第七章
圖形結構(Graph)

本章學習目標

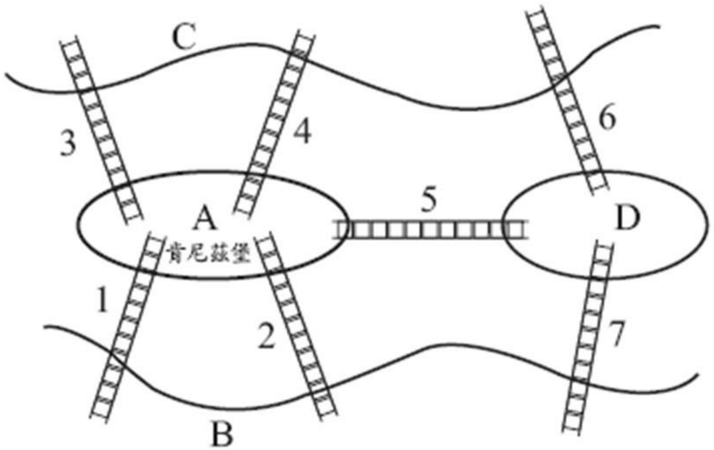
1. 讓學生了解圖形結構的相關專有名詞。
2. 讓學生了解圖形的表示方式及追蹤方法。

本章內容

7-1 圖形理論的起源

圖形的理論是起源於西元十八世紀，有一位數學家尤拉（Eular）為了解決「**肯尼茲堡橋樑**」問題，而想出的一種**資料結構**理論。

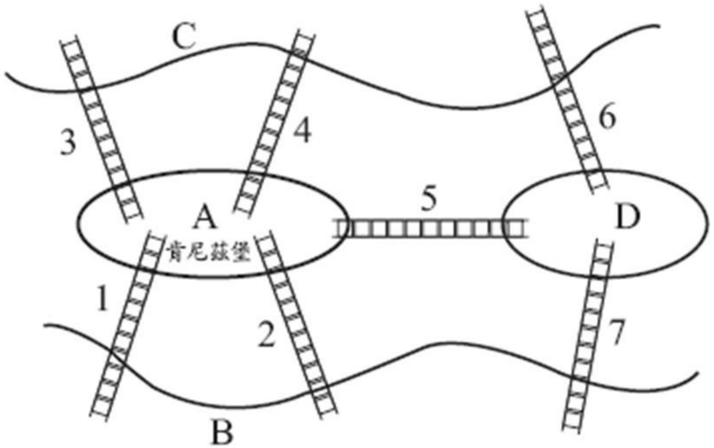
所謂的「**肯尼茲堡橋樑**」問題是：某一個**人**由某**地點**出發，最後再回到**原點**，必須要**經過每一座橋**，並且**只能經過一次**。如下圖所示：

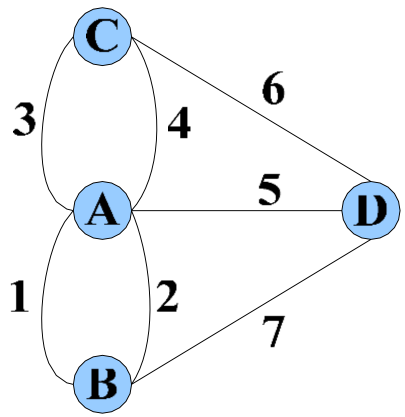


數學家<u>尤拉</u>（Eular）**當時**所使用的**方法**就是：

利用**頂點**（Vertices）：來表示**每塊土地**。如下圖(A,B,C,D塊土地)

邊（Edge）：代表每一座橋樑，如下圖(1~7條邊)





肯尼茲堡橋樑

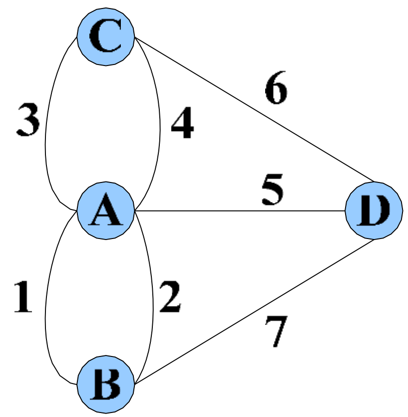
圖形理論

A,B,C,D代表土地。

❷ $\overline{AB}$、$\overline{BA}$、$\overline{AC}$、$\overline{CA}$、$\overline{CD}$、$\overline{AD}$及$\overline{BD}$ 代表橋樑

數學家尤拉（Eular）對「肯尼茲堡橋樑」問題所找出的規則就是「如果每一個頂點的分支度皆為偶數時，才能從某一個頂點出發，經過每一個邊後，再回到出發的頂點。」

而肯尼茲堡的情況為：四個頂點的分支度都是奇數（A的分支度為5，B的分支度為3，C的分支度為3，D的分支度為3），所以最後的結論：就是肯尼茲堡的人**不可能**走過所有的橋樑，到過每個地方，而後又回到肯尼茲堡。



7-1.1 尤拉循環(Eulerian cycle)

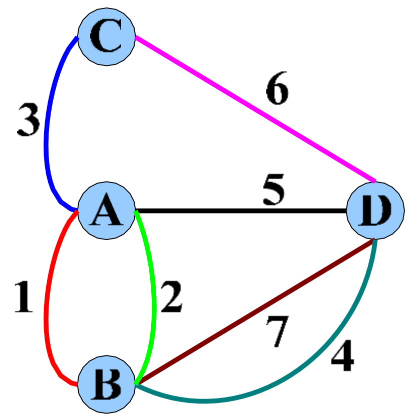
【定義】

從圖形中任何一個頂點出發，經過所有的邊，而且只能經過一次，最後再回到原出發頂點的路徑。

【條件】所有頂點的分支度必需均為偶數。

【圖解】A,B,C,D四個頂點的分支度必需均為偶數。

故下圖符合尤拉循環的條件。



【路徑】(A,B) (B,D) (D,B) (B,A) (A,D) (D,C) (C,A)

7-1.2　尤拉鏈(Eulerian chain)

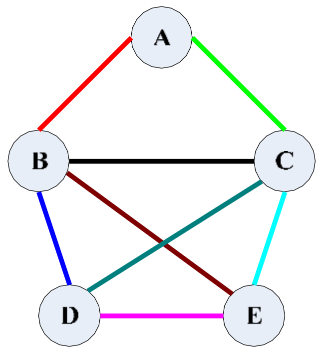
【定義】

　　從圖形中任何一個頂點出發，經過所有的邊，而且只能經過一次，最後不一定要再回到原出發頂點的路徑。

【條件】允許其中兩個頂點的分支度為奇數，其餘為偶數。

【圖解】D,E兩個頂點的分支度為奇數，其餘為偶數。

　　故下圖符合尤拉鏈的條件。



【路徑】(D,B) (B,A) (A,C) (C,B) (B,E) (E,D) (D,C) (C,E)

7-2 圖形(Graph)

【定義】

圖形(Graph)**是由頂點**(Vertices)**和邊**(Edges)**所組成，以數學式表示：**

$$G=(V,E)$$

其中，V**為所有頂點的集合**

E為所有邊的集合

例如：

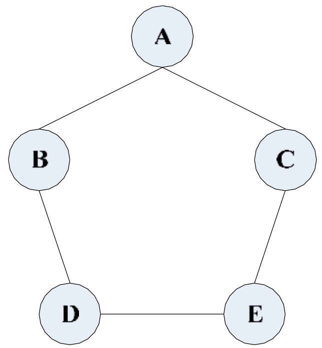
1. 表示圖形所有頂點的集合　V(G)={V1,V2,V3,…,Vm}，其中m>0

2. 表示圖形所有邊的集合　E(G)={E1,E2,E3,…,En}，其中n>0

一般而言，我們可以將圖形結構分為無向圖形(Undirected Graph)與有向圖形(Directed Graph)兩種，其說明如下：

1. 無向圖形(Undirected Graph)
   (1) 邊(Edges)是沒有方向性的。
   (2) 邊(V1,V2)與邊(V2,V1)是相同的。並且邊以( )表示。
2. 有向圖形(Directed Graph)
   (1) 邊(Edges)是有方向性的。
   (2) 邊<V1,V2>與邊<V2,V1>是不相同的。並且邊以< > 表示。
   若有一個邊為<V1,V2>，其中V1為頭(head)，V2為尾(tail)，方向為：從V1指向V2

【舉例1】假設有一個**無向圖形**，如下圖所示：



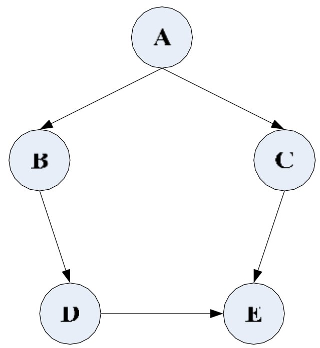
請問，此**無向圖形**的**頂點**(V)和**邊**(E)如何表示呢？

【解答】

V(G)={A,B,C,D,E}

E(G)={(A,B),(A,C),(B,A),(B,D),(C,A),(C,E),(D,B) ,(D,E) ,(E,C) ,(E,D)}

【舉例2】假設有一個**有向圖形**，如下圖所示：



請問，此**有向圖形**的**頂點**(V)和**邊**(E)如何表示呢？

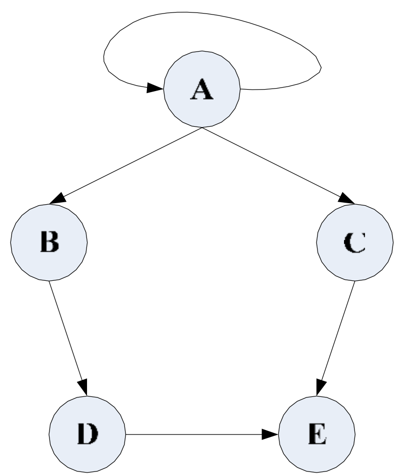
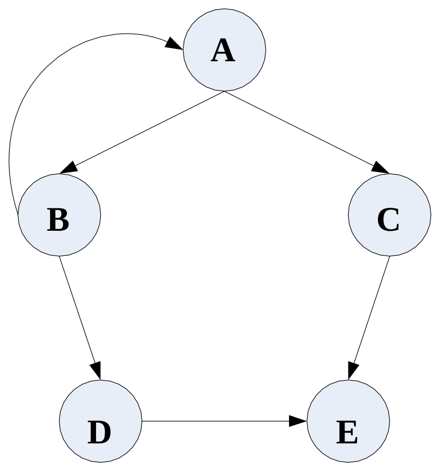
【解答】

V(G)={A,B,C,D,E}

E(G)={<A,B>,<A,C>,<B,D>,<C,E>,<D,E>}

【非圖形結構】

關於**自身迴路**(Self Loop)與**重邊**(Multi Edges)的情形，都屬於**非圖形結構**，在本章節中**不加以討論**。

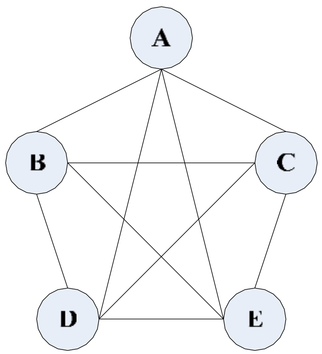
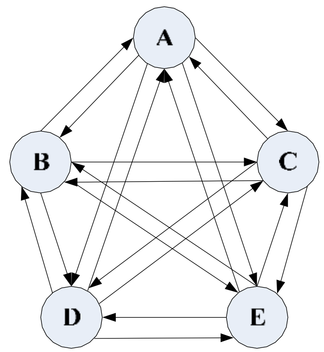
圖 (A)自身迴路	圖 (B)重邊
	

圖形結構中常用的專有名詞：

1. 完整圖形(complete graph)

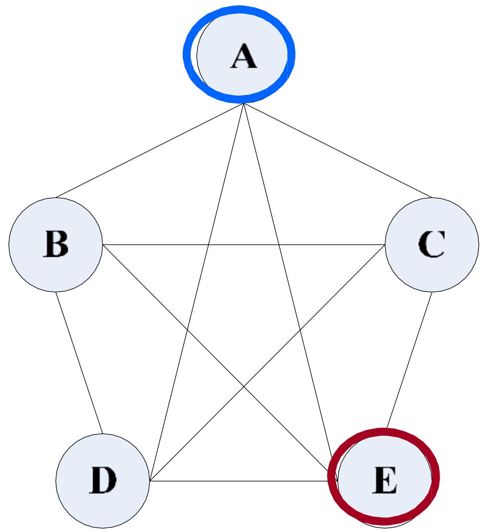
(1) 在「**無向圖形**」中，若有n **個頂點**，並且恰好有n(n-1)/2 **個邊**，則稱為「完整圖形」。

(2) 在「**有向圖形**」中，若有n **個頂點**，並且恰好有n(n-1) **個邊**，則稱為「完整圖形」。

<table>
<tr><th>圖 (A)無向圖形</th><th>圖 (B)有向圖形</th></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>N=5
E=5(5-1)/2=10</td><td>N=5
E=5(5-1)=20</td></tr>
</table>

