CH6、Graph

圖論

目錄：

定義

術語(10 個)

Complete Graph、Subgraph、Path、Length、Simple Path、Cycle

Connected、Connected Component、Strongly Connected、Degree

表示方式

Adj Matrix、Adj List、Multi Adj List、Index+Array

Traversal(追蹤)

DFS、BFS

Spanning Tree

Minium Cost Spanning Tree

[補充]Union & Find(3 種方法)

Kruskal's、Prim's、Sollin's

Shortest Path Problem

	演算法	假設條件	時間複雜度	
One to all	Dijkstra's	邊的加權不可負值	$O(n^2)$	Greedy Algorithm
One to all	Bellman	允許有負值 但不可有負循環	$O(n^3)$	Dynamic Programming
All to all	Floyd-Warshall	允許有負值 但不可有負循環	$O(n^3)$	Dynamic Programming

A^+, A^*矩陣

AOV Network、AOE Network

Topological Order

Articulation Point、Biconnected Graph、Biconnected Component

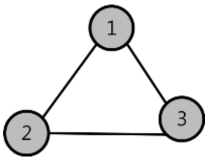
圖形

Def：圖形是由頂點集合以及邊集合所組成，表示如下：

G=　<V,　E>

圖形　頂點集合　邊集合

例：



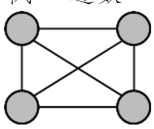
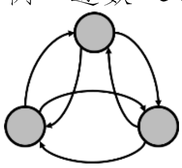
G=<V, E>；V={1, 2, 3}；E={(1, 2), (1,3), (2, 3)}

1. 無向圖 Undirected Graph, UG
   Def：G=<V, E>，若 Vi, Vj ∈ V，則邊(Vi, Vj) = (Vi, Vj)
2. 有向圖 Directed Graph, Digraph
   Def：G=<V, E>，若 Vi, Vj ∈ V，則邊(Vi, Vj)≠(Vi, Vj)

術語

一、Complete Graph(完整圖/完全圖)

Def：

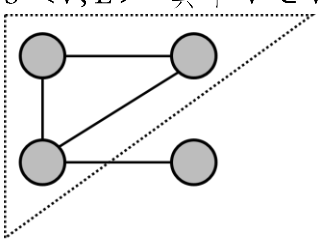
1. 以無向圖而言：n 個頂點，具 n(n-1)/2 條邊稱之(任 2 點可”直接”到達對方)
   例：邊數=4×(4-1)/2 = 6
   
2. 以有向圖而言：n 個頂點，具 n(n-1)稱之
   例：邊數= 3×(3-1) = 6
   

二、Subgraph(子圖)

Def：

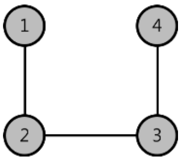
G=<V, E>則 G 的子圖表示：

S=<V', E'>，其中 V' ∈ V，E' ∈ E 稱之



三、Path(路徑)

Def：由邊集合組成



四、Length(長度)

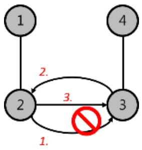
Def：指 Path 中有多少的邊

⇨ 上例：Length：3

五、Simple Path(簡單路徑)

Def：除起點和終點可相同，其餘過程中，不可經過相同頂點

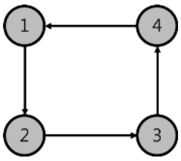
例：



六、Cycle(迴圈)

Def：為一 Simple Path，且起點跟終點必相同

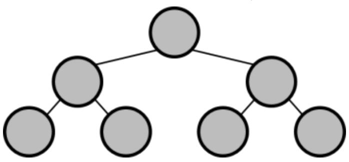
例：

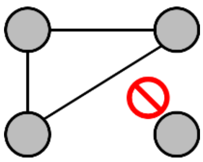


七、Connected(連通)(for UG)

Def：所有任意成對的頂點皆有路徑有相通

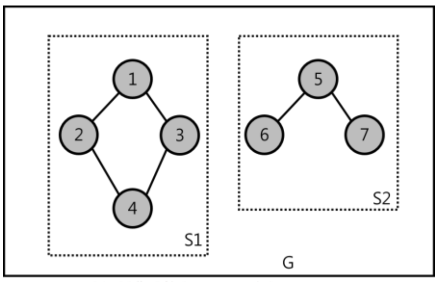
例：Tree 必定連通





八、Connected Component(連通元件)

例：

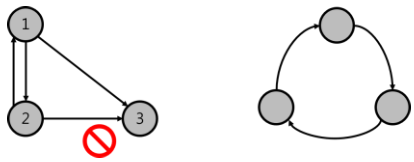


S1, S2為G的Connected Component

九、Strongly Connected(強連通)

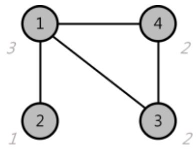
Def：概念同七，但針對有向圖

例：



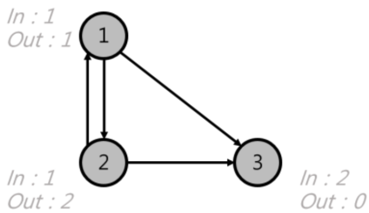
十、Degree(分支度)

1. For UG：



$e = (\sum di) / 2$

2. For DG：令 in-di 為 Vi 的 in-degree，out-di 為 Vi 的 out-degree，則：$\sum$ in-di $= \sum$ out-di $= e$



例：True/False

(1) Tree is connected undirected graph？
(2) G is connected，當|V|=n，則最少邊=n-1？
(3) UG 中，|V|=n，若|E| ≥ n，則必為 Connected？

(1) True
(2) True
(3) False

[補充]演算法

Euler Cycle：指每個邊都經過一次的 Cycle

Hamilton Cycle：指每個頂點都經過一次的 Cycle

Euler Trail：每個邊都經過一次的 Path

Hamilton Trail：每個頂點都經過一次的 Path

圖的表示法

1. 相鄰矩陣(Adjacency Matrix)
2. 相鄰串列(Adjacency List)
3. 相鄰多元串列(Multiple Adjacency List)
4. 索引(Index + Array)

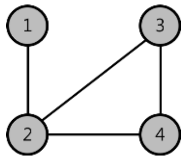
相鄰矩陣(Adjacency Matrix)

一、For UG 而言：

G=<V, E>, |V|=n，|E|=e，則宣告一個二維陣列 A(1: n, 1: n)，其中

$$A[i, j]= \begin{cases} 0, \text{ if } (vi, vj) \notin E \\ 1, \text{ if } (vi, vj) \in E \end{cases}$$

例：

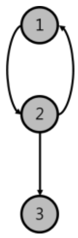


$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & 0 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 \\ 4 & 0 & 1 & 1 & 0 \end{array}$$

⇨ 對稱矩陣為了省空間，只存上三角、或下三角矩陣即可

二、For DG 而言：

例：



$$\begin{array}{c|ccc|} & 1 & 2 & 3 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 3 & 0 & 0 & 0 \end{array}$$

