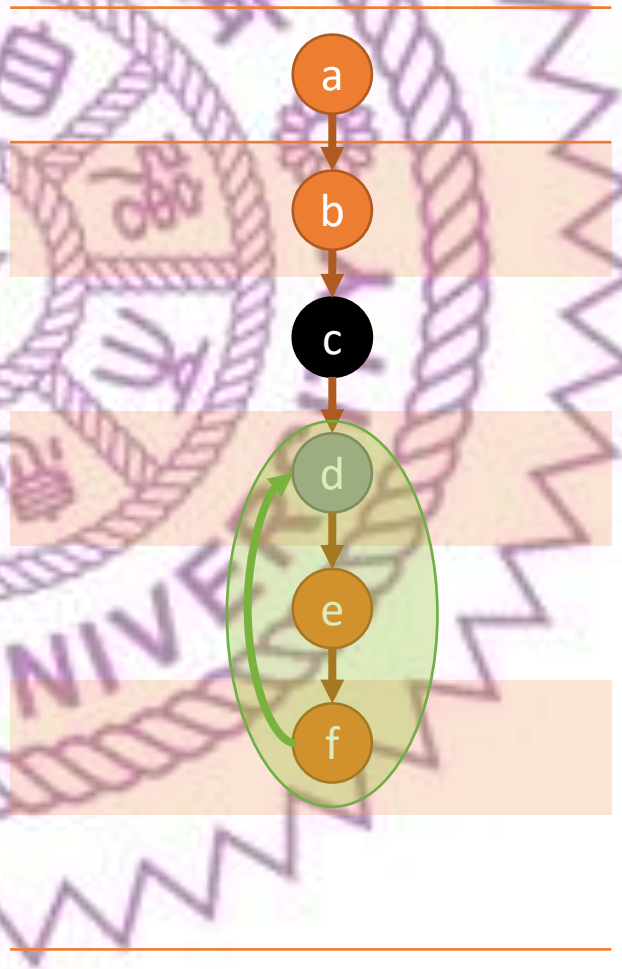


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	x
low	1	2	3	4	4	4	x

stk | a | b | c |

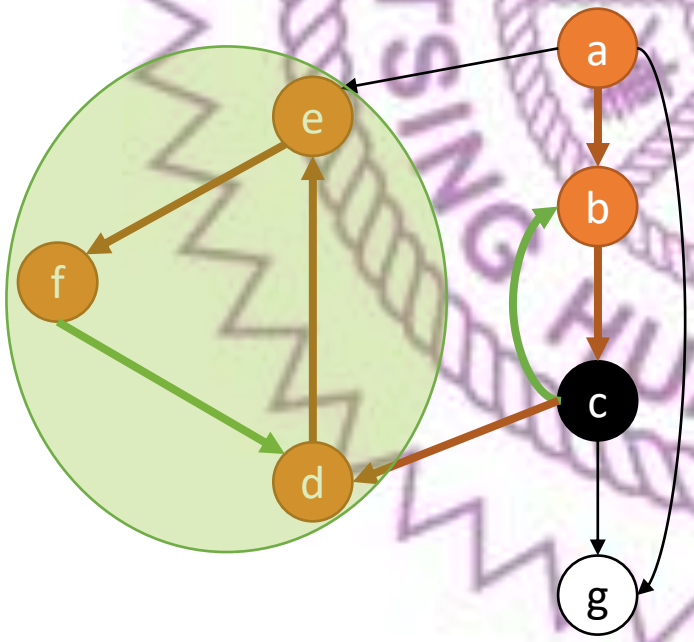


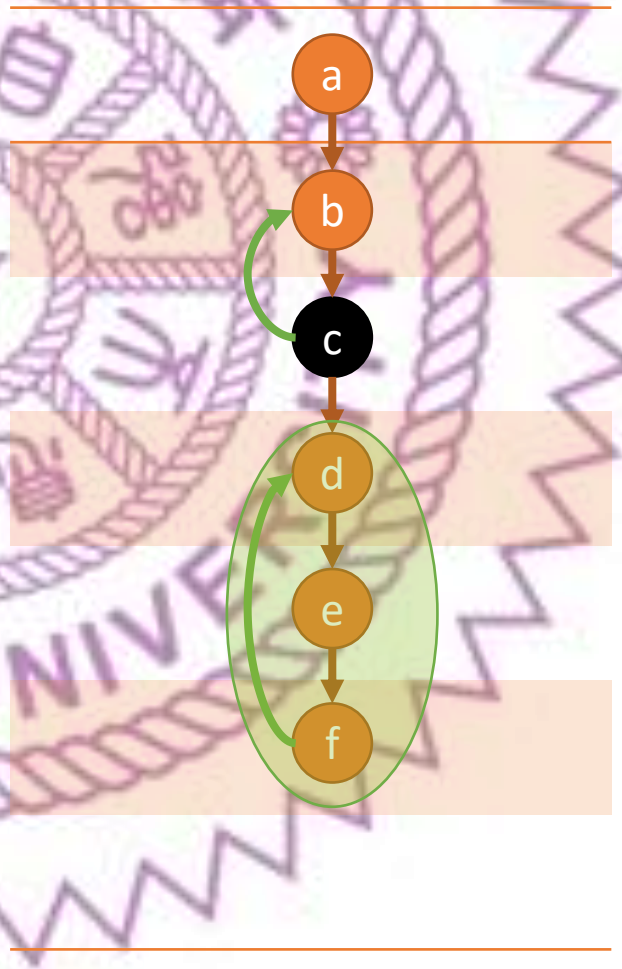


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	x
low	1	2	2	4	4	4	x

stk

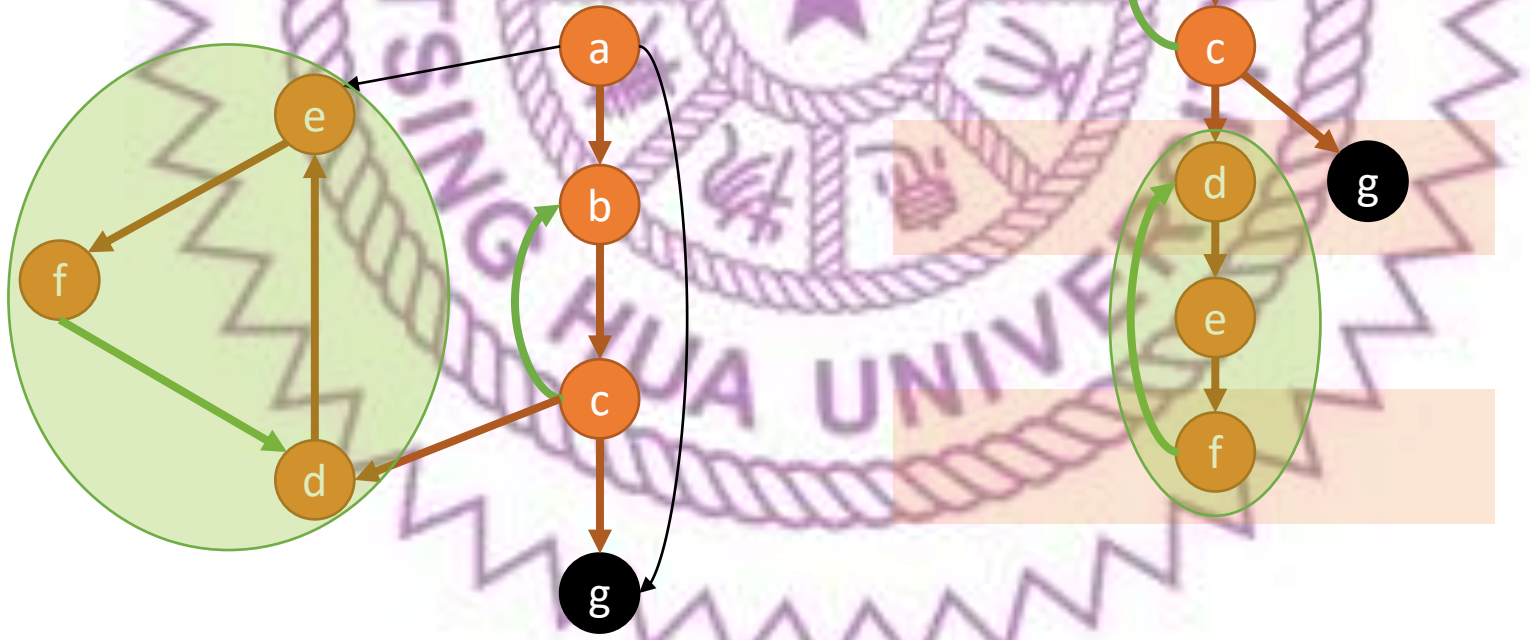




DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