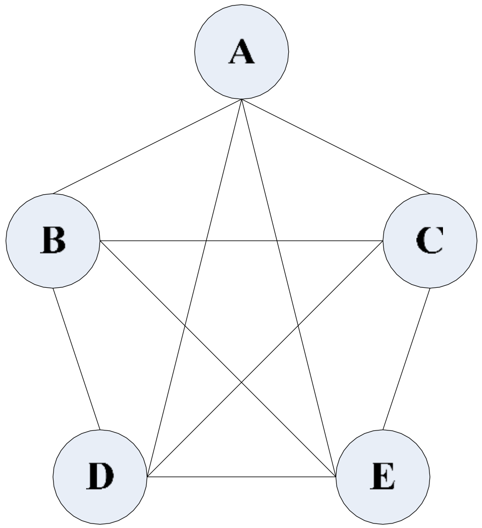
2. 路徑 (path)

在圖形 G 中，**相異兩點**間所**經過的邊**稱為**路徑**，如下圖中，A到E的路徑有{(A,B), (B,E)}、{(A,C), (C,E)}或{(A,B),(B,D),(D,E)}等路徑，它**不只有一條路徑**。

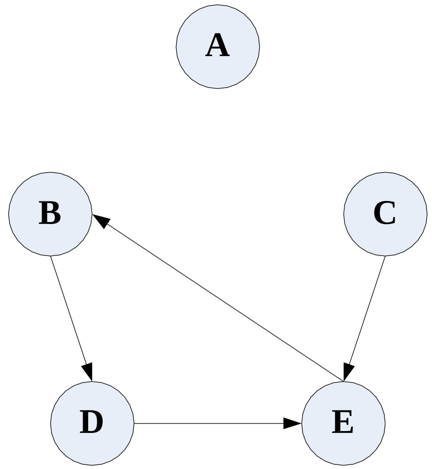


3. 簡單路徑 (simple path)

指除了起點(第一個節點)與終點(最後一個節點)外的所有節點都不相同的路徑。亦即路徑不會有循環現象。如果起點及終點為同一個點的簡單路徑稱為循環。如下圖(A)中，{(A,B),(B,D),(D,E),(E,C),(C,A)}起點及終點都是A，所以是一個循環路徑。



圖(A)

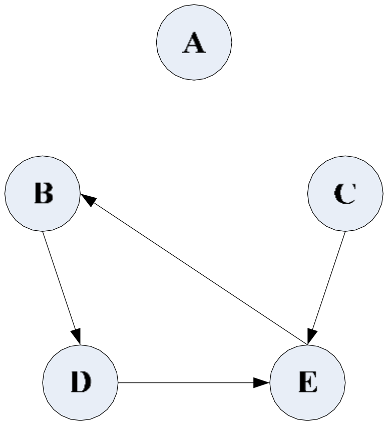


圖(B)

上圖(B)的頂點 A 為簡單路徑，路徑 C , E , B 為簡單路徑，
路徑 B , D , E ， B 就不是簡單路徑(即為循環路徑)。

4. 循環路徑 (cycle)又稱：迴圈

指起點和終點皆相同的**路徑**。例如下圖的{B , D , E ， B }起點及終點都是B 。

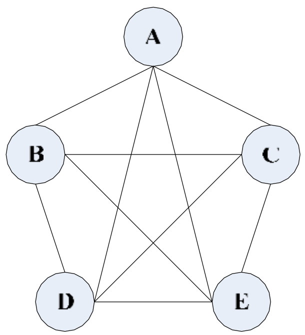
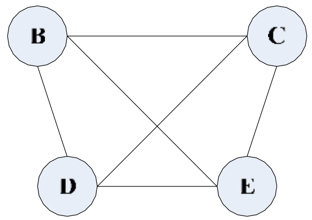


5. 子圖（Sub-graph）

假設 G'=(V', E') 並且 G=(V,E)

若$V' \subseteq V$與$E' \subseteq E$

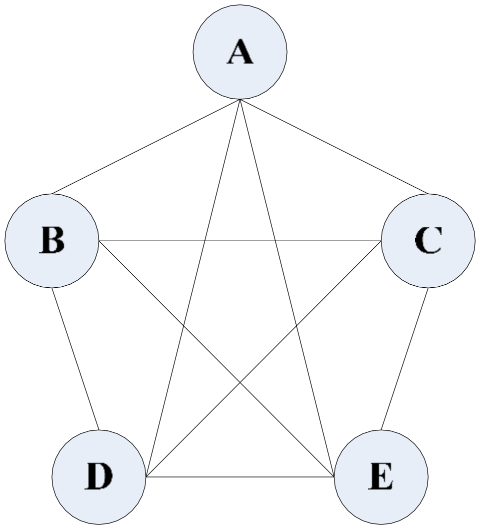
則G'=(V', E') 是 G=(V,E) 的子圖

G圖形	G’ 子圖
	

6. 連通（Connected）

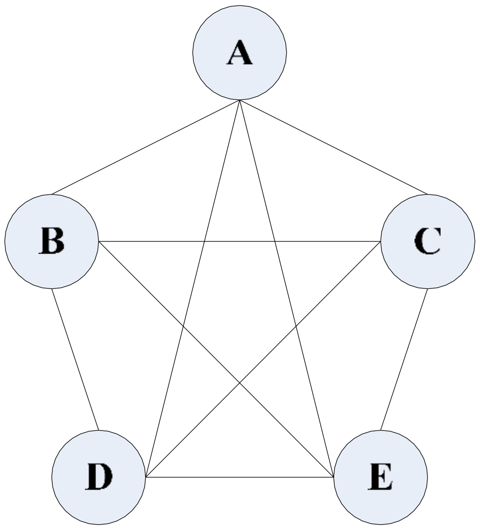
在**無向圖形**中，若**頂點**Vi到**頂點**Vj間存在**路徑**，則Vi和Vj是**相連**的。

例如：**頂點**A到**頂點**B間存在路徑，則A和B是相連的。

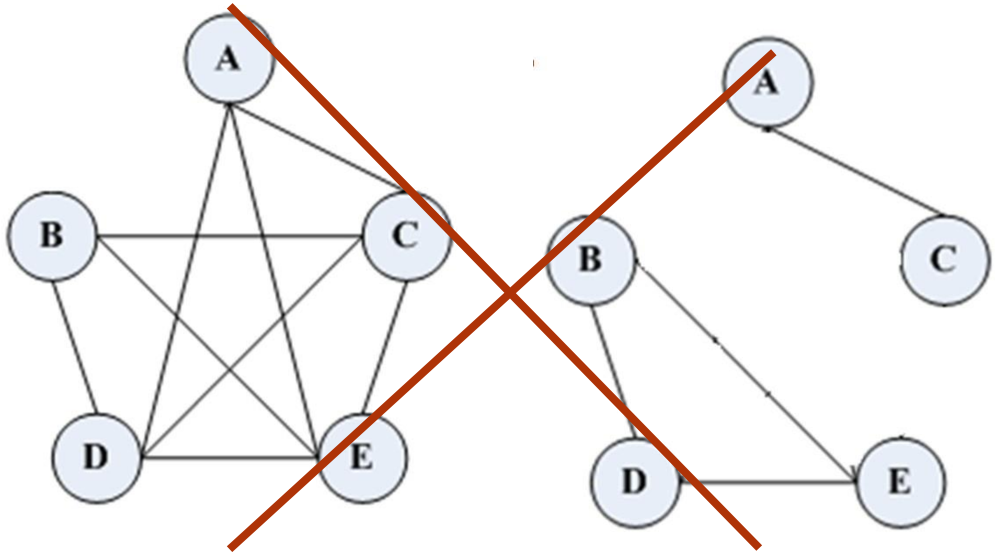


7. 相連圖形（Connected Graph）

如果圖形G中，任**兩個頂點**均為**相連**，則此圖形稱為**相連圖形**，否則稱為**非相連圖形**。



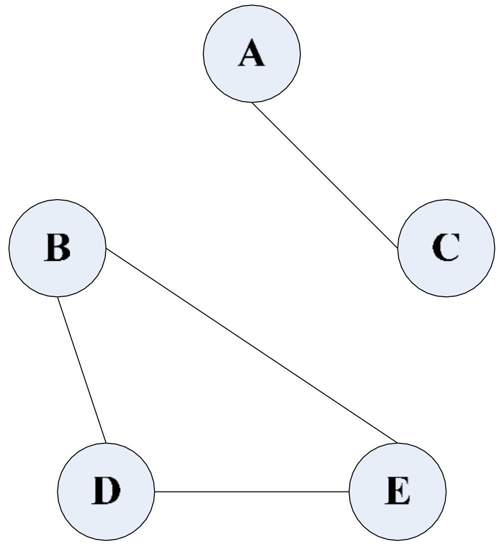
相連圖形



非相連圖形

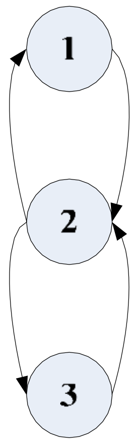
8. 連通單元（Connected Component）

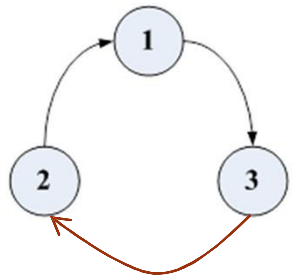
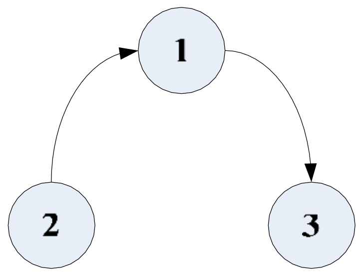
又稱單元(component)，是指該圖形中最大的連通子圖(maximal connected subgraph)，如下圖可以視為2個連通單元。



9. 緊密連通（Strongly Connected）

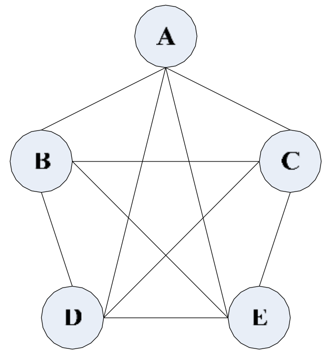
在有向圖形（Directed Graph）之中，任一個頂點具有從Vi 到Vj 的路徑並且也有一條從Vj到 Vi的路徑（Directed Graph）

<table>
<tr><th>緊密連通</th><th>不具有緊密連通</th></tr>
<tr><td>


</td><td>
</td></tr>
</table>

10. 相鄰（Adjacent）

(1) 無向圖G=(V,E)

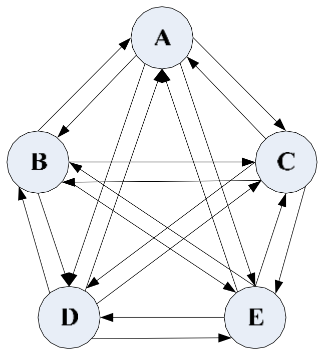


❶A,B∈V

❷(A,B)∈E，其中A,B代表相異兩個頂點

則稱頂點A 與頂點B 相鄰

(2) 有向圖G=(V,E)



❶A,B∈V

❷<A,B>∈E

則稱A 相鄰至B

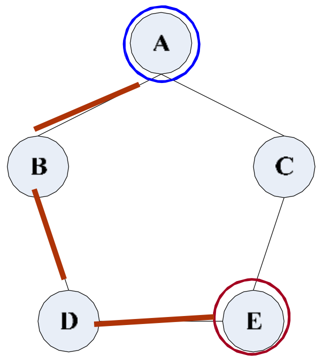
並且稱B 相鄰自A

11. 路徑長度（Path Length）

路徑長度k 代表路徑上的邊（Edge）的數量。

例如：在下圖中，頂點A到頂點E的路徑為：(A,B) (B,D) (D,E)，

則路徑長度k=3



12.分支度（Degree）

在圖形結構中的分支度，必須要探討兩種情況：

(1)無向圖

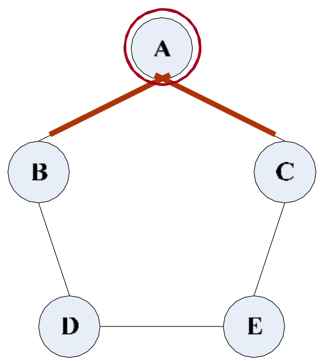
(2)有向圖

其說明如下：

(1)無向圖G=(V,E)

頂點A 的分支度= 附著於A 的邊總數，

如下圖中A頂點的分支度為2 。



12.分支度（Degree）(續)

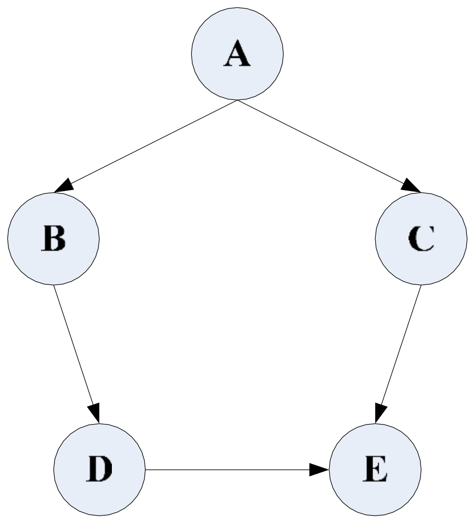
(2)有向圖G=(V,E)

❶入分支度(in - degree)：指某一個頂點v擁有「箭頭」的邊數。

如下圖中，A 頂點的入分支度為0，而E頂點的入分支度為2 。

❷出分支度(out - degree)：剛好和入分支度相反，也就是某一頂點v擁有「尾」的邊數。

如下圖中，A頂點的出分支度為2，而E頂點的出分支度為0 。



7-3 圖形的表示法

基本上，圖形的表示法，常見兩種方法：

一. 相鄰矩陣(Adjacency Matrix)