⇨ 無對稱矩陣特質

Note：

1. *欲 check<vi, vj>是否有邊存在，直接 check：A[I, j]之值，若為 0 則不存在、若為 1 則存在* $\Longrightarrow O(1)$
2. *欲計算 vi 之 Degree：O(n)*
   For UG：將第 I 列加總，即是 Loop $\Longrightarrow O(n)$
   For DG：
   找 vi 的 in-degree：第 i 行的加總
   找 vi 的 out-degree：第 i 列的加總
3. *欲求總邊數：*$O(n^2)$
   UG：$e = (\sum di)/2$
   DG：$e = \sum$ *in-di 或 out-di*

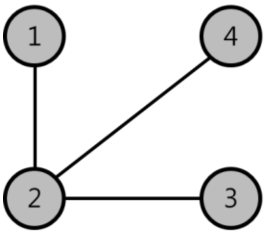
相鄰串列(Adjacency List)

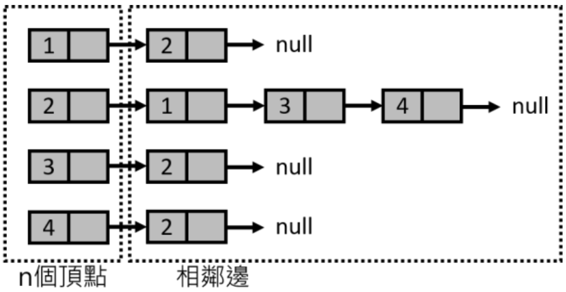
一、For UG 而言

G=<V, E>，|V|=n，|E|=e，以 n 條 link list 表示 Graph，其中 Node Structure 如下：

Vertex	Link
頂點 Number	指向下一個相鄰邊

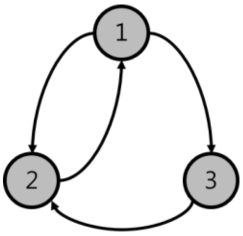
例：

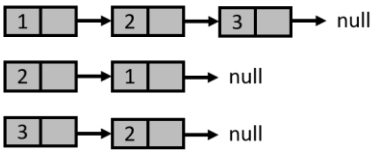




二、For DG 而言：

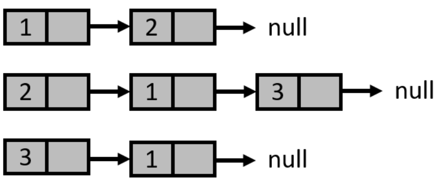
例：





Note：

1. *欲 check<vi, vj>是否有邊存在， check vi 此 list，是否有 vj 存在* $\Longrightarrow O(e)$
2. *欲計算 vi 之 Degree：O(n)*
   For UG：count vi 此條 list 後的 Node 數 $\Longrightarrow degree_i \Longrightarrow O(e)$
   For DG：
   找 vi 的 in-degree：從各 list，找是否有存在 vi 之總數 $\Longrightarrow O(n+e)$
   找 vi 的 out-degree：同 UG $\Longrightarrow O(e)$
3. *欲求總邊數：O(n+e)*
   UG：$e = (\sum di)/2$
   DG：$e = \sum$ *in-di 或 out-di*
4. *欲加速，建反向相鄰串列(以 in-degree 表示)*
   同上例：



Count vi list 上的 Node 數，即 in-di $\Longrightarrow O(e)$

比較表

	Adj Matrix	Adj List
頂點多、邊少	O(n2)：不好，且 Sparse Matrix(耗空間)	適合
邊多	好	不好
Check 邊是否存在	O(1)	O(e)，不佳
求總邊數 或連通等應用*(註)*	$O(n^2)$，較差	O(n+e)，較佳

註：
當 e≪n2，ex：e=n-1，則 O(n+e)，Adj List 較佳
當 e≈n(n-1)/2，所以 $O(n+e)=O(n+n^2)$，則 Adj List 較差

例：欲使印出所有和 vi 相鄰頂點之時間和 vi 之相鄰頂點數”成正比”，需採用何種方式？
以 Adj List 表示；Adj Matrix 總是印同樣次數

相鄰多元串列(Multiple Adjacency List)

Def：

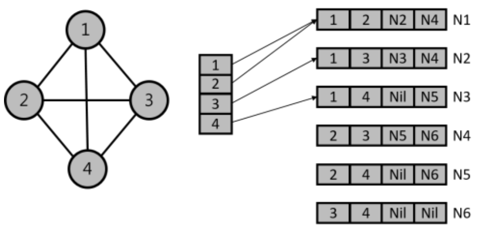
1. 圖形個邊以 Edge Node 表示，結構如下：

vi	vj	Link for vi	Link for vj

Link for vi 指向包含 vi 的下一個 Edge Node
Link for vj 指向包含 vj 的下一個 Edge Node

2. 另外準備一陣列 Vertex[1: n]，其中 Vertex[i]為指向第一個出現 vi 的 Edge Node

例：

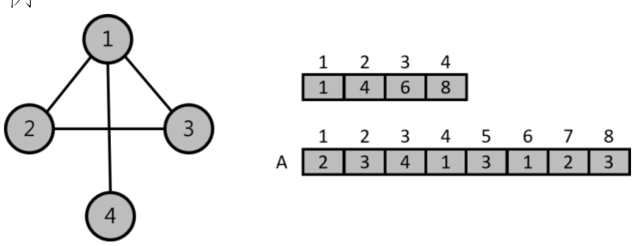


索引(Index + Array)

Def：準備

1. 一維陣列 A：記錄所有頂點之相鄰頂點編號
2. 用一 index，記錄各頂點在 array 中之起始位置

例：



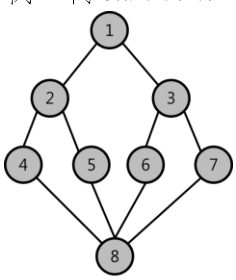
DFS 深先搜尋(Depth First Search)

目的：拜訪所有 Node 一次

步驟：挑一個頂點為拜訪起點(s)

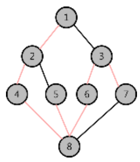
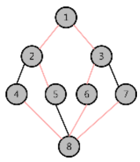
1. 從 s 拜訪來走訪的相鄰點
2. 若　(1)有：目前 Node 設為 s
　　　(2)無：回溯到目前 Node 之上一個 Node，並 goto 1.
3. Repeat 1.及 2.，直到　(1)皆拜訪順利完成
　　　(2)若無頂點可挑，又未拜訪完(不連通)

例：問 Start Vertex 為 v1、v5，則 DFS 結果為何？



1. v1, v2, v4, v8, v5, v6, v3, v7

2. v5, v2, v1, v3, v6, v8, v4, v7

(習慣上以選小的優先)