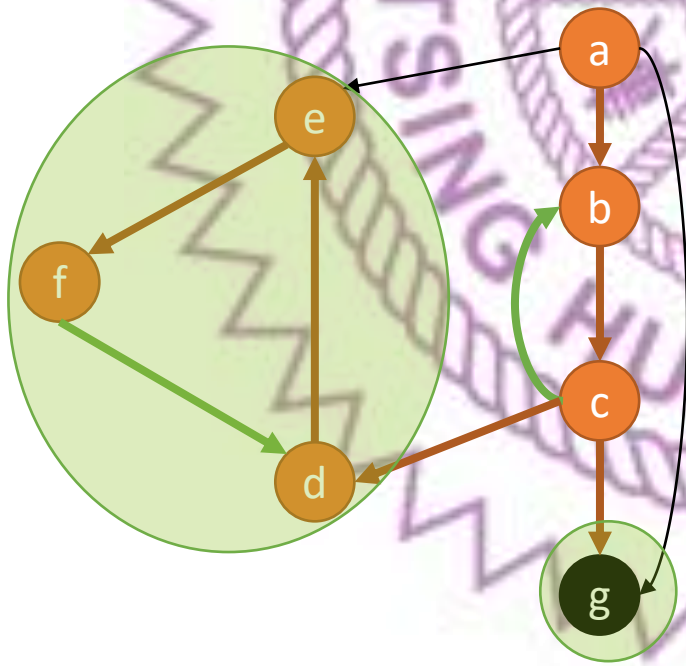
stk | a | b | c | g |

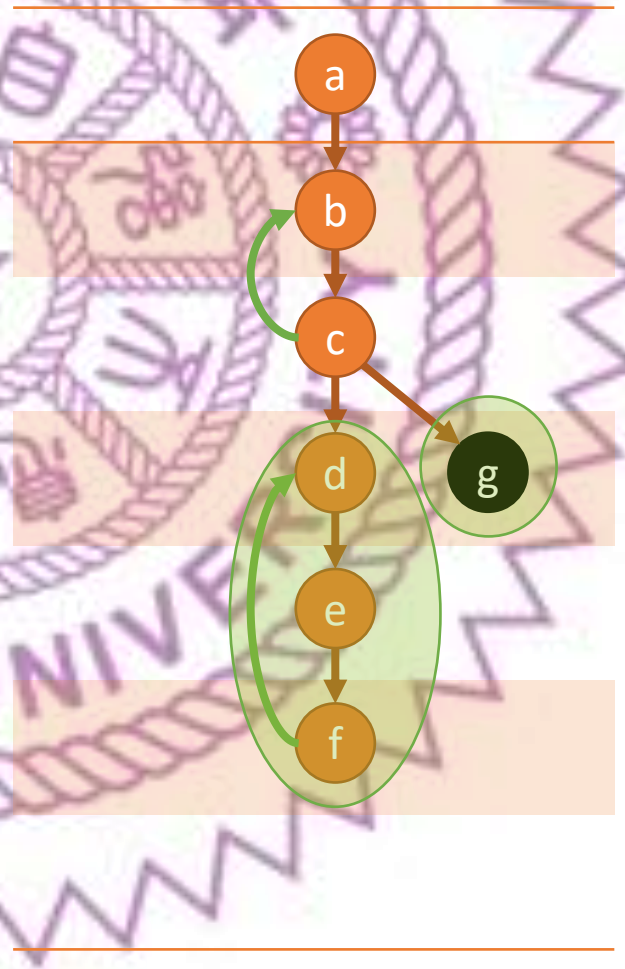


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

stk

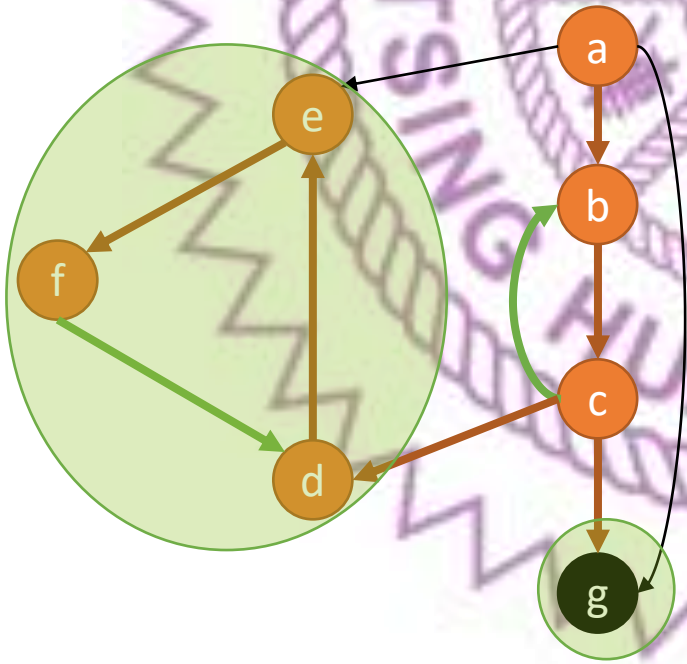


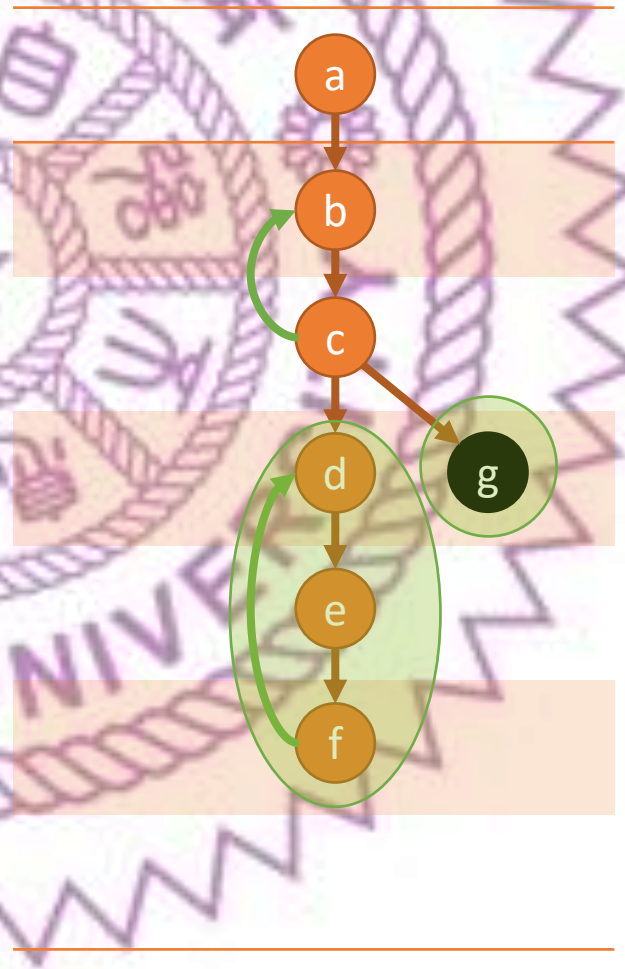


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

stk | a | b | c |

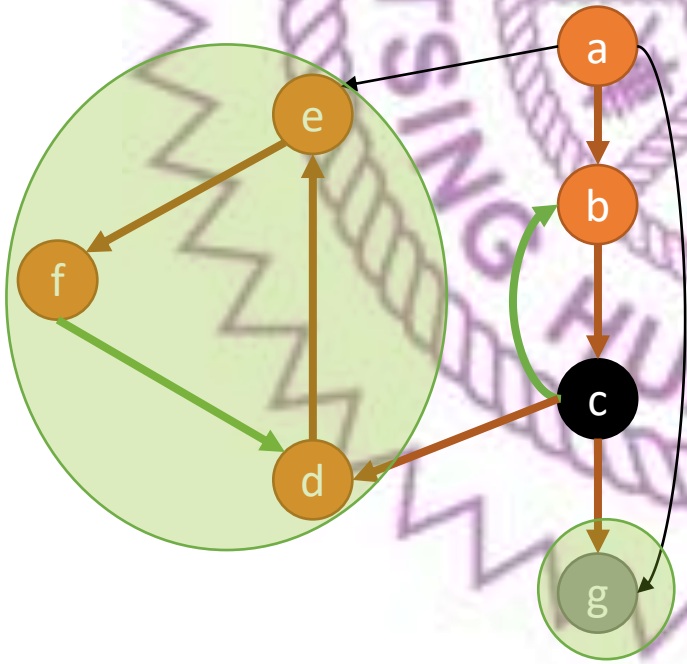