二. 相鄰串列(Adjacency Lists)

7-3.1 相鄰矩陣(Adjacency Matrix)

【定義】

若圖形G = (V , E) 是具有n 個頂點的圖形，並且n >= 1時，則要表示圖形G 的相鄰矩陣，我們可以利用一個n × n 的二維陣列來表示，稱其為相鄰矩陣(adjacency matrix) 。

【特性】

1. 矩陣A[i][j] = 0 表示邊E(i,j)不存在

2. 矩陣A[i][j] = 1 表示邊E(i,j)存在

3. 無向圖的相鄰矩陣以對角線對稱，亦即A[i][j]=A[j][i]

4. G=(V,E),|V|=n, 頂點i∊V 的分支度（Degree）

(1) 無向圖

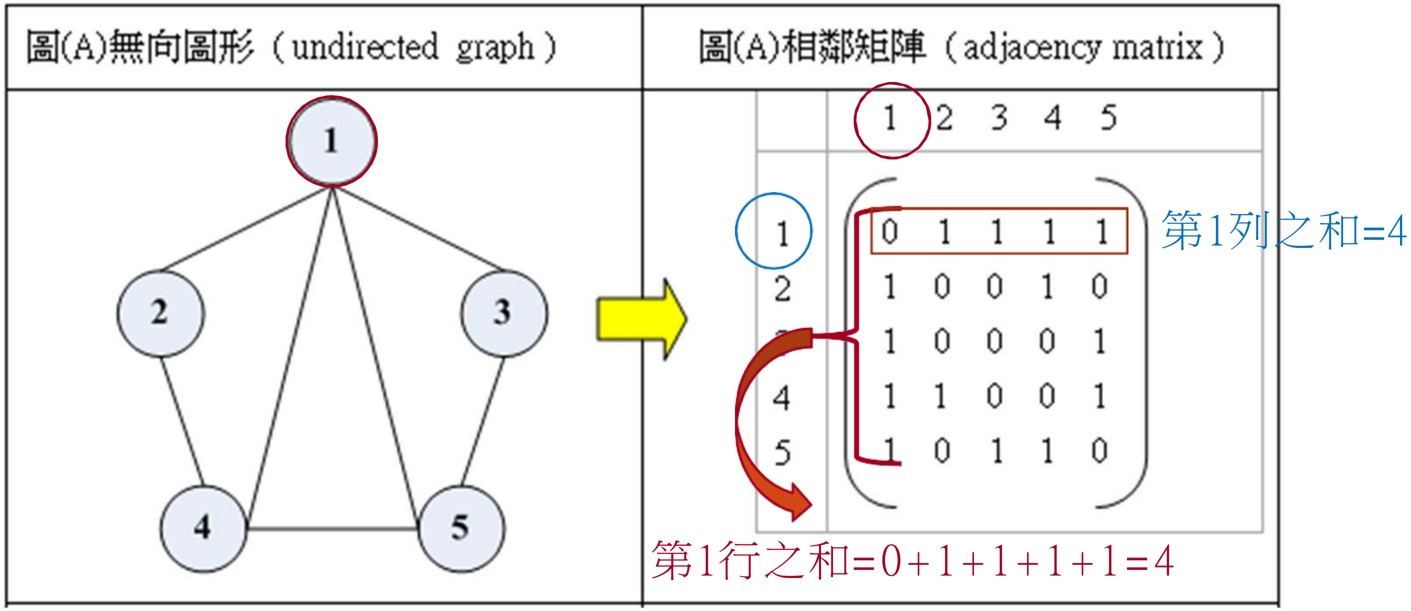
$$d(i) = \sum_{j=1}^{n} A[i][j]$$

，其中i代表某一**頂點**

(2) 有向圖

$$d_{out}(i) = \sum_{j=1}^{n} A[i][j]$$

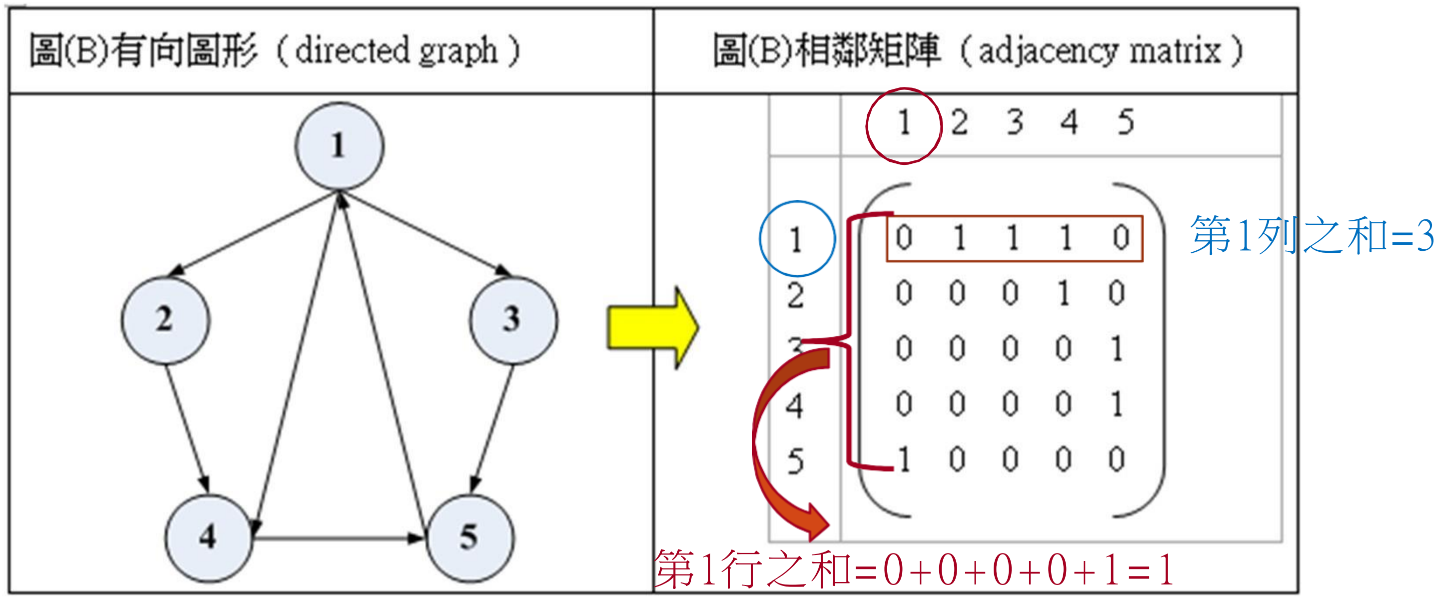
$$d_{in}(i) = \sum_{j=1}^{n} A[j][i]$$



在上圖(A)無向圖形的相鄰矩陣中，若A[i][j]=1時，則代表圖形G中有一條邊(Vi,Vj)存在。反之，若A[i][j]=0時，則代表圖形G中沒有一條邊(Vi,Vj)不存在。

對無向圖形而言，任一頂點i的分支度是第i列或第i行之和。

例如：在上圖(A)無向圖形的相鄰矩陣中，頂點1的分支度為4 。

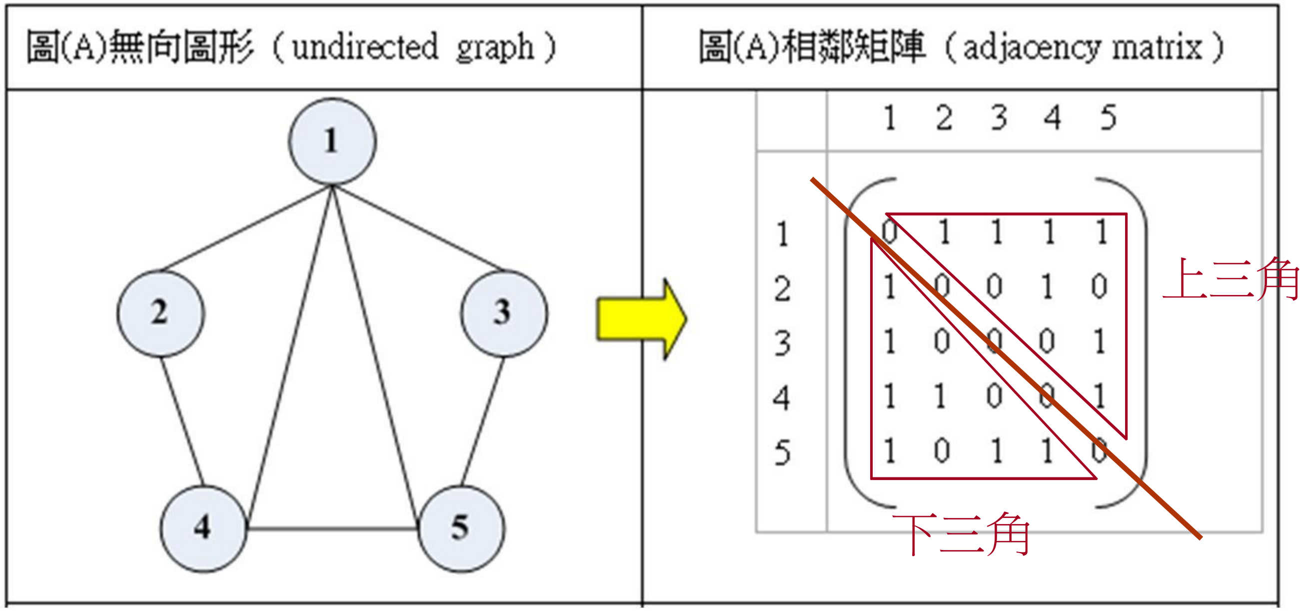


在上圖(B)有向圖形的相鄰矩陣中，若A[i][j]=1時，則代表圖形G中有一條邊<Vi,Vj>存在。反之，若A[i][j]=0時，則代表圖形G中沒有一條邊<Vi,Vj>不存在。

對有向圖形而言，各「列之和」就是「出支度」，而各「行之和」是「入支度」。例如：在上圖(B)有向圖形的相鄰矩陣中，頂點1的出支度為3，而入支度為1。

特別注意一點就是，在無向圖形中的相鄰矩陣就是一種「對稱矩陣」，

因此，為了節省記憶體空間，我們可以只儲存上三角或下三角即可。



＊有向圖形的相鄰矩陣，則不一定會是「對稱矩陣」。

7-3.2 相鄰串列(Adjacency Lists)

【定義】

假設圖形G=(V,E)包含n 個頂點(n≧1)時，則我們可以使用n 個鏈結串列來存放該圖形，每個鏈結串列分別代表一個頂點及其相鄰的頂點。而以「串列結構」來表示圖形，它有點類似相鄰矩陣，不過忽略掉矩陣中0的部份，直接把1的部份放入節點裡。

【節點結構】

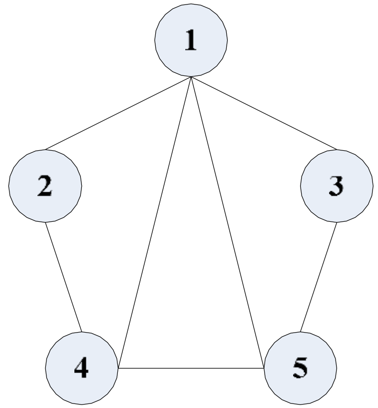
在相鄰串列中所使用的節點結構是由2個欄位所組成，分別為頂點欄位及指標欄位。如下圖所示：

頂點欄位	指標欄位
Vertex	**Link**

【節點結構之定義】

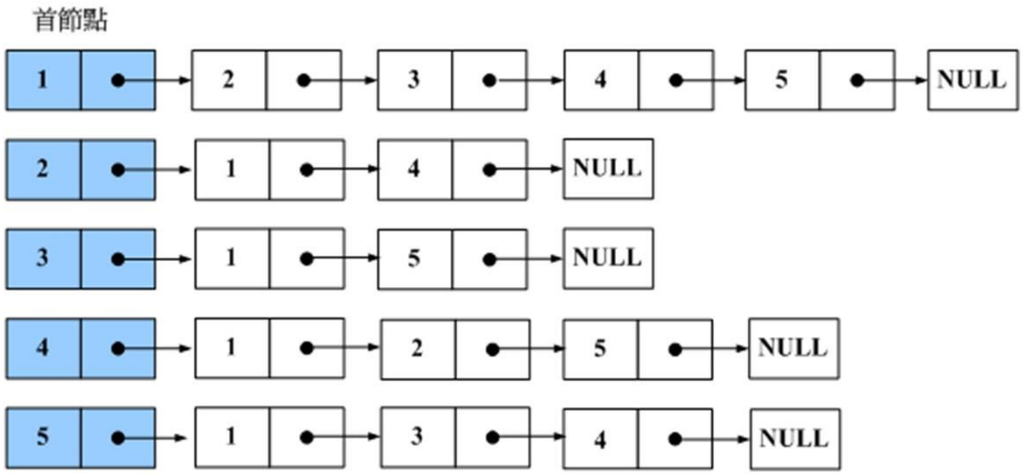
```
typedef struct gnode    //定義相鄰串列的節點結構
{
    int Vertex ;            //宣告頂點欄位
    struct gnode *Link ;    //宣告指標指向下一個節點結構之欄位
} GNODE;
```