例：承上，下列何者非 DFS 之結果？

1. 1, 3, 6, 8, 7, 5, 2, 4
2. 1, 2, 5, 8, 6, 3, 7, 4
3. 1, 2, 4, 8, 6, 7, 3, 5
4. 1, 3, 6, 8, 4, 2, 7, 5

3、4

程式：

```
//visited[1: n] = init= False
//代表 vi 是否走訪

Procedure DFS(v: integer)
        var w: integer;
        begin
                visited[v]=true;
                for each w ∈ Adj(v)
                        if not visited[w] then DFS(W);
        end
```

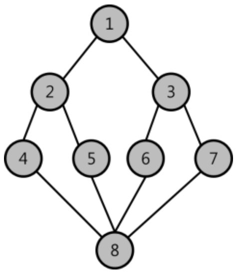
Note：欲 check 圖形是否連通，可採用 DFS 或 BFS 之後，check visited[1: n]之內，是否皆為 true，若否，則不連通

BFS 廣先搜尋(Breath First Search)

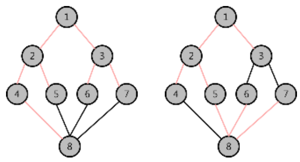
步驟：挑一個 start vertex: s，將 s 加入佇列 Q 中，之後：

1. delete Q
2. 將得到值之未走訪相鄰點，加入佇列 Q 中
3. Repeat 1.及 2.，直到 Q 為空則停

例：從 v1、v5 開始？



1. *v1, v2, v3, v4, v5, v6, v7, v8*
2. *v5, v2, v8, v1, v4, v6, v7, v3*



(結果不見得唯一)

程式：

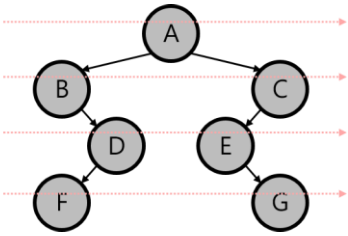
```
//visited[1: n] = init= False
//代表 vi 是否走訪

Procedure BFS(v: integer)
        var w: integer;
        begin
                visited[v]=true;
                ini Q(q);                    //清空佇列
                Enqueue(q, v);               //將 v 加入 q

                while(not Empty(q)) do
                begin                        //將 v 未走訪的相鄰點加到 Queueci
                        Dequeue(q, v);       //從 q 刪一筆給 v
                        for each w ∈ Adj(v) do
                                if not visited[w] then
                                begin
                                        visited[w]=true;
                                        Enqueue(q, w);
                                end
        end
```

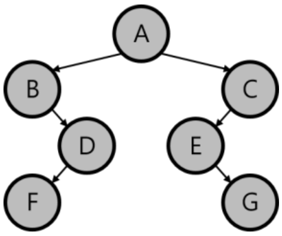
小結：

	DFS	BFS
	Stack	Queue
以相鄰矩陣表示	$O(n^2)$	$O(n^2)$
以相鄰串列表示(以演算法角度)	O(n+e)	O(n+e)
以相鄰串列表示(以 DS 角度)	*O(e)*	*O(e)*

考試一般以演算法版為主

[補充]

1. BFS 應用
   Binary Tree 的 level order traversal：若將 Tree 視為圖，則等同於採 BFS
   Level order traversal 定義：由上而下，由左而右拜訪 Node
   例：
   

   A, B, C, D, E, F, G
2. DFS 應用(root 為 start vertex) ≈ preorder(DLR)
   例：
   

   A, B, D, F, C, E, G

Graph Traversal **應用**

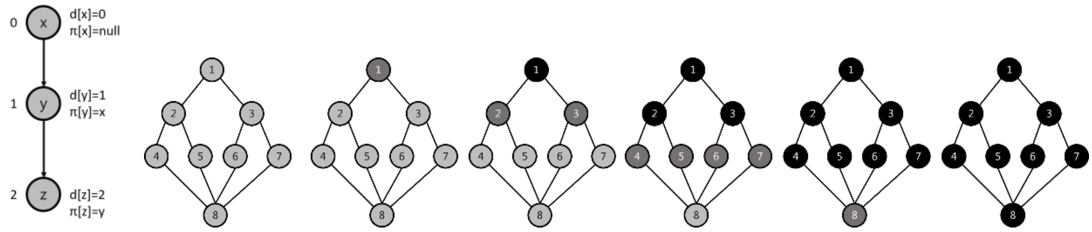
1. check graph 是否連通
2. 找出圖形的連通元件(單元)
3. 找出一連通圖的"Spanning Tree(後面會談)"
4. Check 圖是否有 cycle(本章談)

[演算法版]之 BFS、DFS

BFS 之演算法版

說明：

1. 頂點分三類
   (1) 白色：未走訪
   (2) 灰色：放入 Queue 中
   (3) 黑色：已走訪
2. 令 d[u]表起點到 u 之 path length
3. 令 π[u]表頂點 u 之 parent in BFS order 之中



程式：

```
BFS(s: start vertex)
{
        for each u G.v.-{s} do                //初始化
        {
                color[u]=white;
                d[u]=;
                π[u]=null;
        }
        color[s]=gray;
        d[s]=0;     π[s]=null;   createQ(Q);
        Enqueue(Q, s);
        while(Q!=null) do
        {
                u=dequeue;                    //1.
                for each v ∈ Adj(u) do        //2.處理走訪的相鄰點
                {
                        if(color[v]==white)
                        {
                                color[v]=gray;
                                d[v]=d[u]+1;
                                π[v]=u;
                                Enqueue(Q, v);
                        }
                }                             //3.
                color[u]=Black;
        }
}
```

Note：在各邊沒有加權值情況下，若給一無向圖，要求出"某一點(如 s)到各頂點之最短路徑長度"，可以用 BFS(s)之演算法求得(各 Node 的 d[u]即是)

DFS 之演算法版

說明：在有向圖中：

DFS 邊分為：

1. Tree Edge
2. Back Edge
3. Forward Edge
4. Cross Edge

Note：

Edge(U, v)可用 color 作判斷，當此邊是 First expored*(第一次探索)*，若 color(v)為：

1. white ⟹ (u, v)是 Tree Edge
2. gray ⟹ (u, v)是 Back Edge
3. black ⟹ (u, v)可能是：
   (1) Forward Edge：if d[u] < d[v]*(點 u 比點 v 早探索)*
   (2) Cross Edge：if d(u) > d[v] *(點 u 比點 v 晚探索)*