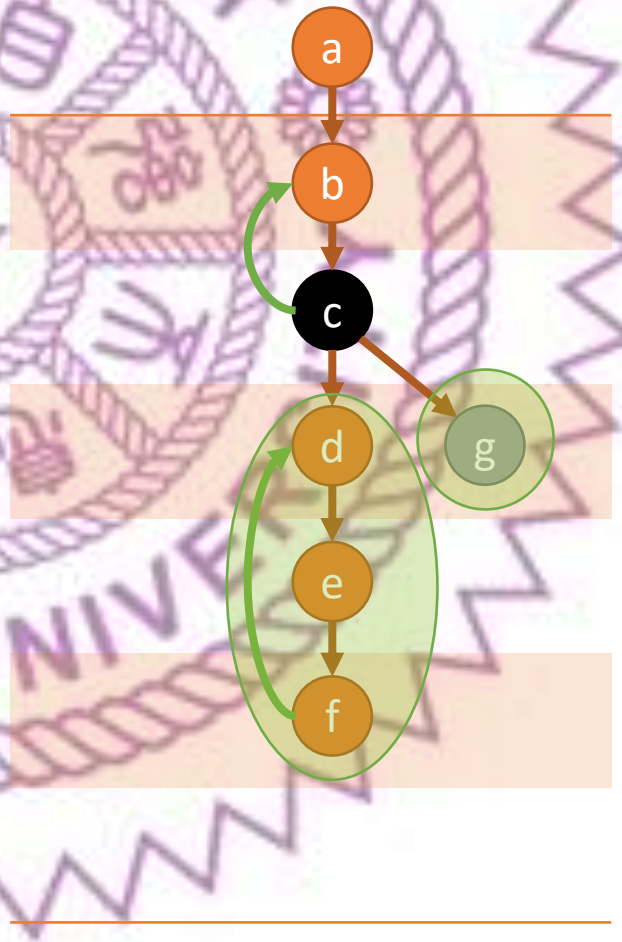


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

stk | a | b | c |

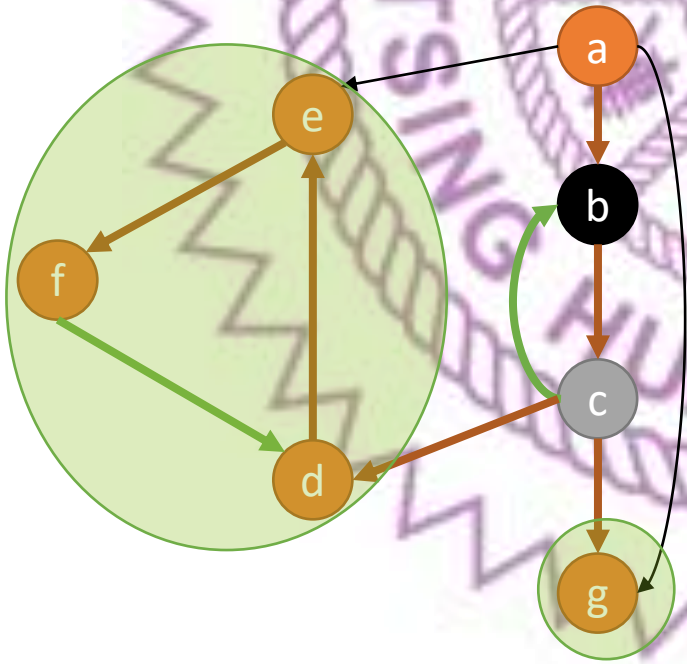


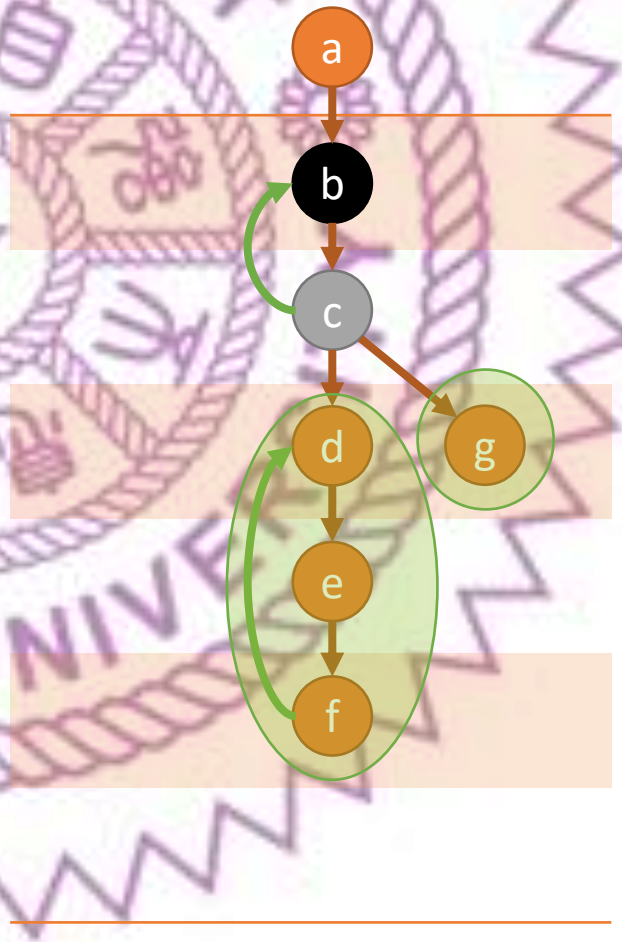


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

stk | a | b | c |

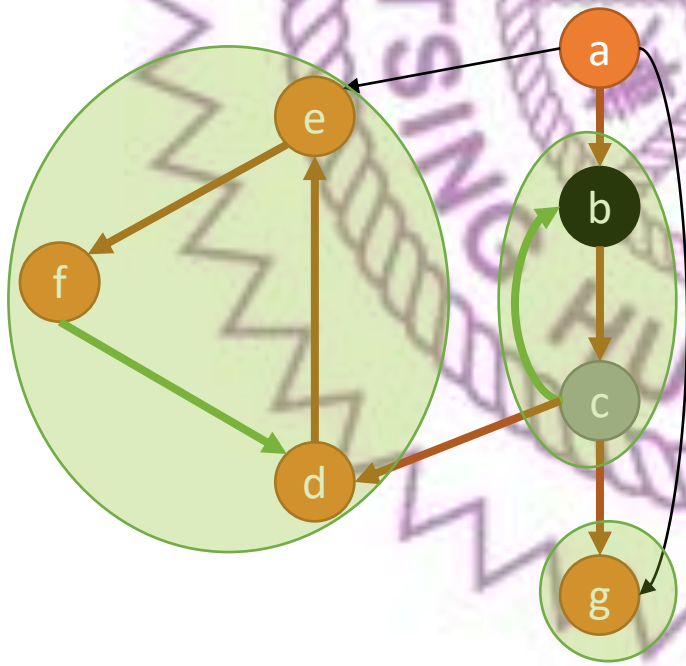


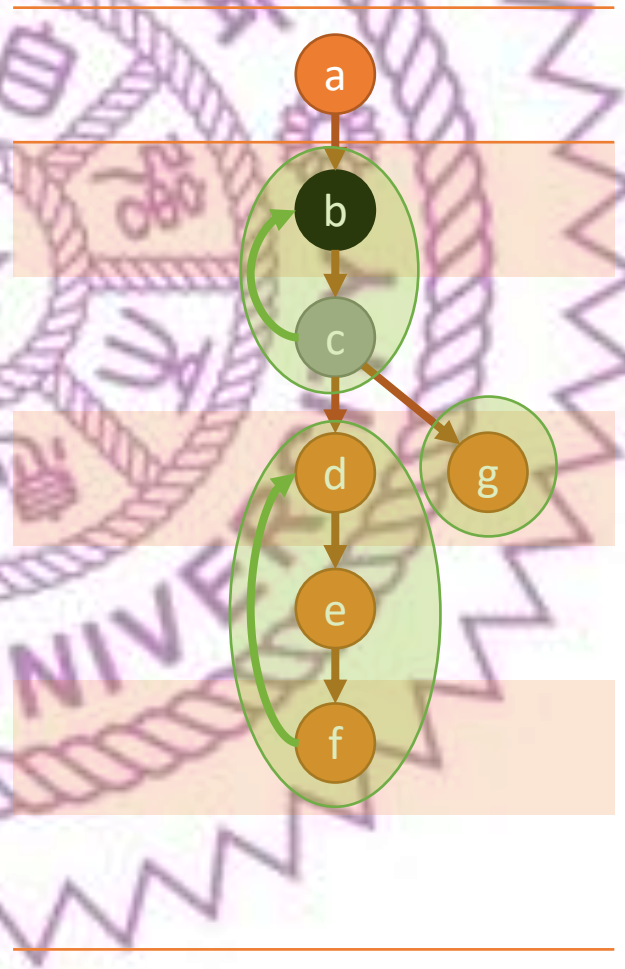


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