【實例1】請將下面的無向圖形(undirected graph)轉成相鄰串列。



無向圖形

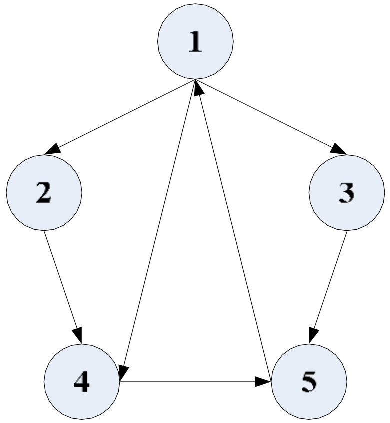
【解答】



說明：在無向圖中，5個頂點7條邊共需5個串列首節點及14個節點，

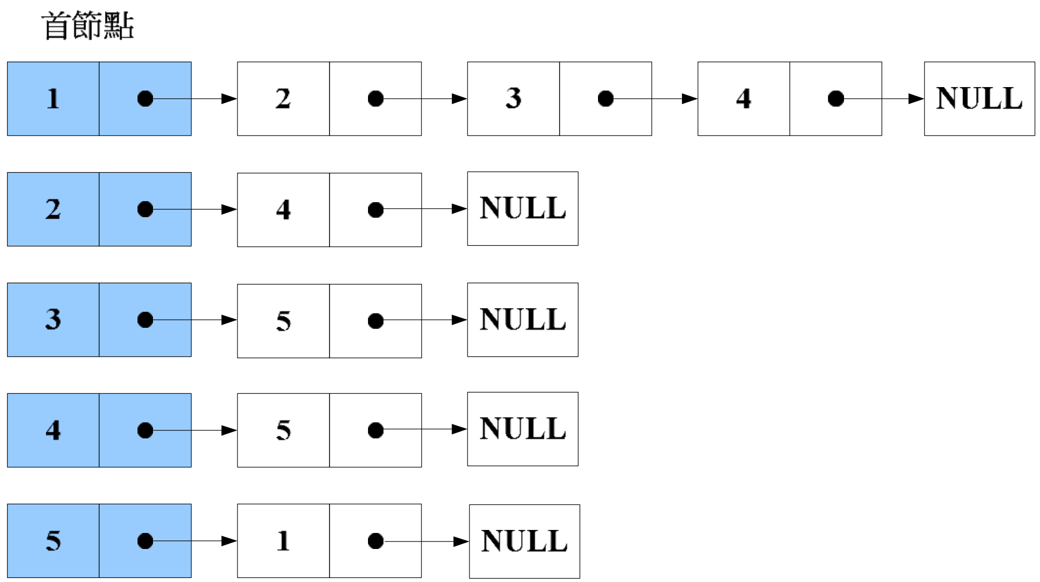
因此，在無向圖形中，節點數目為邊數的2倍。

【實例2】請將下面的有向圖形(directed graph)轉成相鄰串列。



有向圖形

【解答】



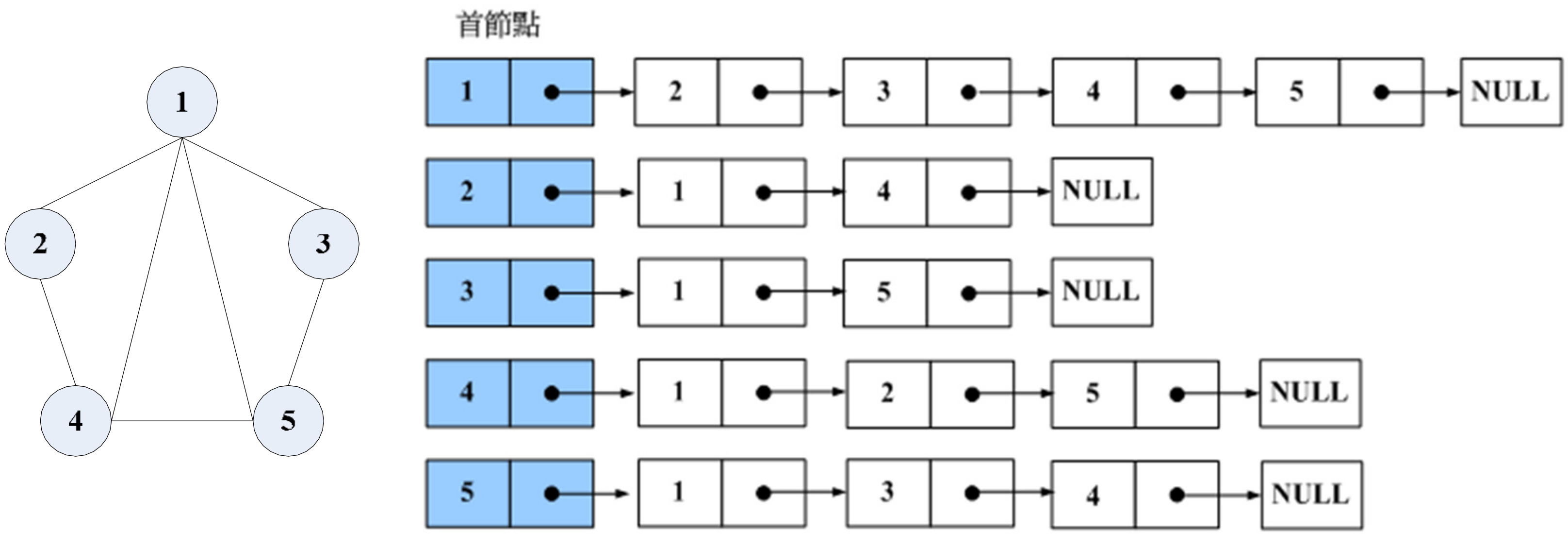
說明：在有向圖中， 5個頂點7條邊共需5個串列首節點及7個節點，

因此，在有向圖形中，節點數目恰等於邊數。

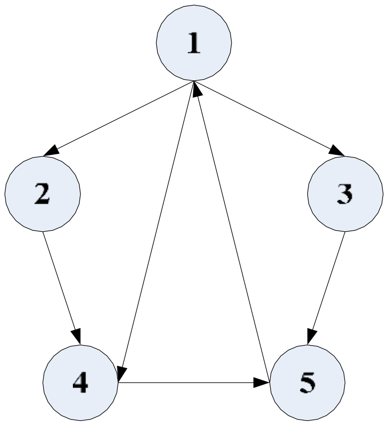
【分析】

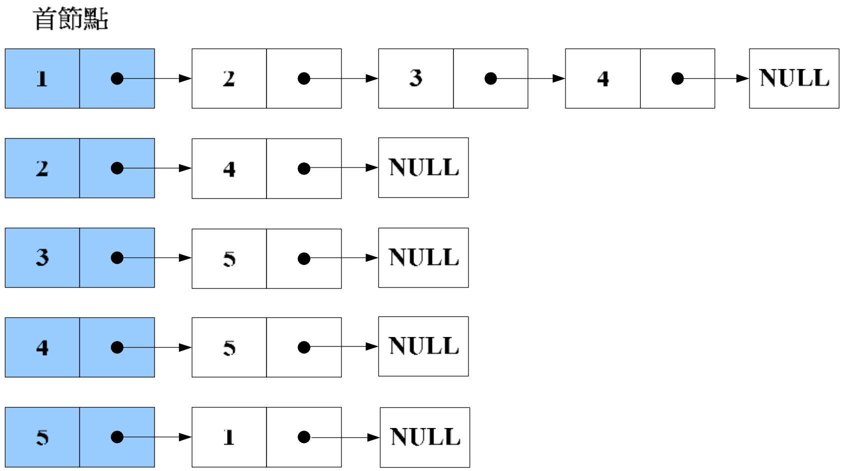
1. 每一個頂點使用一個串列。

2. 在無向圖中，n個頂點e條邊共需n 個串列首節點及2*e個節點



3.有向圖中，n個頂點e條邊共需n個串列首節點及e個節點。

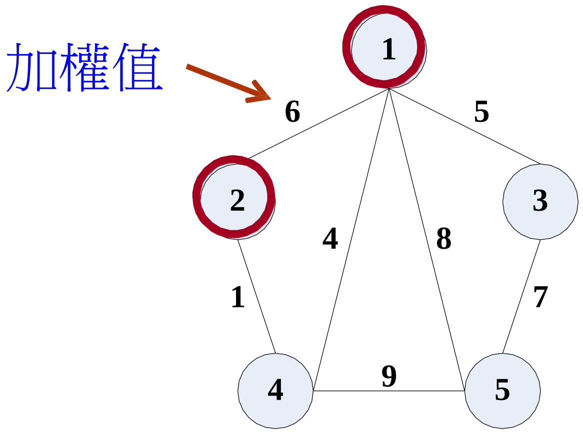




7-4 加權圖形

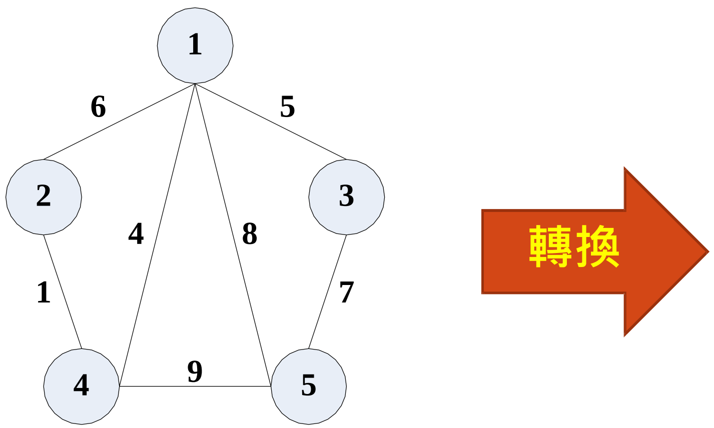
在上面的例子中，圖形上的每一個邊都沒有任何的權重，亦即任一頂點到其他頂點之間的關係強度都是相同的。但是，當我們要表示的資料與資料之間的關係強度是有不同時，那就必須要利用到「加權圖形」來呈現。

何謂「加權圖形」呢？是指在圖形上的每一個邊上都給予一個權重值(weight)，此權重值可以用來表示距離、成本、時間或關係強度等等，如下圖中，頂點1與頂點2之間邊的加權值為6 。



7-4.1 加權圖形的相鄰矩陣之表示法

假設加權圖形利用相鄰矩陣來表示時，這與前面介紹的相鄰矩陣略有一些些不同，前面所介紹的圖形表示法是以兩個頂點之間若有相連，則以” 1” 來表示，若無，則以” 0” 。而在加權圖形中相異兩個頂點若有相連，則以加權值來表示之，若無，則以” ∞” 來表示。以圖的加權圖形為例，將它轉換成加權圖形的相鄰矩陣之表示法：



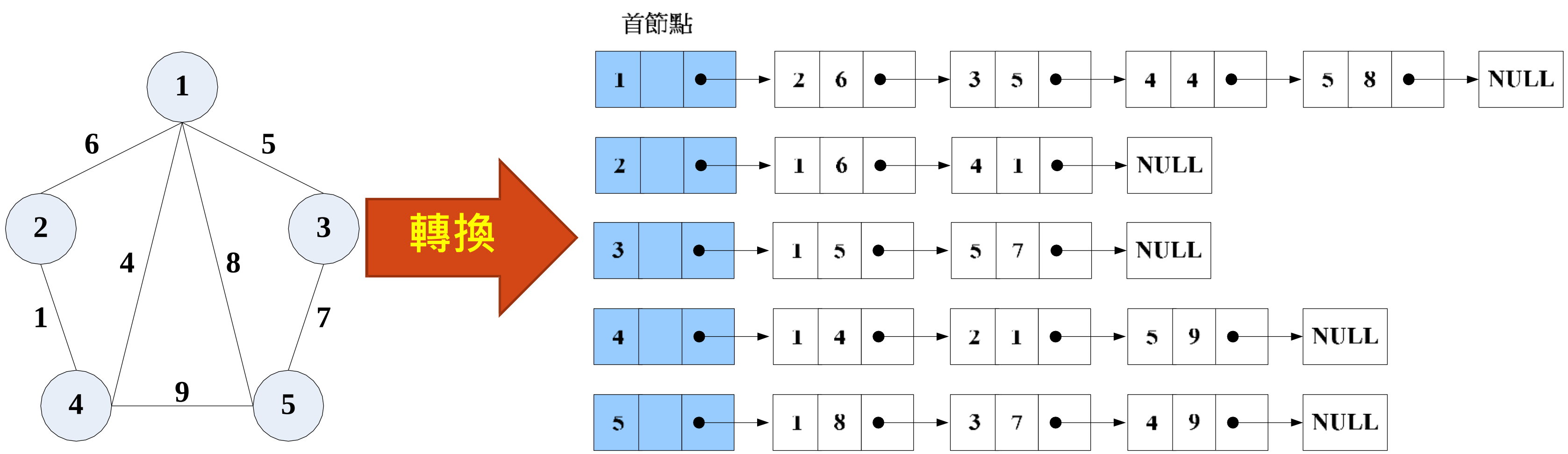
$$\begin{array}{c|c} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \hline \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 6 & 5 & 4 & 8 \\ 6 & 0 & \infty & 1 & \infty \\ 5 & \infty & 0 & \infty & 7 \\ 4 & 1 & \infty & 0 & 9 \\ 8 & \infty & 7 & 9 & 0 \end{bmatrix} \end{array}$$