1. 無向圖中，只有 Tree Edge 及 Back Tree
2. 無 Back Edge ⟹無 cycle
   有 Back Edge ⟹有 cycle

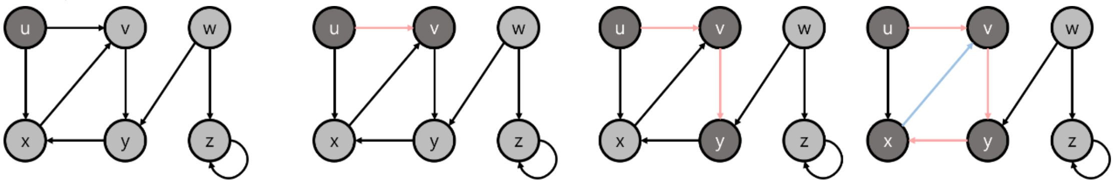
程式：

```
DFS(G)
{
        for each vertex u ∈ G.v. do     //初始化：O(|v|)
        {
                color[u]=white;
                π[u]=null;
        }
        Time=0;                         //全域
        for each vertex u ∈ G.v. do     //O(|v|)
                if(color[u]==white)
                        then DFS-visited(u);
}
DFS-visited(u)
{
        color[u]=gray;
        Time=Time+1;
        d[u]=Time;
        for each v ∈ Adj[u] do
        {
                if(color[v]==white) do
                {
                        π[v]=u;
                        DFS-visited(v);     //Recursive
                }
        }
        color[u]=Black;
        f[u]=Time=Time+1;
}
```

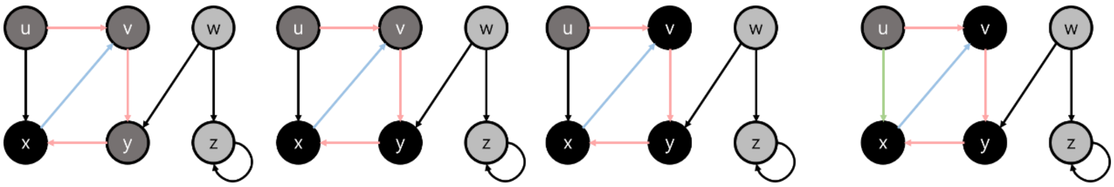
Note：
d[u]代表 vertex u 於 DFS 中的 first discover time
f[u]代表 vertex u 於 DFS 中的 finished time

遇到白 ：Tree Edge(紅)
遇到灰 ：Back Edge(藍)
遇到黑 、本身較小/早 ：Forward Edge(綠)
遇到黑 、本身較大/晚 ：Cross Edge(黃)

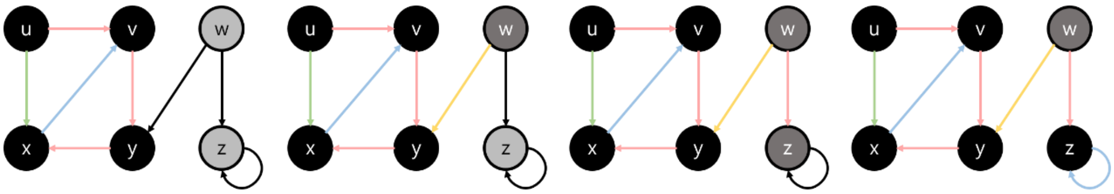
1~4 步驟：



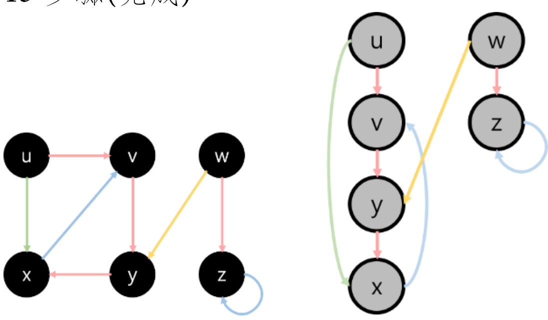
5~8 步驟：



9~12 步驟



13 步驟(完成)：



	u	v	w	x	y	z
起始時間	01	02	09	04	03	10
結束時間	08	07	12	05	06	11

Cycle 的判別：

在 DFS 中，若有 Back Edge，則有 cycle，即(u, v)若：

1. color[v]=gray，且
2. $\pi[u] \neq v$

則有 cycle

程式：

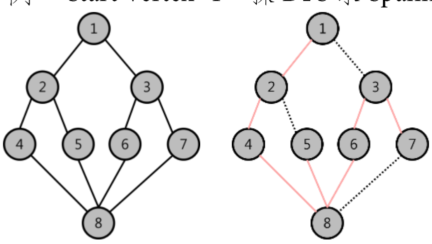
```
是否有 Cycle
DFS-visited(u)
{
        color[u]=gray;
        for each v ∈ Adj[u] do
        {
                if(color[v]==white) do
                {
                        π[v]=u;
                        DFS-visited(v);    //Recursive
                }
                if(color[v]==gray && π[uu]≠v)
                        return true;        //有 cycle
        }
        color[u]=Black;
        f[u]=Time=Time+1;
}
```

Spanning Tree(展開樹、擴張樹)

Def：若一無向圖 G=<V, E>，其中|V|=n，|E|=e，若為 Connected，則其 Spanning Tree: S=<V, T>，其中 T 為 DFS 或 BFS 所經過的邊集合，令 B 為未走過的邊集合，則 S 具：

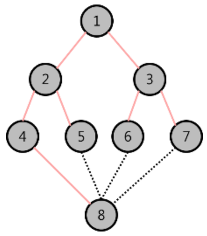
1. E=T+B
2. 自 B 取任一邊加入 T，會有 cycle
3. S 中任何成對頂點，只存在一條 Simple Path

例：Start Vertex=1，採 DFS 求 Spanning Tree



Spanning Tree 不見得唯一

例：承上，採 BFS 之 Spanning Tree？



結論：

1. G 不為 Connected ⟹ 無 Spanning Tree
2. G 的 Spanning Tree 可能 ≥ 1 個
3. Spanning Tree S=<V, T>，|V|=n，|T|=n-1(邊為頂點數-1)
4. Spanning Tree 無 Cycle
5. Tree 一定是 Spanning Tree

最小成本擴張樹(Minimum Cost Spanning Tree, MCST)

Def：G=<V, E>為一 Connected Undirected Graph，且 Edge 上"會有加權"或"成本"，則在 G 中所有可能的 Spanning Tree 中，挑出成本總和最小的 Spanning Tree 謂之

應用：

1. 電路佈局期望連通之最小成本
2. n 個城市之交通連線之最小成本

Note：

1. *若所有邊 cost 皆不相同 ⟹MCST 只會有一個*
2. *當多個相同成本的邊 ⟹MCST 則不見得唯一*

MCST 方法*(Shortest Path Problem 也是 3 種方法)*

演算法	特性	Time Complexity
Kruskal's	邊	O(|E|log|E|)
Prim's	頂點	$O(\|V\|^2)$
Sollin's	Tree	$O(\|V\|^2)$

上述的演算法皆採用 Greedy 演算法*(Huffman 亦採用此法)*

[補充]