stk

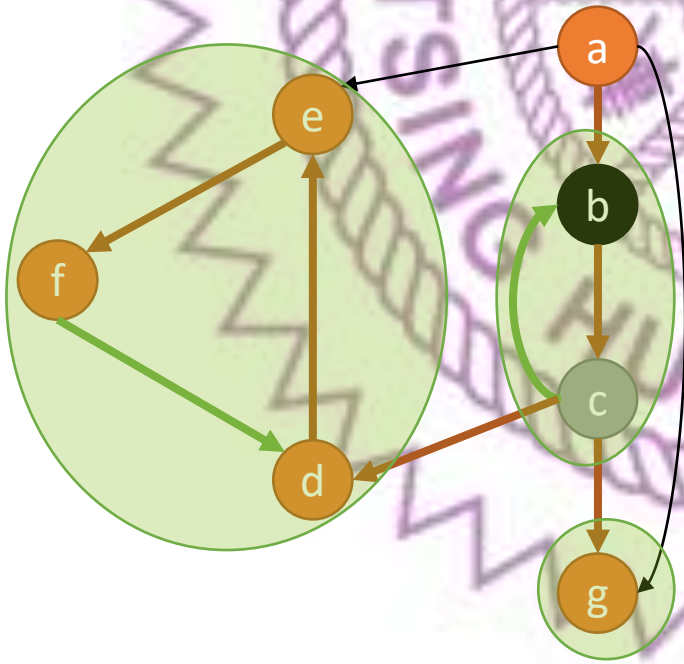


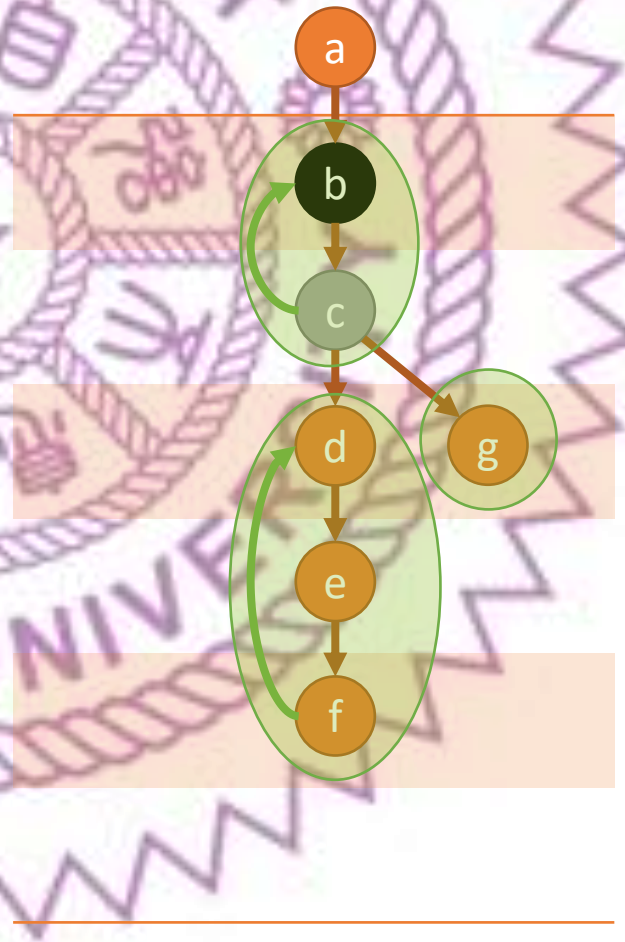


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

stk | a |

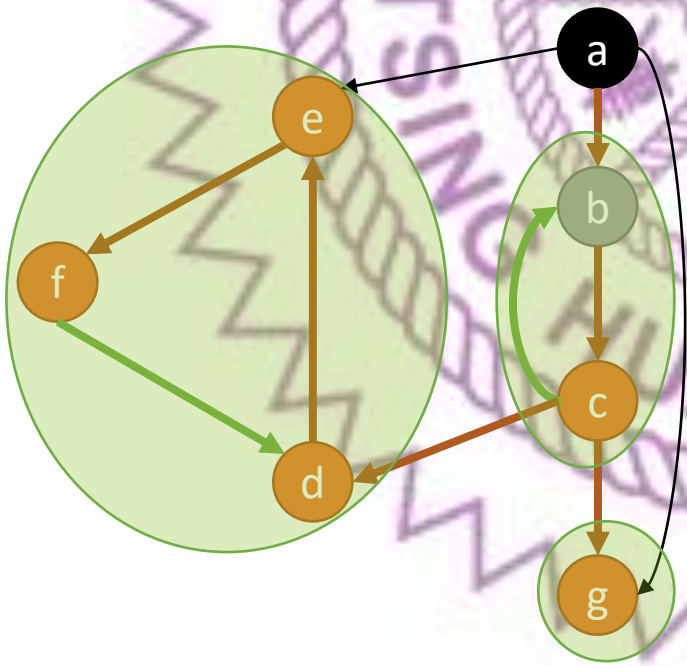


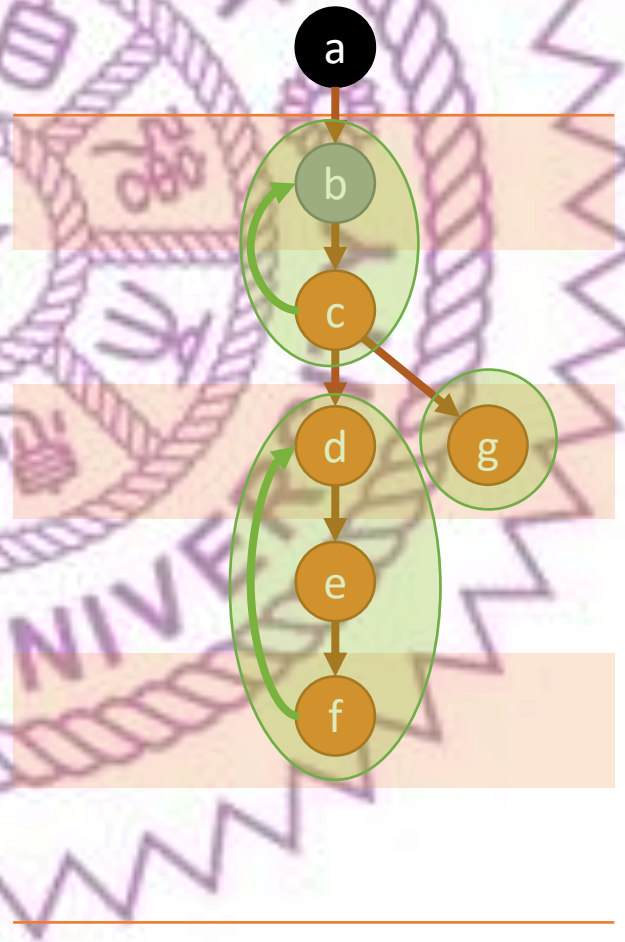


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

stk | a |

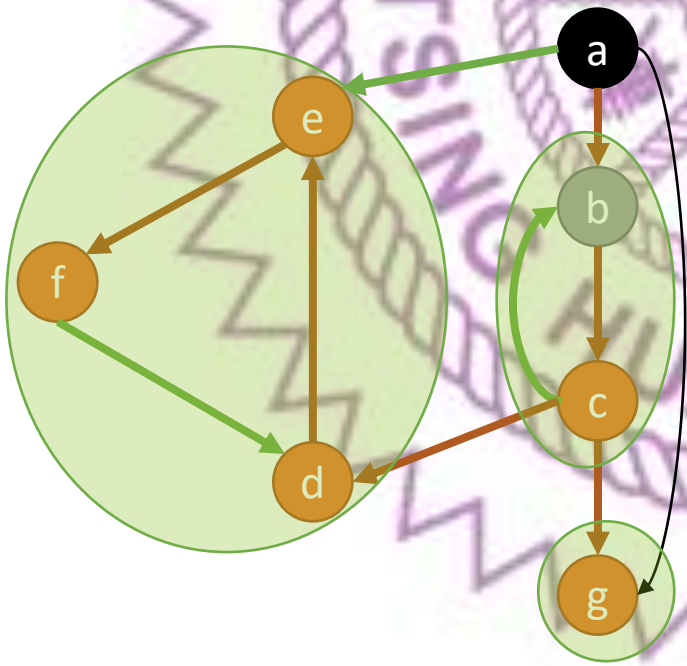


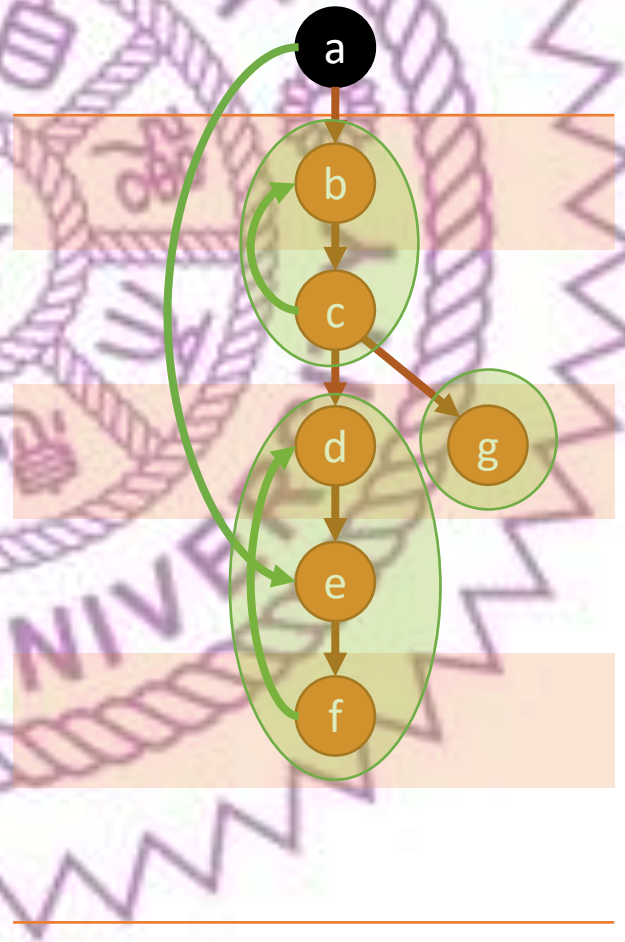