7-4.2 加權圖形的相鄰串列之表示法

在利用相鄰串列來表示加權圖時，其相鄰串列的結構必須要再加上一個「權重欄位」來存放加權值。如下圖所示：

節點	權重欄位	指標
Vertex	Weight	Link

例如：將以下的加權圖形轉換成加權圖的相鄰串列之表示法：



7-5 圖形的走訪方式

樹有前序法、中序法和後序法三種走訪(追蹤)方式；

圖形的走訪和樹的走訪概念相同，都是要能夠走訪到所有頂點。

圖形的走訪方法有兩種：

1. 深度優先搜尋法（Depth-First-Search，DFS）
2. 廣度優先搜尋法（Breadth-First-Search，BFS）

7-5.1 深度優先搜尋法 (Depth-First Search；DFS)

【定義】

1. 深度優先搜尋法是利用「堆疊(Stack)」來處理。
2. 以先深後廣的方式是從圖形的某一頂點開始走訪，被拜訪過的頂點就會被標示已拜訪的記號。接著走訪此一頂點的所有相鄰並且未拜訪過的頂點中的任意一個頂點，並標示已拜訪的記號，再以該點為新的起點繼續進行先深後廣的搜尋。

【作法】

(1) 先拜訪起始頂點V。

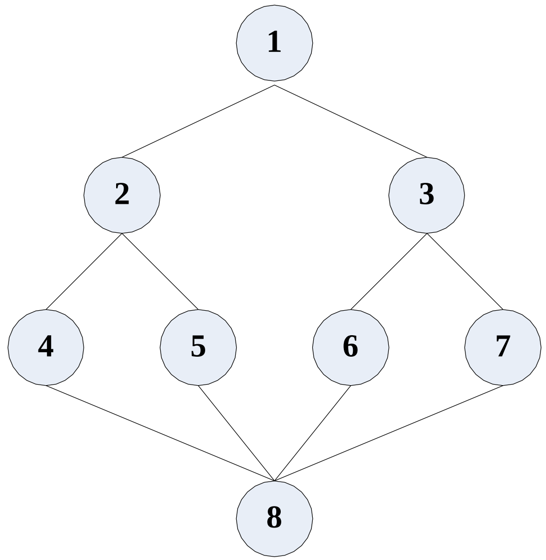
(2) 接著再選擇與頂點V相鄰而且尚未被拜訪過的頂點W，以W 為起始點來做深入搜尋。

(3) 若有一頂點其相鄰的頂點皆被拜訪過時，就退回到最近曾拜訪過之頂點，繼續執行深入搜尋。

(4) 若從任何已走過的頂點，皆無法再找到未被走過的相鄰頂點時，此搜尋就結束了。

【以堆疊(Stack) 來實作】

假設現在有一個圖形，如下所示：

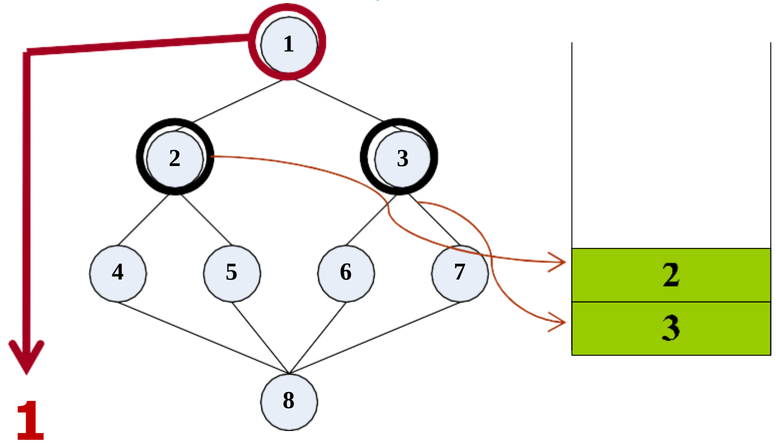


上圖經深度優先搜尋法處理後，其輸出為：1 , 2 , 4 , 8 , 5 , 6 , 3 , 7

以堆疊的處理情形如下所示：

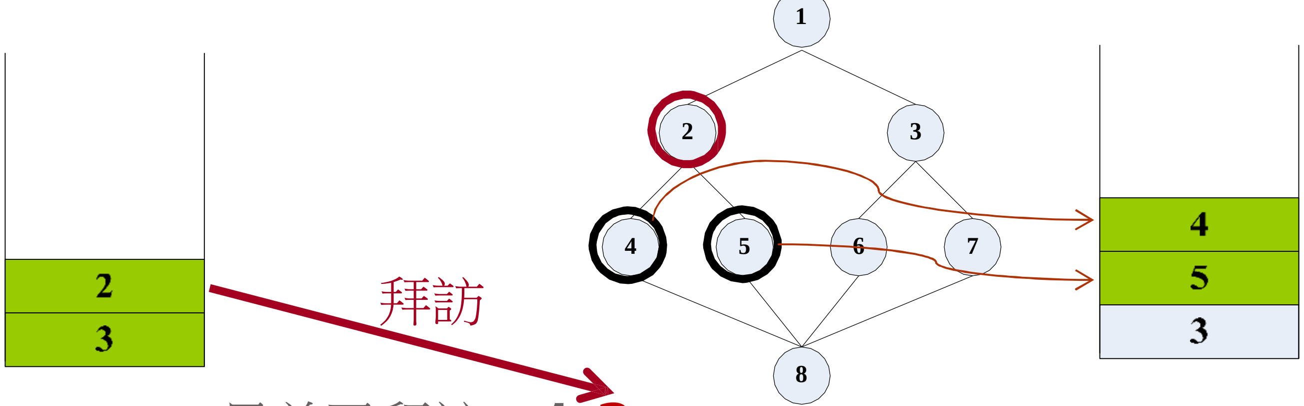
【以堆疊(Stack)來實作】(續)

步驟**1**：拜訪❶，將相鄰的❷及❸放入堆疊中。



目前已拜訪：**1**

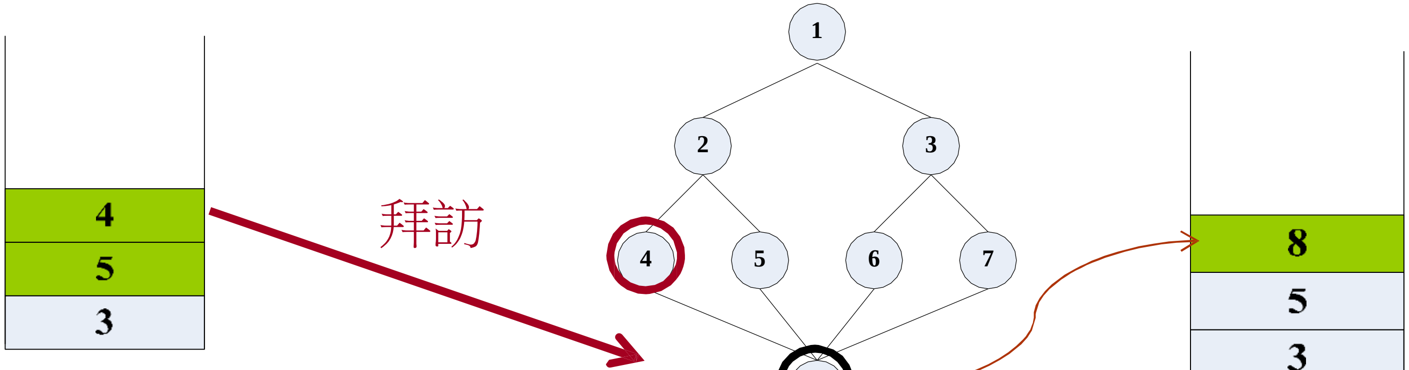
步驟**2**：拜訪❷，再將與❷相鄰且未拜訪過的❹及❺放入堆疊中。



目前已拜訪：**1,2**

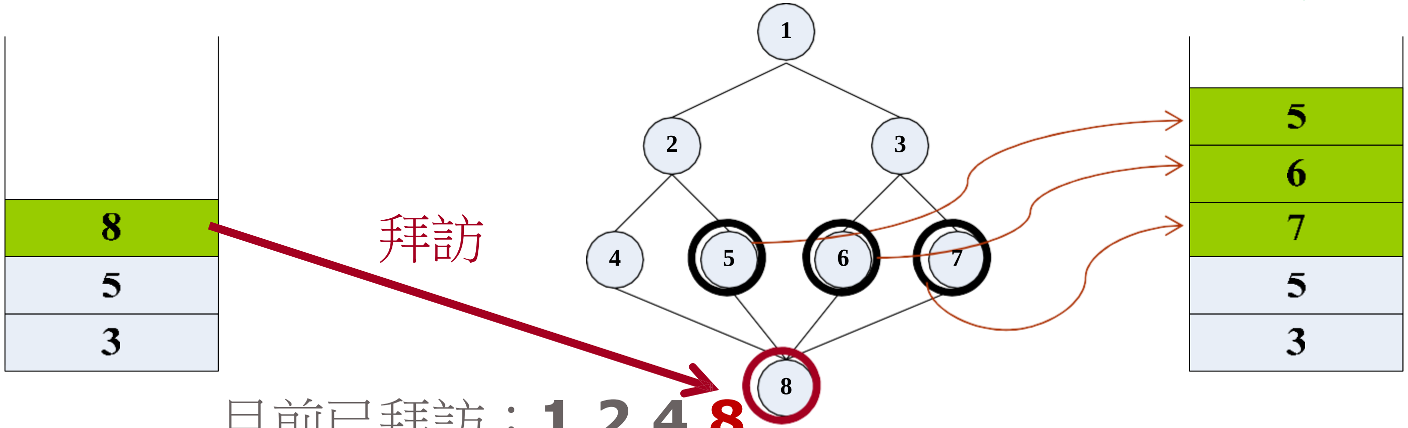
【以堆疊 (Stack) 來實作】(續)

步驟**3**：拜訪❹，再將與❹相鄰且未拜訪過的❽放入堆疊中。



目前已拜訪：**1,2,4**

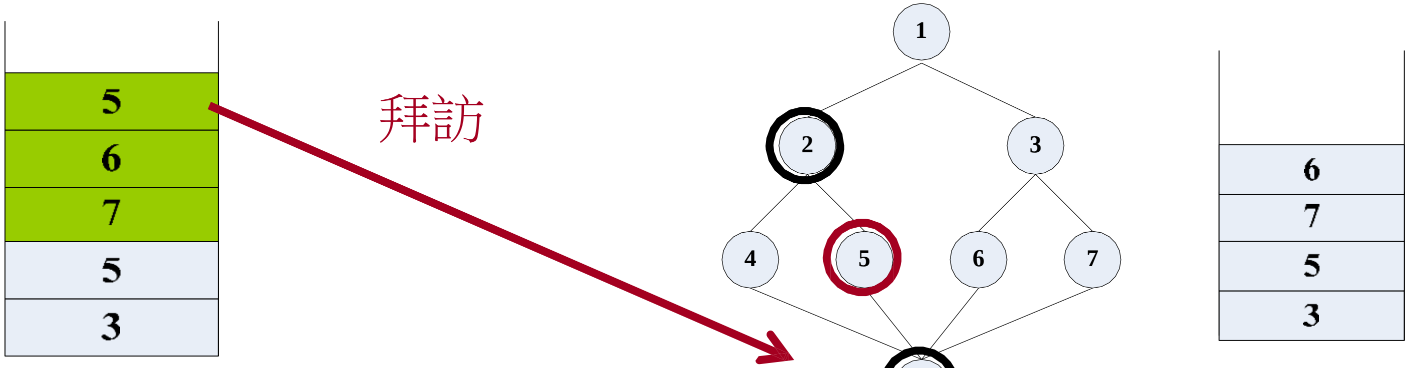
步驟**4**：拜訪❽，再將與❽相鄰且未拜訪過的❺❻❼放入堆疊中。



目前已拜訪：**1,2,4,8**

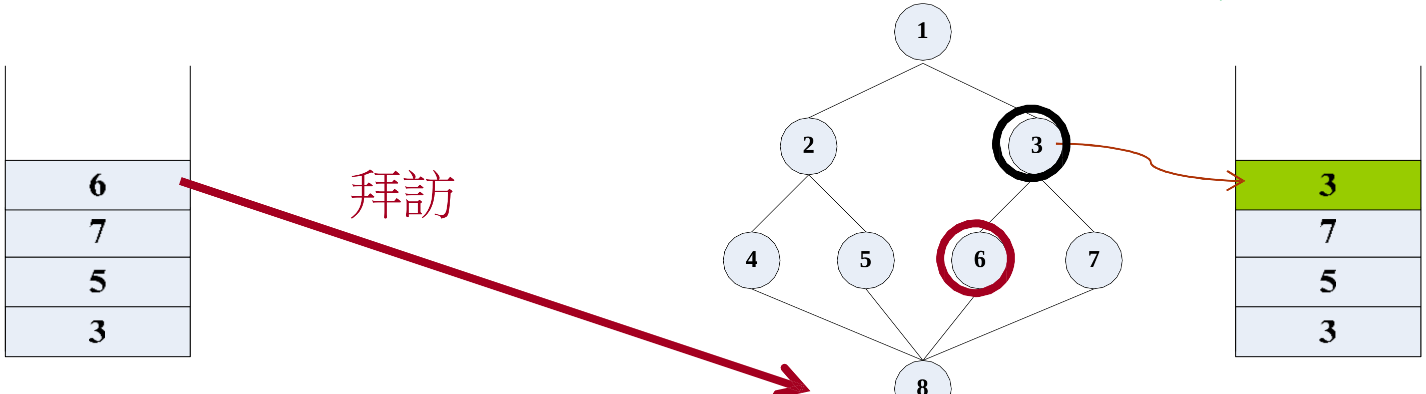
【以堆疊 (Stack) 來實作】(續)