Disjoint Set：

Def：各 Set 均無共通元素、相互的交集為 Φ

Ex：s1={1, 7, 8, 9}、s2={2, 5, 10}、s3={3, 4, 6}

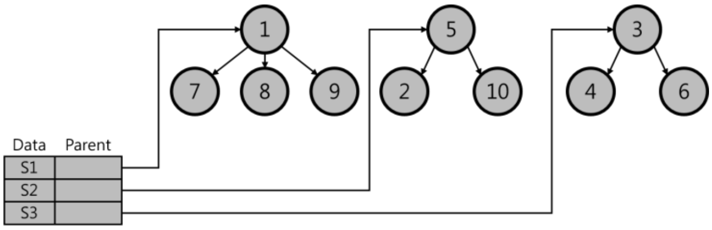
表示方式採"Tree"後續可運用於 Graph

表示方式：
[法一]以 Linked List 呈現
Node structure：

Data	Parent

Parent 會指向各 Set 的父點，欲表示 Set 任取一元素為 Root，其餘為子樹

例：



[法二]以 Array 呈現
承上例：

Data	1	2	3	4	5	6	7	8	9	10
Parent	Φ	5	Φ	3	Φ	3	1	1	1	5

註：Φ 代表 Null

Union & Find 運作

1. Union(i, j)：將 Set_i、Set_j 聯集(即合併成一個 Set)
2. Find：找出元素位於之集合 ⟹ 大多用於”Check x, y，2 元素是否在相同集合”，if(Find(x)==Find(y))then 同一集合；else 不同集合

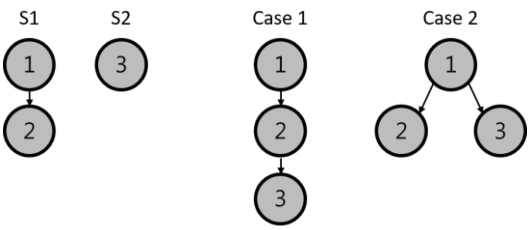
Note：Find(x)：找出 Root 所在

例：Find(7)=1、Find(4)=3

可知：7 與 4 於不同集合

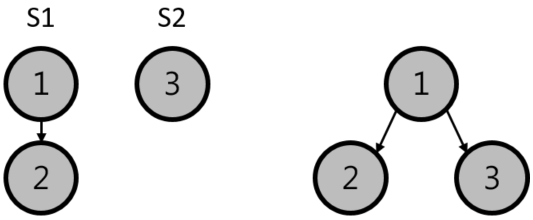
Implement Union & Find

[法一]任意 Union 與 Simple Find
概念：



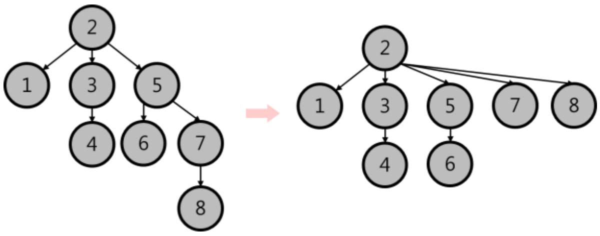
問題：Union 最差之下：此時 Find(x)=O(n)

[法二]Union-by height 與 Simple Find
Def：高的合併低的
例：



特質：高度同的 2 個 Set，Union 結果：高度不會變，因此：樹高= O(log n) ⟹ Find(x)=O(log n) ⟹ 類似 Binomial Heap

[法三]Union by height & Find with path compression
Def：在尋找 x 的 Root 過程中，x 到 Root 之路徑上所有 Node(除 Root 之外)，之 Parent 均改成指向 Root
例：



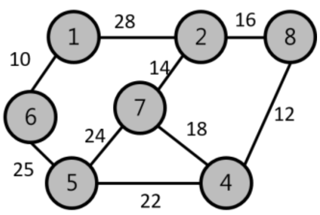
之後 Find 的 Time Complexity：O(Ackerman(m, n)) = O(α (m, n)) ≈ O(1)

結論：欲做 m 次 Union/Find 動作，總花費時間為：O(m*α (m, n)) ≈ O(m)
[法一] < [法二] < [法三]之效能
O(n) < O(log n) < O(1)

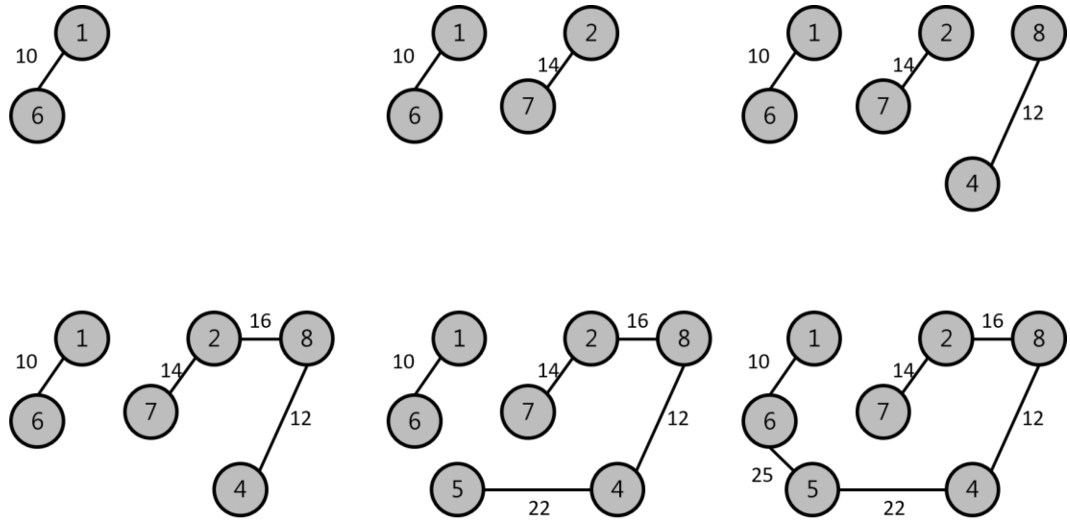
Kruskal's 演算法

步驟：
1. 依序挑最小成本邊(vi, vj)
2. 若(vi, vj)之加入不會使 Spanning Tree 形成 cycle，則加入；否則放棄之
3. Repeat 1.、2.，直到挑了 n-1 個邊(若挑完不足 n-1 邊，表示不連通)

例：



步驟 1~6：



程式：

```
T={Φ};                                          //空集合
while((|T|<n-1) && (E not empty))
{
        choose (v, w) from E of lowest cost;    //1.
        delete (v, w) from E;
        if((v, w) not create cycle in T)        //2.
                add discard (v, w);
        else
                discard (v, w);
}
if(|T|<n-1)
        cout<<"No Spanning Tree.";              //3.
```

Time Complex：

O(e log e), |E|=e

說明：

-最多執行 e 回合(無邊可挑)

-e=|E|：總邊數

-又各回合需做：

(1) delete min cost edge(*用 Heap，所以 O(log e)*)

(2) check 加入 T 是否有 cycle(*採 Union & Find 之方法：O(1)*)