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

stk | a |

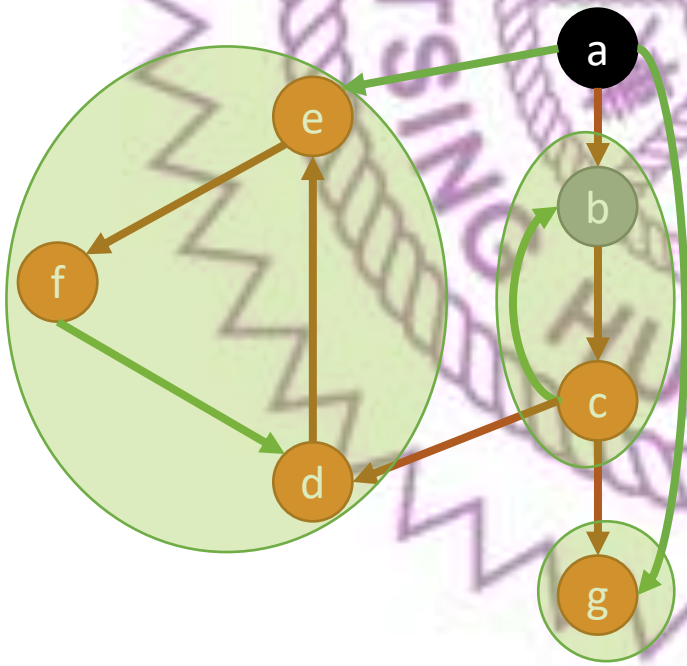


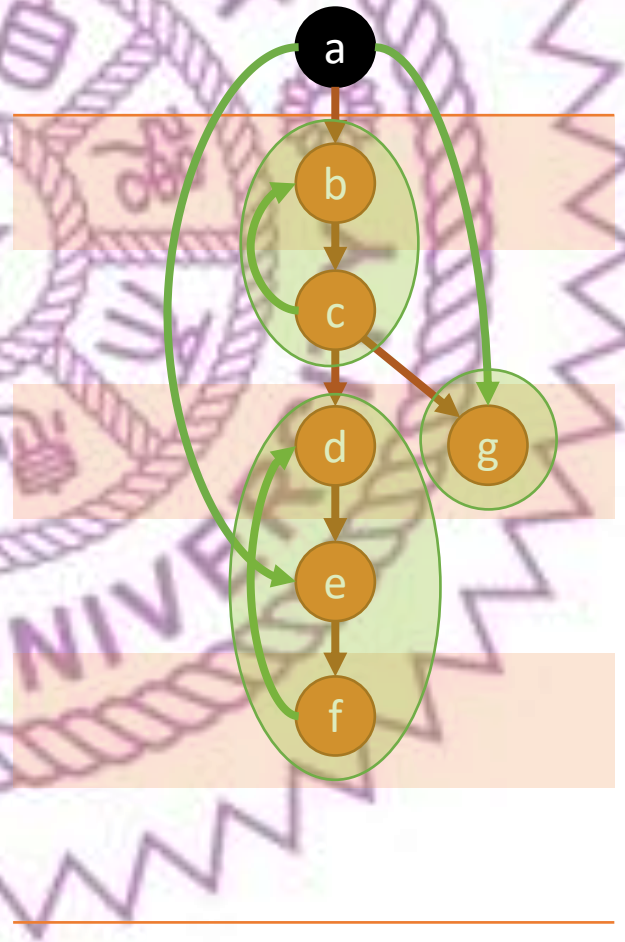


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

stk | a |

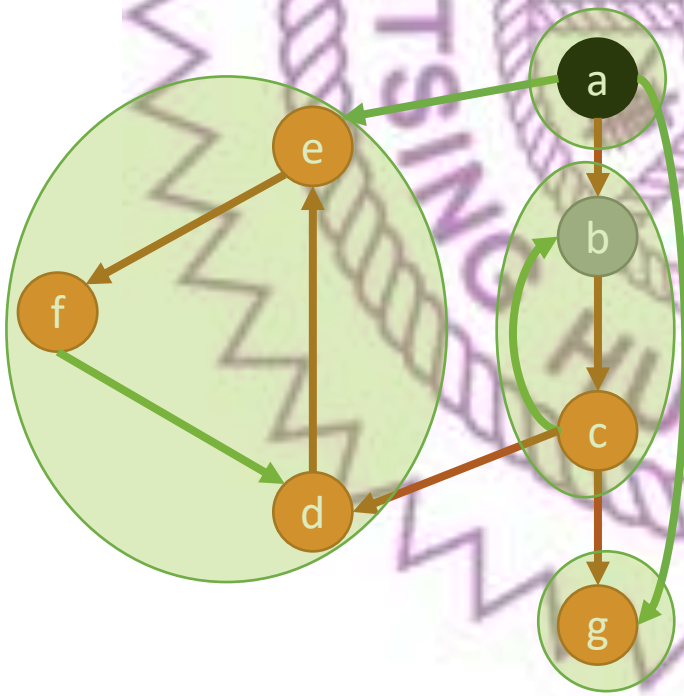


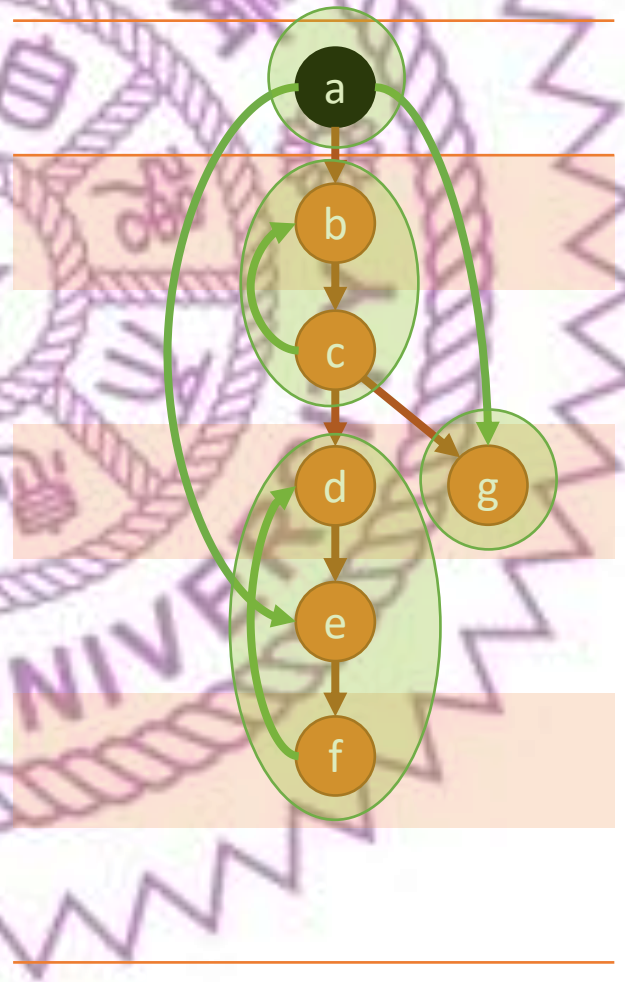


DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

stk

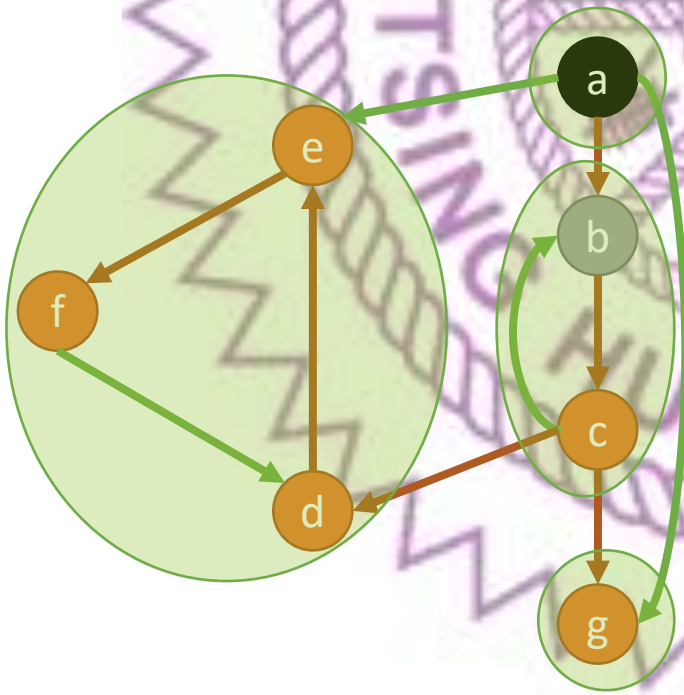