步驟5：拜訪❺(因為相鄰❷及❽都已拜訪過了)。



目前已拜訪：1,2,4,8,5

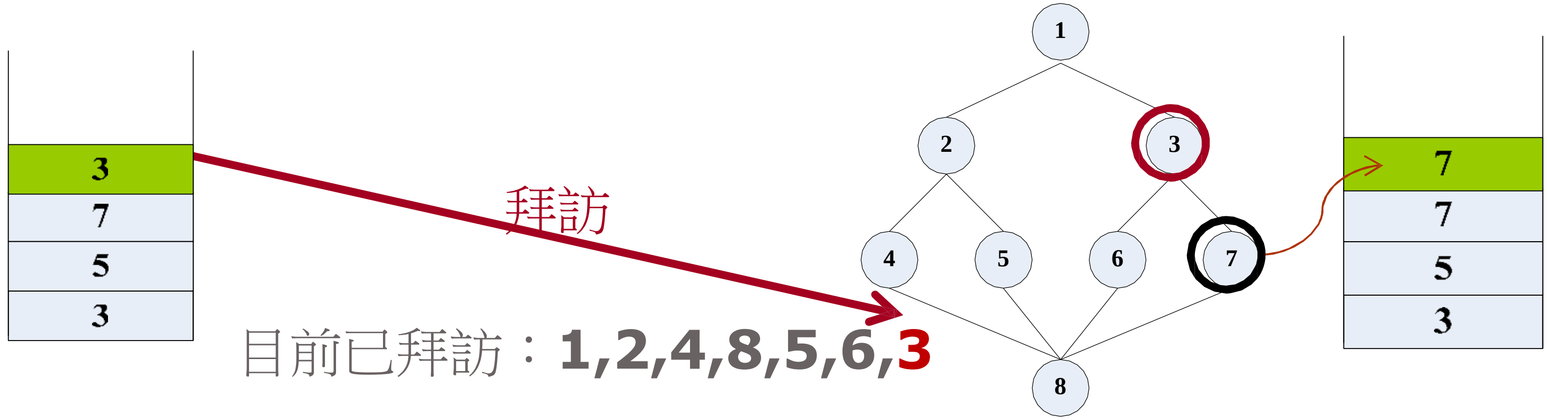
步驟6：拜訪❻，再將與❻相鄰且未拜訪過的❸放入堆疊中。



目前已拜訪：1,2,4,8,5,6

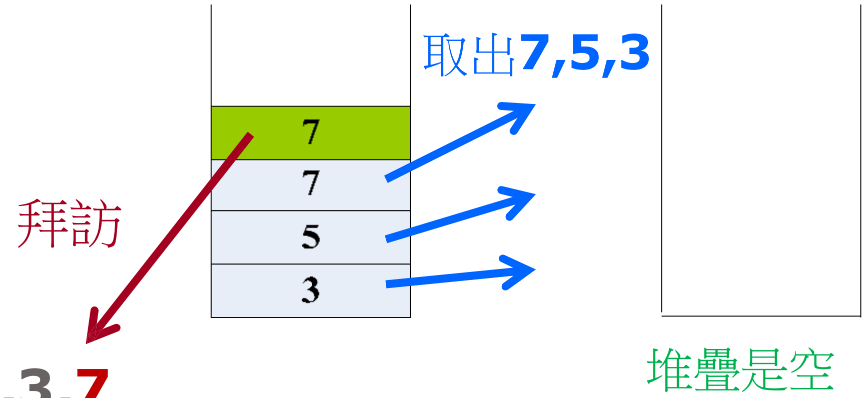
【以堆疊(Stack)來實作】(續)

步驟**7**：拜訪❸，再將與❸相鄰且未拜訪過的❼放入堆疊中。



目前已拜訪：**1,2,4,8,5,6,3**

步驟**8**：拜訪❼之後，由於皆已拜訪過了，因此，最後序由堆疊中取出**7,5,3**，讓堆疊是空的，表示搜尋結束。



目前已拜訪： **1,2,4,8,5,6,3,7**

所以，深度優先搜尋法的走訪順序為： **1,2,4,8,5,6,3,7**

7-5.2 廣度優先搜尋法
(Breadth First Search ; BFS)

【定義】

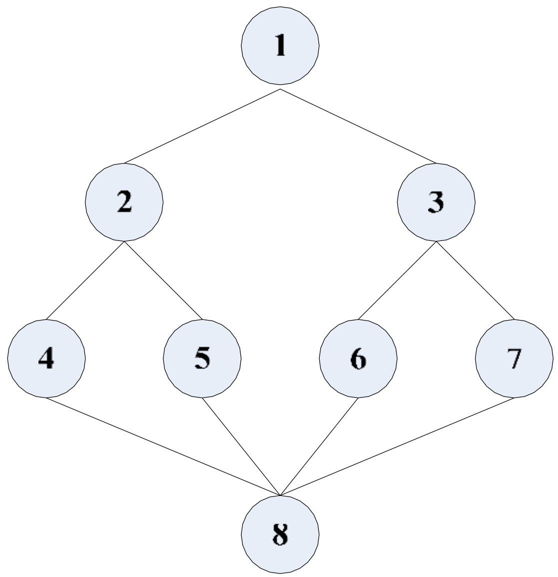
1. 廣度優先搜尋法以「佇列(Queue)」及「遞迴」技巧來走訪。
2. 以先廣後深的方式是從圖形的某一頂點開始走訪，被拜訪過的頂點就會被標示已拜訪的記號。接著走訪此一頂點的所有相鄰並且尚未拜訪過的頂點中的任意一個頂點，並標示已拜訪的記號，再以該點為新的起點繼續進行先廣後深的搜尋。

【作法】

1. 首先準備一個佇列為Qu
2. 再將起始頂點v 加入到Qu 之中（亦即進行Enqueue動作）
3. 如果 **Qu** 不為空，則執行下列步驟，否則跳到步驟 **4**：
   1. 從 Qu 中取出一頂點v（亦即進行Dequeue動作）
   2. 將所有與頂點v相鄰串列上所有尚未被拜訪過的頂點加入到Qu 之中(亦即進行Enqueue動作）
   3. 回到步驟 3
4. 結束

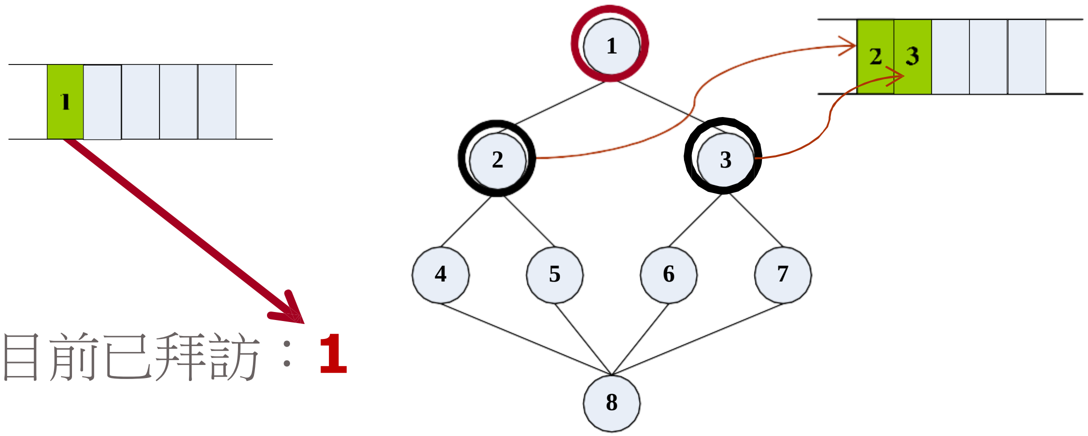
【以佇列(Queue)實作】

假設現在有一個圖形，如下圖所示：

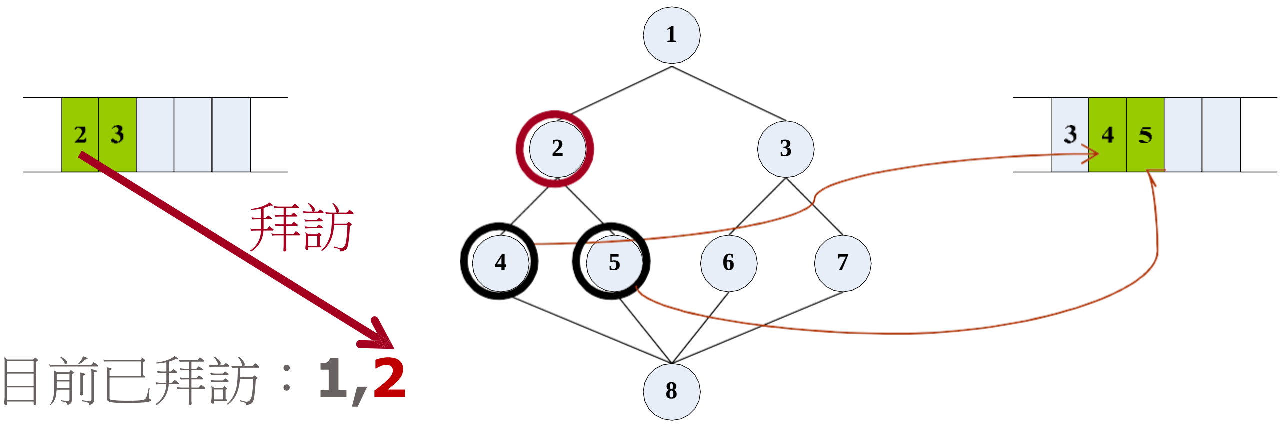


以佇列的處理情形如下所示：

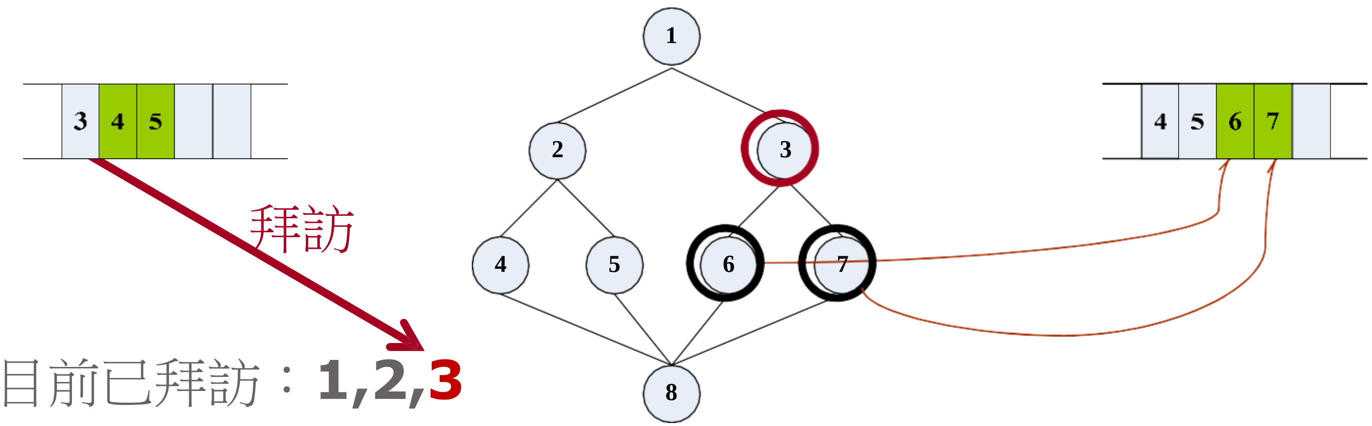
步驟1：首先拜訪「起始頂點1」加入到佇列中，並輸出❶，將相鄰的❷及❸放入佇列中。