⇨ e*(log e +1) ⟹ O(e log e)

How to check 加入 T 是否有 cycle？

用 Disjoint Set 的 Union & Find 作：

程式：

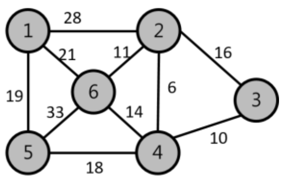
```
if(Find(u) ≠ Find(q))
        add(u, w) to T;
else
        discard (u, w);      //代表有 cycle
```

Prim's 演算法

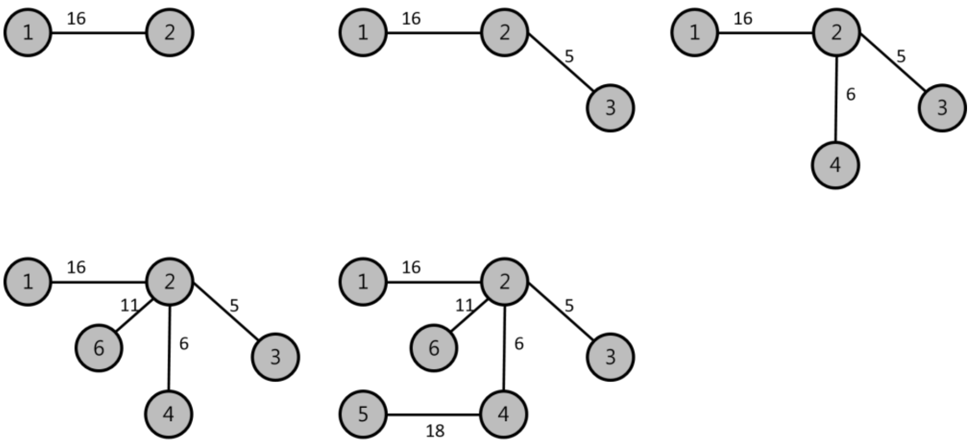
作法：

1. 一圖 G=<V, E>含有 n 個頂點，V={1, 2, 3, …, n}，另外設 U={1}，尋找一最短的邊(u, v)，其中 $u \in U$、$v \in V$
2. 將邊(u, v)中的 v 頂點加到 U 集合中
3. Repeat 1.、2.直到 U=V 或無邊可挑*(代表不連通)*

例：



步驟 1~5：



程式：(令 G 至少有一個 Vertex)

```
Tv={1};
for(T=0; |T|<n-1; add(u, v) to T)
{
        Let(u, v) be a least-cost edge such that u ∈ Tv and V ∉ Tv;   //1.
        if(no such edge)                                             //2. ⟹ O(n²)
                break;
        add v to Tv;
}                                                                    //3.
if(|T|<n-1)
        cout<<"No Spanning Tree";
```

比較表

	Kruskal's	Prim's
異	以"Edge"為出發點	以"Vertex"為出發點
	需 check cycle	無 cycle 議題
	需於一開始，即知所有成本 (在 data stalbe 之下適用 ，變動頻繁下不適用)	只需知道和頂點相關的邊 (相反，Internet 適用)
同	求 MCST	

Time Complexity：

	Kruskal's	Prim's
[DS 版]Adj Matrix	O(|E|log|E|)	$O(n^2)$
[演算法版]Adj List(Binary Heap 製作)	O(|E|log|E|)	O(|E|log|V|)
[演算法版]Adj List(Fibonacci Heap 製作)	O(|E|log|V|)	O(|V|log|V|+|E|)

[DS 版]：效能分析

平均下 Kruskal's 的效率較佳

當$|E|=e$ 很小(ex：$e=n-1$)，則 $O(e\log e) \Longrightarrow O(n \log n) < O(n^2)$

⇨ 適用 Kruskal's

當$|E|$很大(ex：$e=(n*(n-1)/2)$，則 $O(e \log e) \Longrightarrow O(n^2 \log n^2) > O(n^2)$

⇨ 適用 Prim's

Sollin's 演算法

Def：

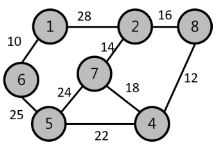
1. 初始時，各頂點各自為一 Set
2. (1)針對各 Tree，挑出 Min Cost 之 Tree Edge

   (2)刪除重複挑選的 Tree Edge

   (3)repeat (1)與(2)，直到只剩一棵 Tree 或無邊可挑(No Spanning Tree)

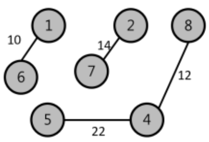
   (4)if $|T|<n-1$，則 No Spanning Tree*(圖不連通)*

例：

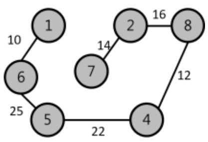


步驟 1~2：

(1, 6)
(2, 7)
(3, 4)
~~(4, 3)~~
(5, 4)
~~(6, 1)~~
~~(7, 2)~~



(6, 5)
(2, 3)
~~(3, 2)~~

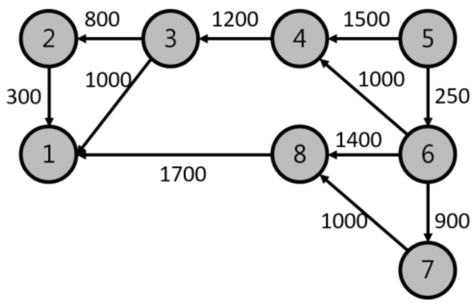


	演算法	假設條件	時間複雜度	
One to all	Dijkstra's	邊的加權不可負值	$O(n^2)$	Greedy Algorithm
One to all	Bellman	允許有負值 但不可有負循環	$O(n^3)$	Dynamic Programming
All to all	Floyd-Warshall	允許有負值 但不可有負循環	$O(n^3)$	Dynamic Programming

單一頂點到所有頂點的 Shortest Path

Dijkstra's 演算法(Dij)

例：從 v5 到各點之 Shortest Path=？



	已挑集合	此次挑選	DIST(Array)，vi 到各點的 Shortest Path							
Pass	S	Vertex Sellected	1	2	3	4	5	6	7	8
Initial	-	5	∞	∞	∞	1500	0	250	∞	∞
1	5	6	∞	∞	∞	1250	0	250	1150	1650
2	5, 6	7	∞	∞	∞	1250	0	250	1150	1650
3	5, 6, 7	4	∞	∞	2450	1250	0	250	1150	1650
4	4, 5, 6, 7	8	3350	∞	2450	1250	0	250	1150	1650
5	4, 5, 6, 7, 8	3	3350	3250	2450	1250	0	250	1150	1650
6	3, 4, 5, 6, 7, 8	2	3350	3250	2450	1250	0	250	1150	1650
7	2, 3, 4, 5, 6, 7, 8	1	3350	3250	2450	1250	0	250	1150	1650