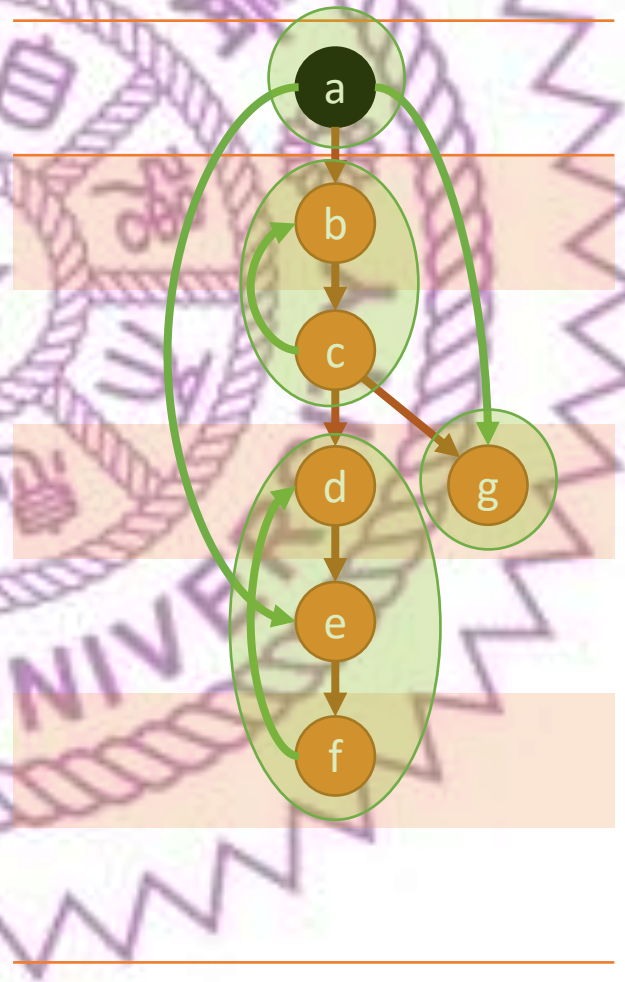


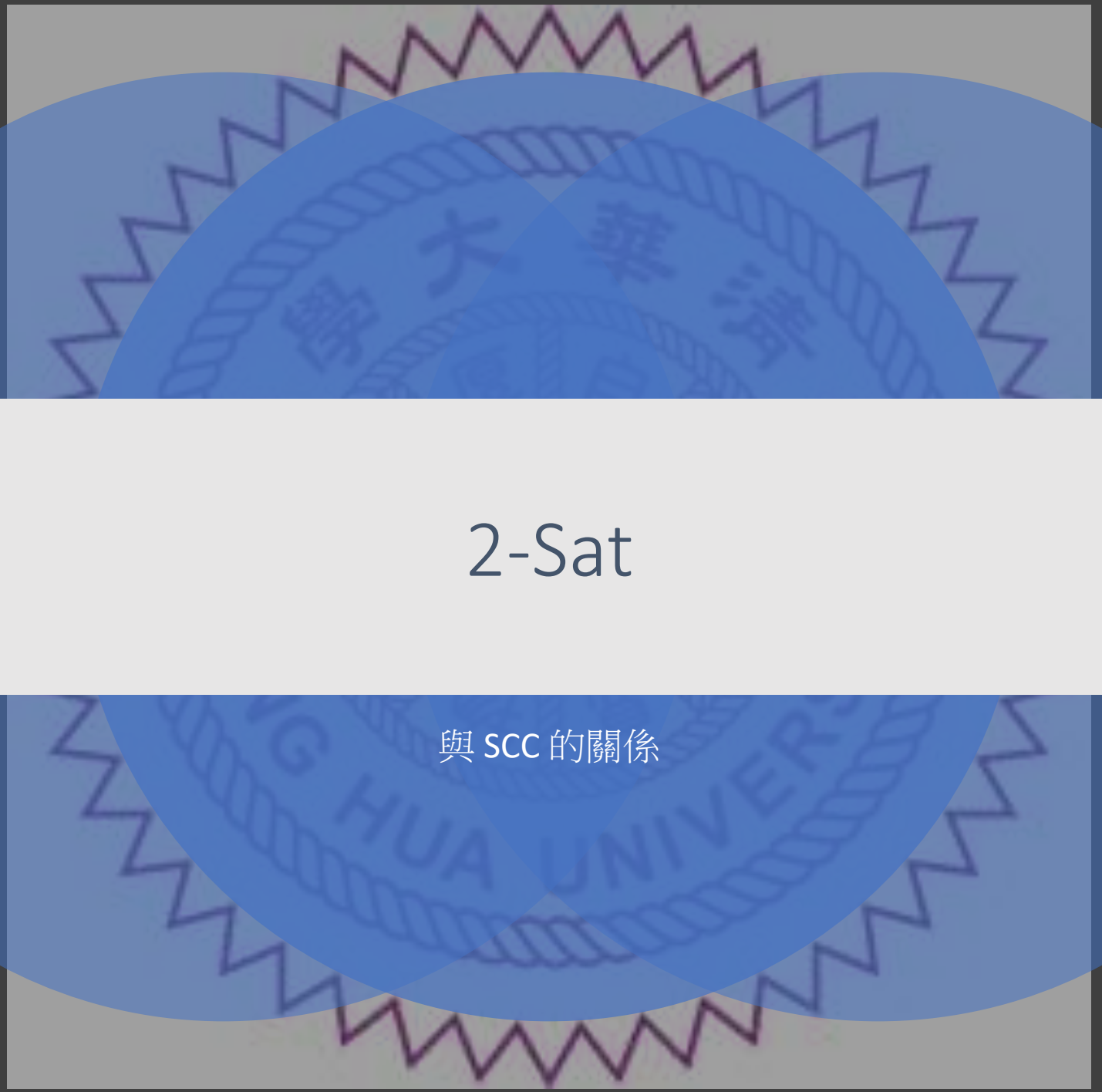
DFS

	a	b	c	d	e	f	g
$deep$	1	2	3	4	5	6	7
low	1	2	2	4	4	4	7

stk







2-Sat

與 SCC 的關係

布林 (boolean) 變數

- 所謂的布林變數，指的是變數只有 true 和 false 兩種狀態
- 假設 a, b 是兩個布林變數，我們來定義以下幾個符號：
  - $a \wedge b$： (and)
    如果 a 是 true 且 b 是 true 的話運算結果就會是 true，反之為 false
  - $a \vee b$： (or)
    如果 a 是 true 或 b 是 true 的話運算結果就會是 true，反之為 false