步驟2：拜訪❷，再將與❷相鄰且未拜訪過的❹及❺放入佇列中。

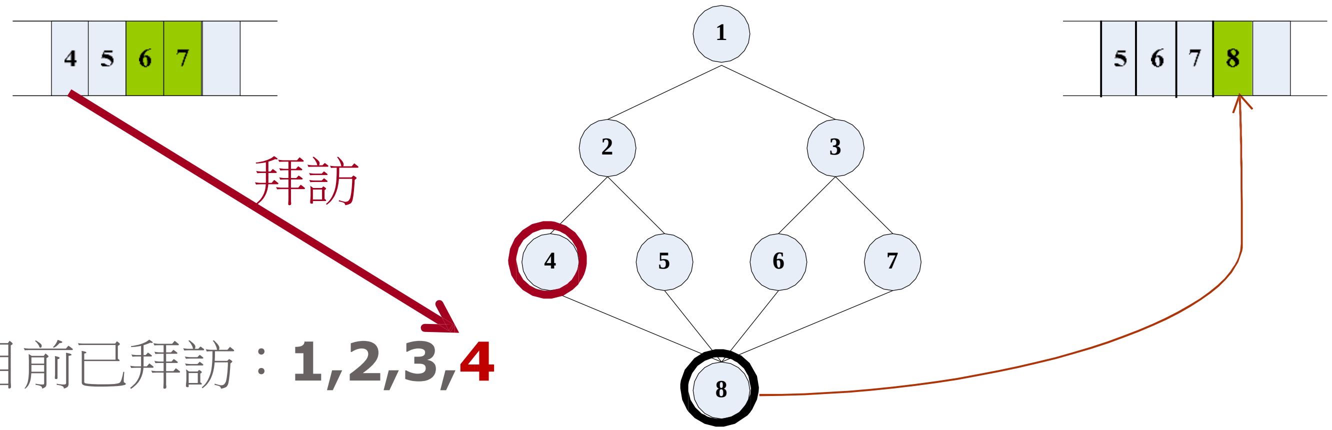


步驟3：拜訪❸，再將與❸相鄰且未拜訪過的❻及❼放入佇列中。



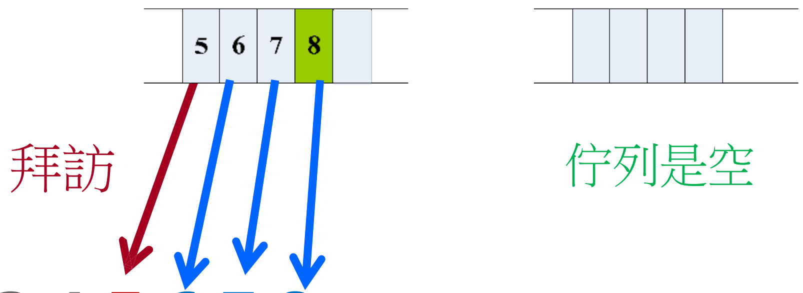
目前已拜訪：**1,2,3**

步驟4：拜訪❹，再將與❹相鄰且未拜訪過的❽放入佇列中。



目前已拜訪：**1,2,3,4**

步驟5：拜訪❺之後，此時你會發現全部頂點都被拜訪過了，因此，最後序由佇列中取出**6,7,8**，讓佇列是空的，表示搜尋結束。



目前已拜訪：**1,2,3,4,5,6,7,8**

所以，先廣後深的走訪順序為：❶❷❸❹❺❻❼❽

7-5.3 DFS與BFS比較

1. 深度優先搜尋法（DFS）以深度（路徑長度）優先，
   可以用「遞迴」和「堆疊」控制要走訪的頂點。
2. 廣度優先搜尋法（BFS）以廣度（分支度）優先，
   可以用「遞迴」和「佇列」來控制要走訪的頂點。

7-5.4 DFS與BFS應用

圖形的走訪可應用於下列幾項：

1. 找出一個無向圖形的擴張樹（spanning tree）
2. 判斷無向圖形是否為一個相連圖形（connected graph）
3. 找出一個無向圖形的相連子圖

7-6 擴張樹(Spanning Tree)

由前一節介紹的深度優先法和廣度優先法得知，如果是一個有n 個頂點的相連圖形，經由這兩種演算法走訪的結果，會得到用最少的邊來連結所有的頂點，且不會形成迴路，這樣的子圖是一種樹狀結構，也就是任何兩個頂點之間的路徑唯一，這種可連結所有頂點且路徑唯一的樹狀結構稱為**擴張樹（spanning tree）**或稱生成樹、擴展樹。

【應用方面】

1.興建的道路

2.網路線的配置

【定義】

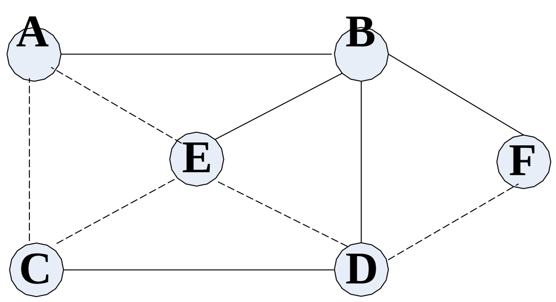
假設G = (V,E)是一個圖形，而 S = (V, T) 是 G 的擴張樹。其中T 是追蹤時所拜訪過的邊，此時擴張樹具有下列幾點特性：

1. E = T + K ，其中K 表示追蹤後所未被拜訪過的邊。

   如下圖中的「實線」+「虛線」所示。

2. 圖形上的任何兩頂點V1及V2 ，在擴張樹S中有唯一的邊。

3. 加入K 中任何一個邊於S 中，會造成循環。如下圖所示：

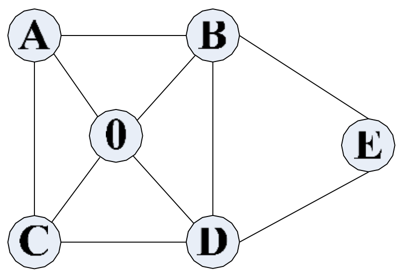
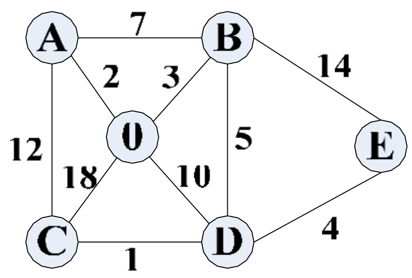


實線：T是追蹤時所拜訪過的邊

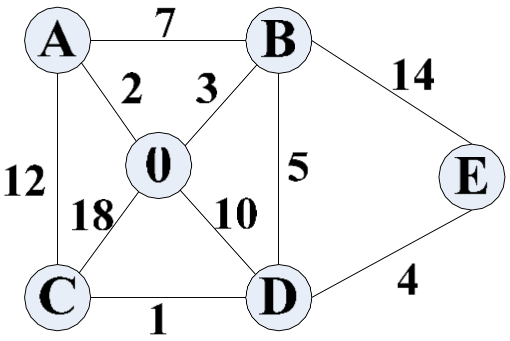
虛線：K表示追蹤後所未被拜訪過的邊

7-7 最小成本擴張樹
(Minimum Cost Spanning Tree)

擴張樹在實際的應用上不止是找出頂點和邊而已，如果一個相連圖形的邊加上權重值（weight）時，此時我們就可以利用邊來代表成本、距離或關係強度，並且希望所產生的擴張樹之所有邊的權重值加總為最小，如果具有這樣性質的擴張樹，就可以稱為最小成本擴張樹（minimum-cost spanning tree）。

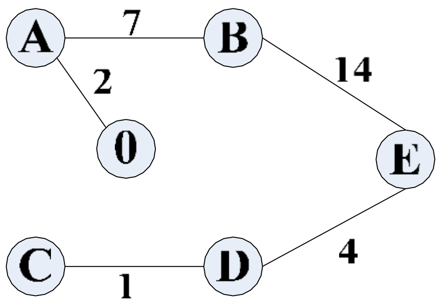
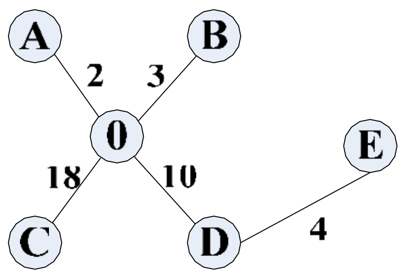
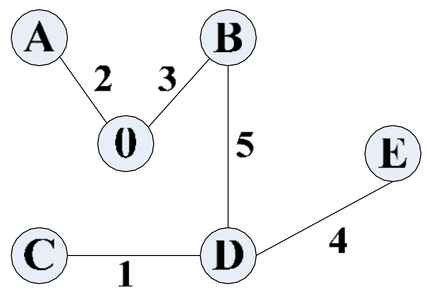
圖 **(A)**各處室的邏輯架構圖	圖 **(B)** 計算機中心到各處室的實際距離，以權重值來表示
	

【舉例】假設下圖為計算機中心(0)到各處室(A~E)的實際距離，
請追蹤此圖形可能的擴張樹。



【解答】

追蹤擴張樹可能有以下三個情況：

圖 **(A)**深度優先度	圖(B)**廣度**優先度	圖**(C)** 最小成本擴張樹
 權重值總和為： **1+2+4+7+14=28**	 權重值總和為： **2+3+4+10+18=37**	 權重值總和為：**1+2+3+4+5=15**

7-7.1 Kruskal 演算法

【定義】

Kruskal 演算法每次挑選一個權重值最小的邊，加入到T中，並以形成最小成本擴張樹，但不可形成迴圈，直到數量達n-1個邊為止。

此種演算法是根據各邊的加權值大小，再由小到大排序後，再選取要加入T的邊。

【作法】

假設有n個頂點的相連圖形，其Kruskal的演算步驟如下：

1. 邊的權重值先由小到大排序。

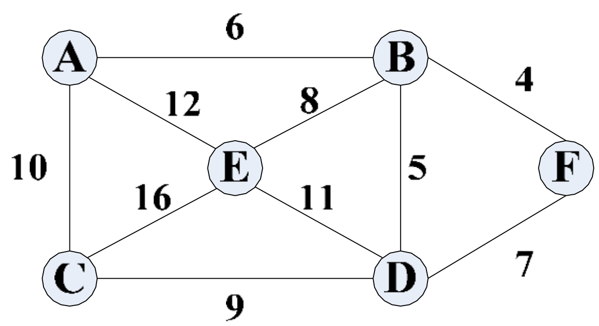
2. 從所有未走訪的邊中取出最小權重值的邊，記錄此邊已走訪，檢查是否形成迴路。

   (1) 如果形成迴路，此邊不能加入MST中，回到步驟2。

   (2)如果尚未形成迴路，此邊加入MST中，如果邊數已達（n-1）條則到步驟3，否則回到步驟2。

3. Kruskal演算法可以找出MST，結束。

【實例】請利用Kruskal 演算法來求出下圖的最小成本擴張樹。

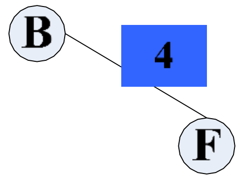


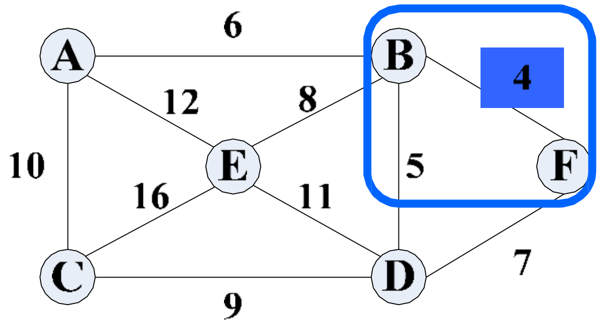
【解答】

步驟1：將所有邊的權重值先由小到大排序

起始頂點	終止頂點	權重值(成本)	由小到大排序
B	F	4	1
B	D	5	2
A	B	6	3
D	F	7	4
B	E	8	5
C	D	9	6
A	C	10	7
D	E	11	8
A	E	12	9
C	E	16	10

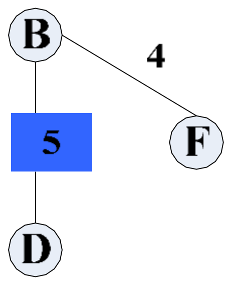
步驟2：選擇最小權重值的邊當作最小成本擴張樹的起點。

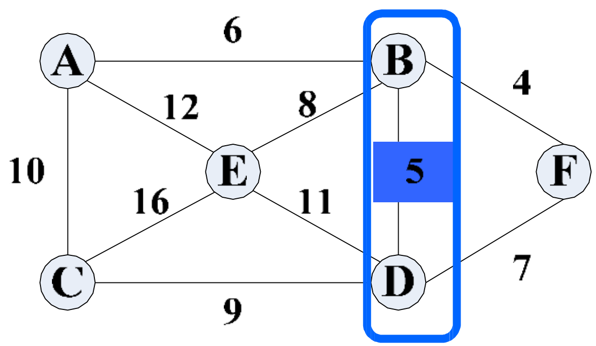




起始頂點	終止頂點	權重值(成本)	由小到大排序
B	**F**	**4**	**1**
B	D	5	2
A	B	6	3
D	F	7	4
B	E	8	5
C	D	9	6
A	C	10	7
D	E	11	8
A	E	12	9
C	E	16	10

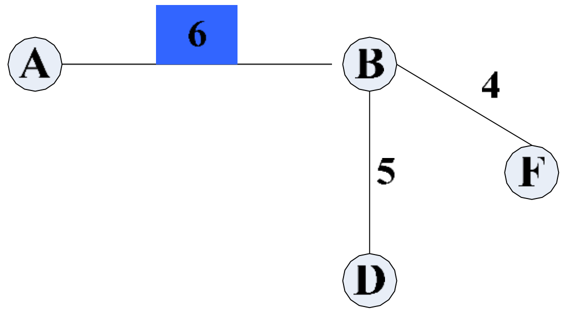
步驟3：依照步驟1的表格之權重值大小，加入第2小的權重值於最小成本擴張樹中。

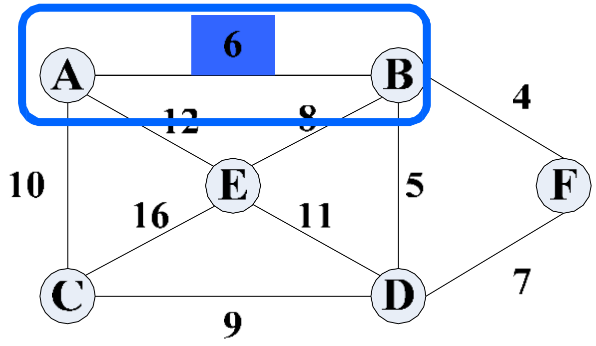




起始頂點	終止頂點	權重值(成本)	由小到大排序
B	F	4	1
B	D	5	2
A	B	6	3
D	F	7	4
B	E	8	5
C	D	9	6
A	C	10	7
D	E	11	8
A	E	12	9
C	E	16	10