Note：演算法所需之資料結構

1. Cost[i][j]= ∞, (vi, vj) ∉ E
(成本矩陣) weight, (vi, vj) ∈ E
0, i=j

2. s[i]= 0, v0 到 vi 之 Shortest Path 尚未找到
1, v0 到 vi 之 Shortest Path 已經找到

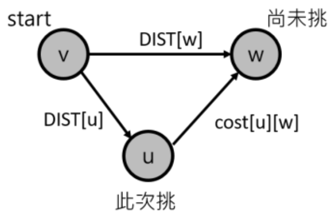
3. DIST[1: n] Array 中，DIST[i]代表 v0 到 vi 的 Shortest Path

程式：

```
Dijs(v, cost[][],    DIST[],            n)
//          成本矩陣   shortest path       頂點數量
{
        boolean s[1:n];
        int num;                                    //回合
        for(i=1; i<=n; i++)
        {
                s[i]=0;
                DIST[i]=cost[v][i];
        }
        s[v]=1;      DIST[v]=0; num=2;              //初始化
        while(num<n)                                //每回合之處理
        {
                for(all w with s[w]=0)
                        choose u;                   //從未選中挑最小值
                s[u]=1;
                for(all w with s[w]=0)
                        DIST[w]=min{DIST[w], DIST[u]+cost[i][w]};
        }
}
```

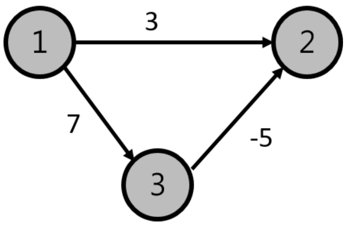
⇨ $O(n^2)$

概念：



⟹ v→w 的 Shortest Path 為：DIST[w]=min{DIST[w], DIST[u]+cost[i][w]}

思考：不可有負邊



先挑 v2 之再挑 v3 ⟹ 但非 Shortest Path，因為 v1→v2→v3 更短，不能更動，因為 v2 已挑選

因此，當做 one-to-all Shortest 且有負邊時，需採用 Bellman-Ford(Dynamic Programming)來求解

Bellman-Ford 演算法

做法：

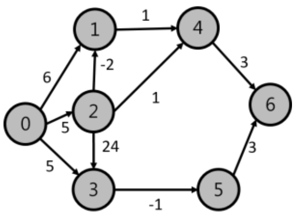
令 Dist[1: n]為一維陣列，其中：

$Dist^l[u]$：代表從起點(v0)到頂點 u 之 Shortest Path*(且此 Path 之總和要≤l，v0 到 u 最多只能走 l 個邊)*

步驟：

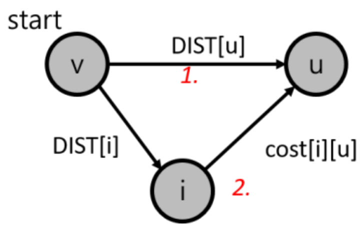
1. $Dist^1$=cost matrix，l=1 ⟹ 需直接到達
2. 依序求出 $Dist^2, Dist^3, \ldots, Dist^{n-1}$*(n 個頂點最多只會經過 n-1 個 Path)*

例：start：v0，採 Bellman，求 Shortest Path？



Pass	Dist[0: 6]						
k	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	5	5
5	0	1	3	5	0	5	3
6	0	1	3	5	0	5	3

概念：



Distk[u]=min{Distk-1[u], Distk-1[i]+cost[i][u]}

程式：

```
for i=0 to (n-1) do
        Dist[i]=cost[v][i];
for k=2 to (n-1) do
        for(each u(u≠v) and u 至少有一射入邊)
                for(each <i, u> in Graph)
                        if(Dist[u] > Dist[i]+cost[i][u])
```

⇨ 不能有負循環

All to all 之 Shortest Path(**任意成對頂點之** Shortest Path)

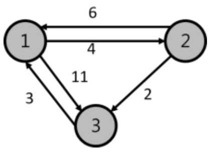
[法一]用 Dij 演算法做 n 次即可 $\Longrightarrow n^*O(n^2) \Longrightarrow O(n^3)$，但不能有負邊，故：

[法二]採用 Floyd-Warshall

Def：G=<V, E>，|V|=n，|E|=e，則 D^k 矩陣為 n×n 矩陣，其中 $D^k[i, j]$代表 vi 到 vj 的 Shortest Path，且中途經過的頂點編號≤k

Note：k⟹0~n ⟹(|V|)頂點決定

例：G=<V, E>，求 all to all 之 Shortest Path？



k=0~3

D^0=w(成本矩陣) 或 cost matrix

$$\begin{array}{c|ccc|} & 1 & 2 & 3 \\ 1 & 0 & 4 & 11 \\ 2 & 6 & 0 & 2 \\ 3 & 3 & \infty & 0 \end{array}$$

D^1=

$$\begin{array}{c|ccc|} & 1 & 2 & 3 \\ 1 & 0 & 4 & 11 \\ 2 & 6 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{array}$$

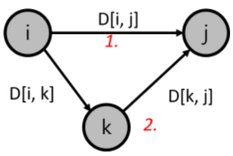
D^2=

$$\begin{array}{c|ccc|} & 1 & 2 & 3 \\ 1 & 0 & 4 & 6 \\ 2 & 6 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{array}$$

D^3=

$$\begin{array}{c|ccc|} & 1 & 2 & 3 \\ 1 & 0 & 4 & 6 \\ 2 & 5 & 0 & 2 \\ 3 & 3 & 7 & 0 \end{array}$$

概念：



Distk[u]=min{Distk-1[u], Distk-1[i]+cost[i][u]}

遞迴表示法：

$$D[i, j] = \begin{cases} w[i][j], & \text{if } k=0 \\ \min(D^{k-1}[i, j], D^{k-1}[i, k]+D^{k-1}[k, j]), & \text{if } k\geq 1 \end{cases}$$

程式：

```
void floyd(int n, int w[][], int D[][])
{
        int i, j, k;
        D=w;                                    //D⁰初始值
        for(k=1;k<=n;k++)                       //回合
                for(i=1;i<=n;i++)               //算此回合各成對頂點之 Shortest Path
                        for(j=1;j<=n;j++)
                                D[i][j]=min(D[i][j], D[i][k]+D[k][j]);
}
```

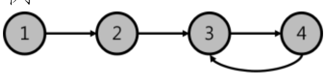
A^+(Transitive Closure Matrix)遞移封閉矩陣

Def：一 n×n 矩陣，|V|=n，其中：

A+[i, j]= 1, if i 到 j 有 Path，且 Path Length≥0

0, otherwise

例：



A^+矩陣

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

$A^*=A^+\cup$單位矩陣(對角線(v_i, v_j)必為 1*(i=j)*)