  - $a \oplus b$： (xor)
    如果 a, b 其中一個 是 true 的話運算結果就會是 true，反之為 false
  - $\neg a$： (not)
    如果 a 是 false 的話結果是 true，反之為 false

2-sat 問題

- 給你 n 個布林變數 $X_1 \sim X_n$
- 給你 m 個條件式，每個條件式只會是以下格式的其中一種：

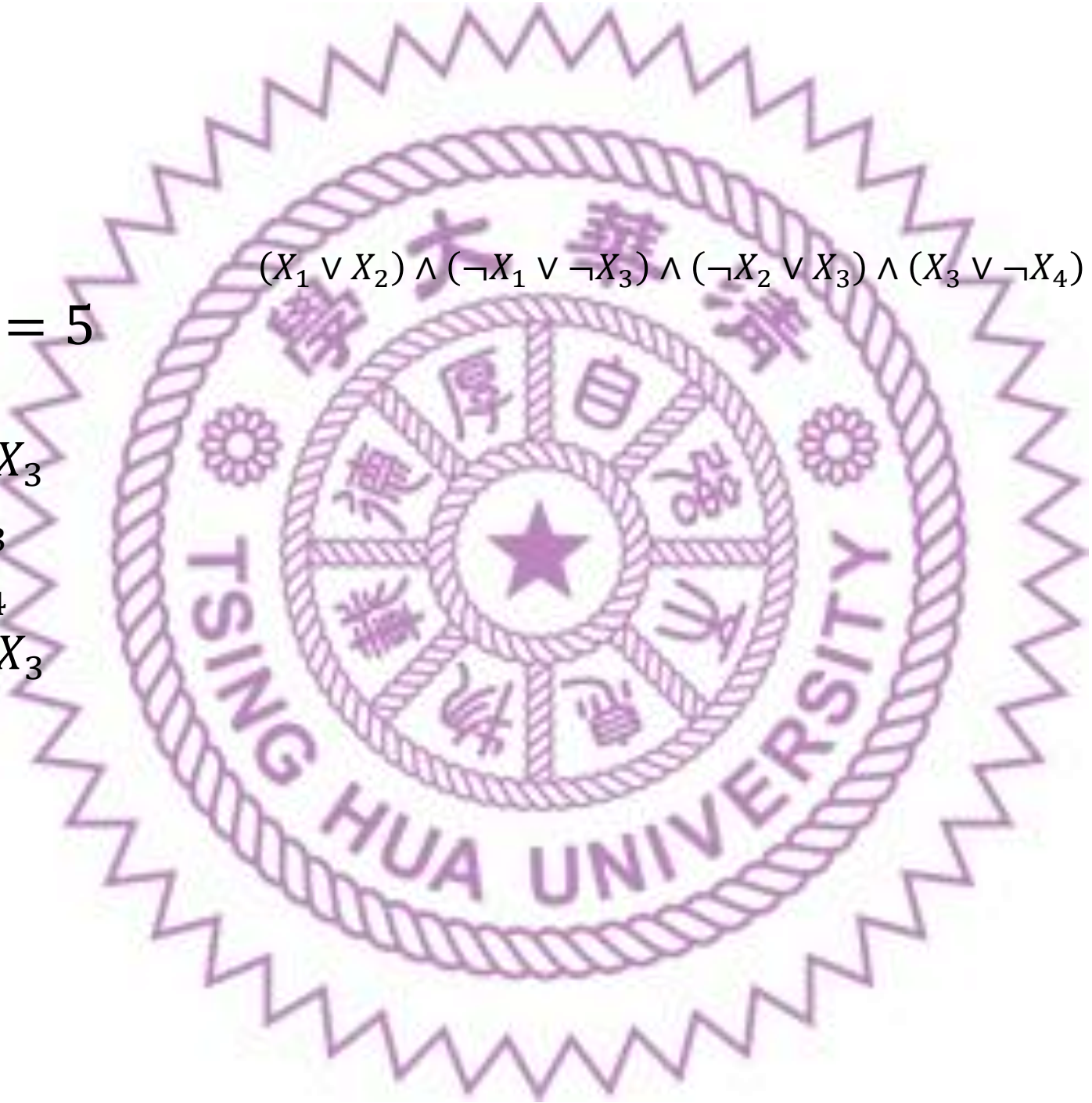
$a \vee b$	$a \oplus b$
$\neg a \vee b$	$\neg a \oplus b$
$a \vee \neg b$	$a \oplus \neg b$
$\neg a \vee \neg b$	$\neg a \oplus \neg b$