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

程式：$O(n^3)$

```
A+(adjM,   A,              n)
//相鄰矩陣 結果的矩陣      頂點數
{
        for i=1 to n do
        {
                for j=1 to n do
                {
                        A[i, j] = adjM[i, j];       //將 adjM 值給 A
                }
        }
        for k=1 to n do
                for i=1 to n do
                        for j=1 to n do
                                A[i, j]=A[i, j] or (A[i, k] and A[k, j]);      //i->k->j
}
```

A^*(Reflexive Transitive Closure Matrix)**反射遞移封閉矩陣**

$A^*=A^+\cup$單位矩陣

```
A*(adjM,    A,                    n)
//相鄰矩陣 結果的矩陣        頂點數
{
        for i=1 to n do
        {
                for j=1 to n do
                {
                        A[i, j] = adjM[i, j];        //將 adjM 值給 A
                }
        }
        for k=1 to n do
                for i=1 to n do
                        for j=1 to n do
                                A[i, j]=A[i, j] or (A[i, k] and A[k, j]) or (i=j);     //i->k->j
}
```

AOV Network(Activity on Vertex)

Def：G=<V, E>為一有向圖，其中 V 表示 activity，邊(v_i, v_j)表示 v_i 的工作需在 v_j 之前完成

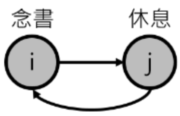
例：$v_i \rightarrow v_j$

用途：安排工作之執行順序

判別：

若 AOV 中：

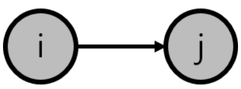
1. 有 cycle：計畫不恰當*(沒有 Topological order 拓樸順序)*
2. No cycle：計畫恰當*(有≥1 組的 Topological order 拓樸順序)*



Topological Order(TO)**拓樸順序**

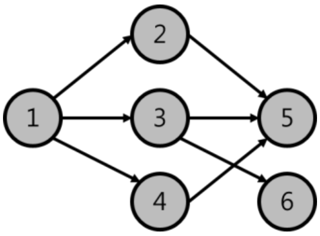
Def：於 AOV 中，若 vi 為 vj 之前導，則在判別計畫是否恰當的過程中，會得出一組拜訪順序，此順序中，vi 必在 vj 之前，謂之

例：



v_i, v_j

例：AOV Network，求 TO？

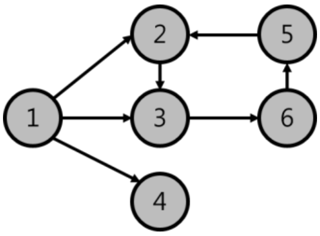


1. *將沒有前導的 Vertex vi output*
2. *將 vi 之射出邊 delete*
3. *Repeat 1.、2.直到 Vertex 皆 output、或無頂點可挑 ⟹ 有 cycle 則無 TO*

v1, v2, v3, v4, v5, v6

Note：TO 不見得唯一

例：



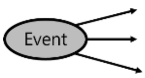
AOE Network (Activity on Edge)

Def：令 G=<V, E>為一有向圖，其中：

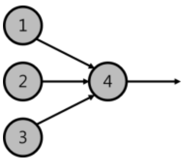
1. Vertex 代表 Event
2. Edge 代表 Activity
3. Edge 上的數值代表所需的工作時*(ex：天數)*

特色：

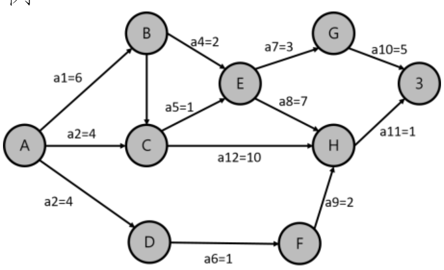
1. Event 要等所有的 leading-in edge 完成，方可開始



2. 當 Event 發生後，其 leading-out 的工作方可開始

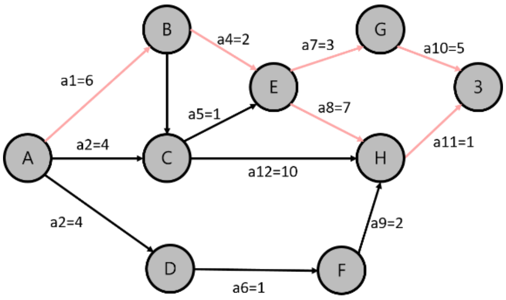


例：



例：

1. 完成工作最短幾天？
2. F 是否可 delay，若可，可 delay 幾天
3. A12 的最早、最晚開始時間為何？



1. *即找出"Critical Path 臨界路徑"：從 start→end 所需的最長路經*
   由圖可知：Critial Path 有 2 條：
   (1) *A, B, E, G, I*
   (2) *A, B, E, H, I*
2. *若在 Critical Path 上的點，謂之 Critical Task：A, B, E, G, H , I，而 F 不在上術路徑上，故可 delay*
   - ⇨ *Delay 需由最晚(由後往前找 CP 最長)-最早(由前往後找 Path 最長)⟹delay*
   - ⇨ *13-8=5 天*
3. *a12 同 2.，5-4=1，可 delay 1 天*

例：承上

1. 加速哪些工作可有效縮短計畫之天數？
2. 哪些事件或工作可 delay？又 delay 幾天不影響進度？

1.

所有 CP 之上的"交集工作"即為有效加速縮短工作天數之 Task

Path 1：A B E G I ⟹ {a1, a4, a7, a10}

Path 2：A B E H I ⟹ {a1, a4, a8, a11}

⇨ *縮短 a1 或 a4 之時間可加速*

2.不在 CP 之上的工作皆可 delay：

(1) *by event*

Event	A	B	C	D	E	F	G	H	I
最早	0	6	4	7	8	8	11	15	16
最晚	0	6	5	12	8	13	11	15	16

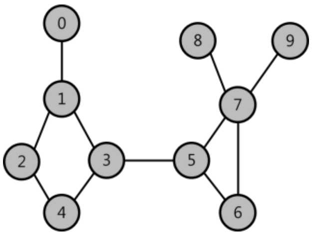
(2) *by activity*

Event	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	a11	a12
最早	0	0	0	6	4	7	8	8	8	11	15	4
最晚	0	1	5	6	5	12	8	8	13	11	15	5

Articulation Point(**切點**)

Def：在一連通圖中，若將某頂點及其連結的邊 delete，會造成圖形成為"Unconnected"，則此謂之"切點"

例：

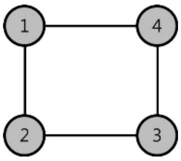


1, 3, 5, 7 為切點

Biconnected Graph

Def：不具備 Articulation Point 之 Connected Undirected Graph

例：



Note：

必然：Complete→Biconnected→Connected

不一定：Connected→Biconnected→Complete

Biconnected Component

Def：令 G=<V, E>是 Connected 之無向圖，令 G'=G 的子圖，且

1. G'是 Biconnected
2. G'是 Max component*(沒有其他子圖可包含 G')*

例：