- 你的目標是要找出一組 X 的解可以讓所有條件都是 true

範例

$(X_1 \vee X_2) \wedge (\neg X_1 \vee \neg X_3) \wedge (\neg X_2 \vee X_3) \wedge (X_3 \vee \neg X_4) \wedge (\neg X_2 \vee \neg X_3)$

- $n = 4,\ m = 5$
  - $X_1 \vee X_2$
  - $\neg X_1 \vee \neg X_3$
  - $\neg X_2 \vee X_3$
  - $X_3 \vee \neg X_4$
  - $\neg X_2 \vee \neg X_3$



範例解答

$(X_1 \vee X_2) \wedge (\neg X_1 \vee \neg X_3) \wedge (\neg X_2 \vee X_3) \wedge (X_3 \vee \neg X_4) \wedge (\neg X_2 \vee \neg X_3)$

- $n = 4,\ m = 5$
  - $X_1 \vee X_2$
  - $\neg X_1 \vee \neg X_3$
  - $\neg X_2 \vee X_3$
  - $X_3 \vee \neg X_4$
  - $\neg X_2 \vee \neg X_3$

X_1	X_2	X_3	X_4
true	false	false	true

轉化成圖論問題

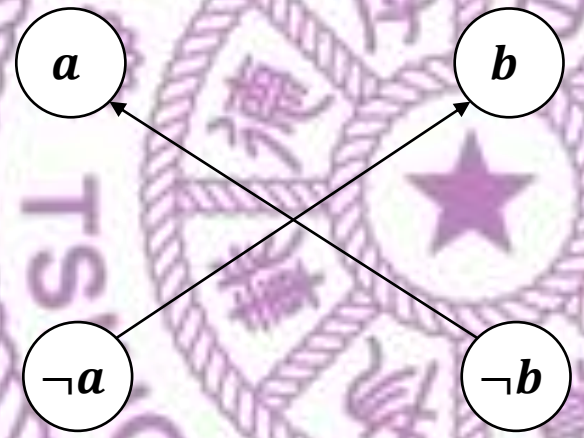
- 對於每個變數 v 建立兩個節點 $v, \neg v$
  分別代表選擇 v 是 true 和 v 是 false 的情況

- 用有向邊 $x \rightarrow y$ 表示如果選擇 x 是 true，那 y 也必須是 true

- 接著會解釋每個 case 要如何建邊

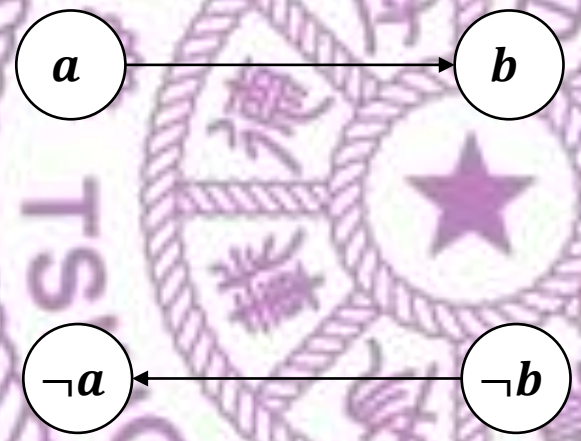
$a \vee b$

- $\neg a \rightarrow b$
- $\neg b \rightarrow a$



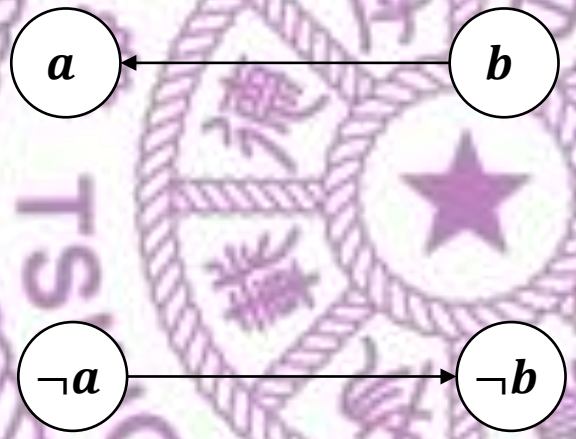
$\neg a \vee b$

- $a \rightarrow b$
- $\neg b \rightarrow \neg a$



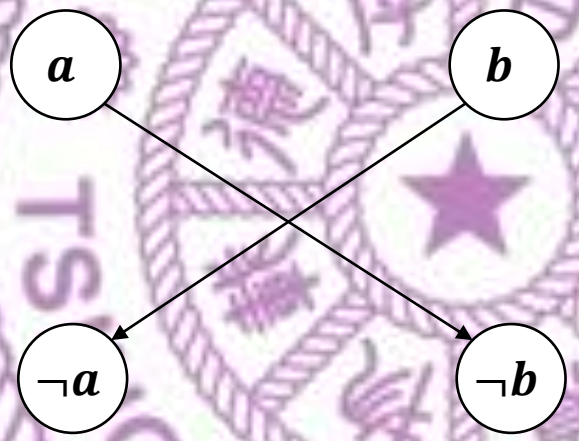
$a \vee \neg b$

- $\neg a \rightarrow \neg b$
- $b \rightarrow a$



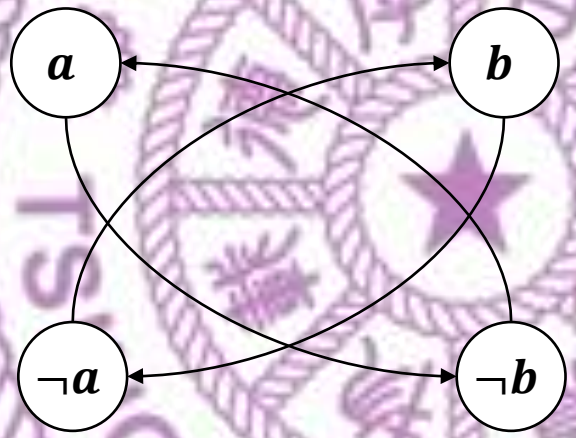
$$\neg a \vee \neg b$$

- $a \rightarrow \neg b$
- $b \rightarrow \neg a$



$$a \oplus b = (a \vee b) \wedge (\neg a \vee \neg b)$$

- $a \rightarrow \neg b$
- $b \rightarrow \neg a$
- $\neg a \rightarrow b$
- $\neg b \rightarrow a$



性質

- 節點 v，$\neg v$ 只能有一個是 true
- 所以如果在建完的圖中發現這兩個點在同一個強連通分量中那顯然就是無解
- 反之就是有解嗎？

找出一組解

- 答案是肯定的
- 原因是找出 SCC 後就能透過縮點後的 DAG 做拓樸排序
  利用拓樸排序就能找出解
- 這裡交給各位思考吧
  https://cp-algorithms.com/graph/2SAT.html