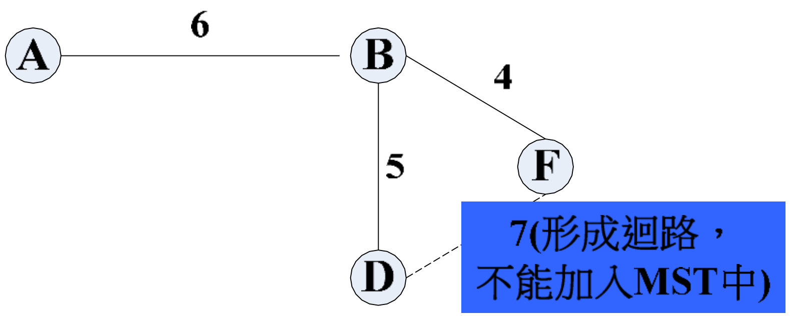
步驟**4**：依照步驟**1**的表格之權重值大小，加入第**3**小的權重值於最小成本擴張樹中。

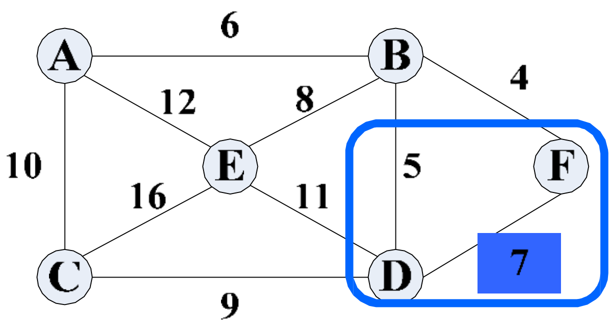




起始頂點	終止頂點	權重值(成本)	由小到大排序
B	F	4	1
B	D	5	2
A	B	6	3
D	F	7	4
B	E	8	5
C	D	9	6
A	C	10	7
D	E	11	8
A	E	12	9
C	E	16	10

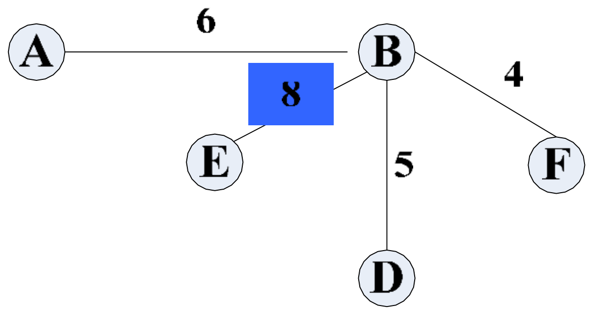
步驟5：依照步驟1的表格之權重值大小，加入第4小的權重值於最小成本擴張樹中，但是形成迴路，不能加入MST中，所以直接跳過。

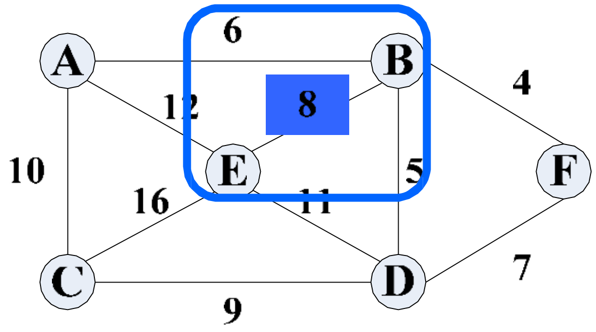




起始頂點	終止頂點	權重值(成本)	由小到大排序
B	F	4	1
B	D	5	2
A	B	6	3
D	F	7	4
B	E	8	5
C	D	9	6
A	C	10	7
D	E	11	8
A	E	12	9
C	E	16	10

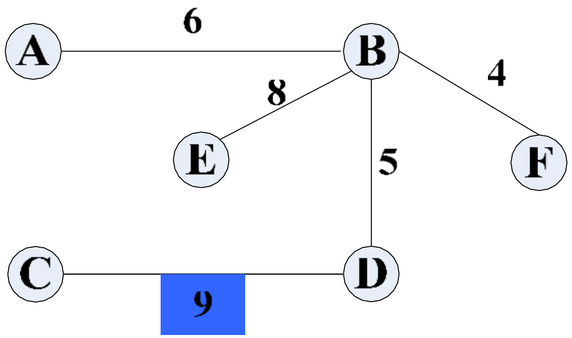
步驟**6**：依照步驟**1**的表格之權重值大小，加入第**5**小的權重值於最小成本擴張樹中。

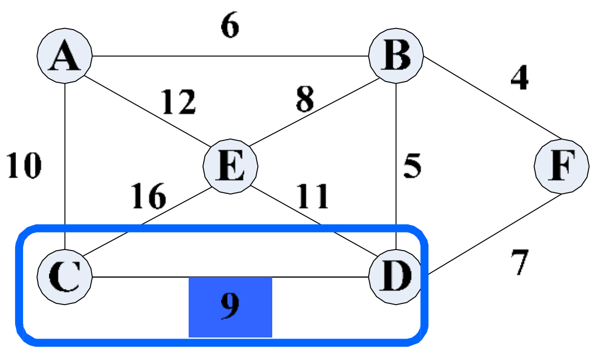




起始頂點	終止頂點	權重值(成本)	由小到大排序
B	F	4	1
B	D	5	2
A	B	6	3
D	F	7	4
B	E	8	5
C	D	9	6
A	C	10	7
D	E	11	8
A	E	12	9
C	E	16	10

步驟7：依照步驟1的表格之權重值大小，加入第6小的權重值於最小成本擴張樹中。由於圖中有6個頂點，而此時MST中已有5個邊，因此最後結果為：





起始頂點	終止頂點	權重值(成本)	由小到大排序
B	F	4	1
B	D	5	2
A	B	6	3
D	F	7	4
B	E	8	5
C	D	9	6
A	C	10	7
D	E	11	8
A	E	12	9
C	E	16	10

7-7.2 Prims 演算法

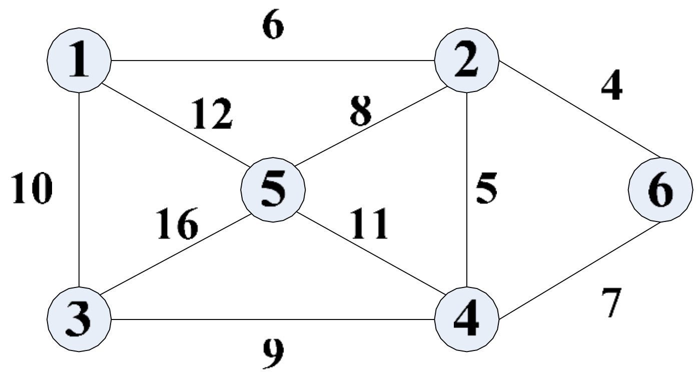
【定義】

假設有一個圖形G = (V,E) ，其中V={1,2,3,...,n}，且最初設定U={1}，U,V 是兩個頂點的集合，並且每次會產生一個邊。亦即從U-V 集合中找一個頂點V，能與U集合中的某頂點形成最小成本的邊，把這一頂點V加入U集合，繼續此步驟，直到U集合等於V集合為止。

【作法】

1. 選出某一節點**U**出發點。
2. 從與**U**節點相連且尚未被選取的節點中，選擇權重最小的邊，加入新節點。
3. 重複加入新節點，直到**n-1**條邊為止。(其中**n**為節點數)

【實例】請利用Prims 演算法來求出下圖的最小成本擴張樹。



【解答】

步驟**1**： U及V 是圖形中頂點的集合，假設U集合的起始點為1

U={1}

V={1,2,3,4,5,6}

步驟**2**：找到邊(1,2)為最小(6)，所以將頂點2加入到U集合中。

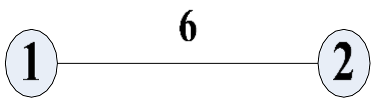
U={1}

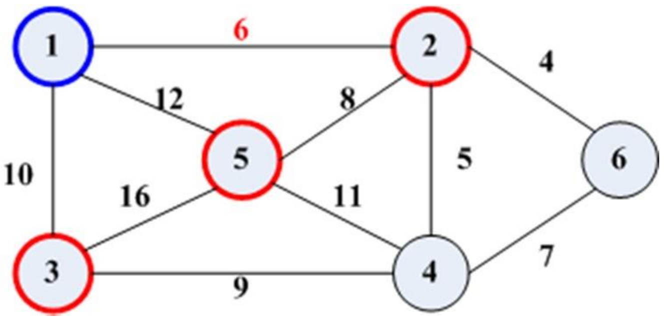
6 10 12

V-U={2, 3, 4, 5, 6}

U={1,2}

V-U={3, 4, 5, 6}





步驟**3**：找到邊(2,6)為最小(4)，所以將頂點6加入到U集合中。

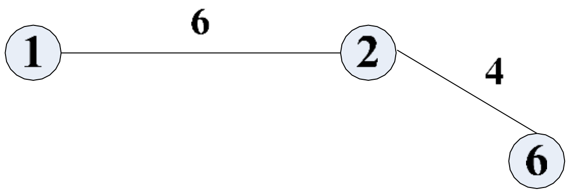
U={1,2}

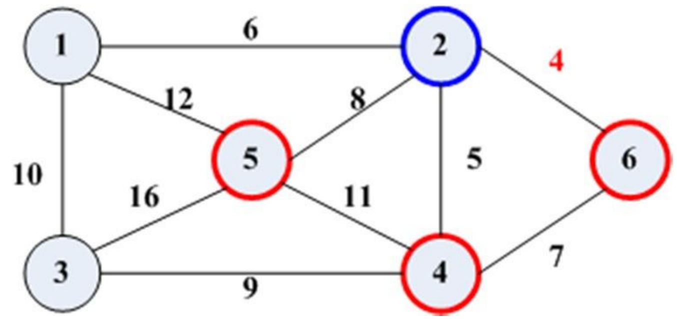
5 8 4

V-U={3, 4, 5, 6}

U={1,2,6}

V-U={3, 4, 5}





步驟**4**：找到邊(2,4)為最小(5)，所以將頂點4加入到U集合中。

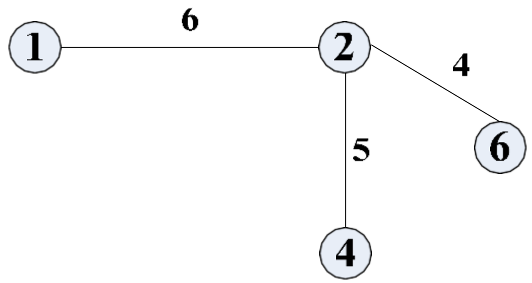
U={1,2,6}

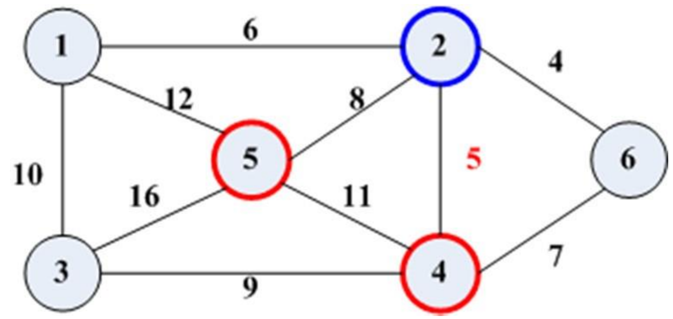
5, 8

V-U={3, 4, 5}

U={1,2,4,6}

V-U={3, 5}





步驟**5**：找到邊(2,5)為最小(8)，所以將頂點5加入到U集合中。

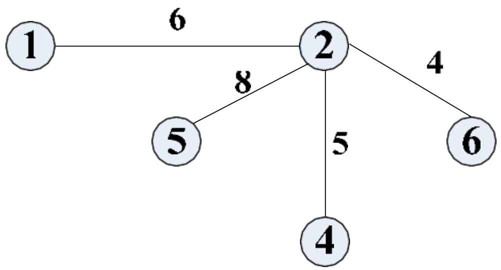
U={1,2,4,6}

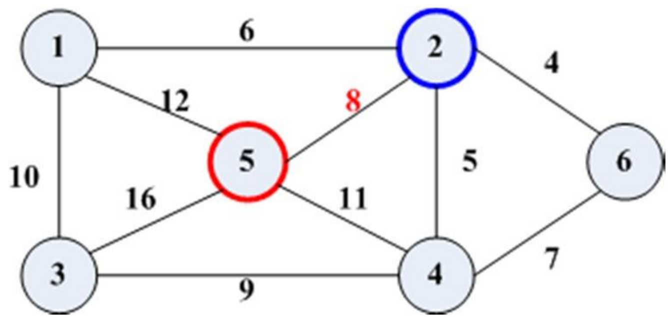
8

V-U={3, 5}

U={1,2,4,5,6}

V-U={3}





步驟**6**：找到邊(4,3)為最小(9)，所以將頂點3加入到U集合中。

U={1,2,4,5,6}

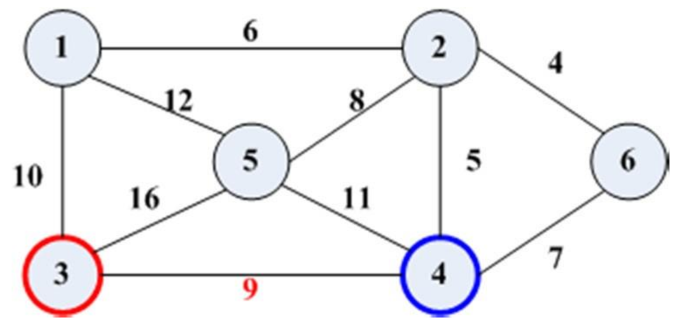
9

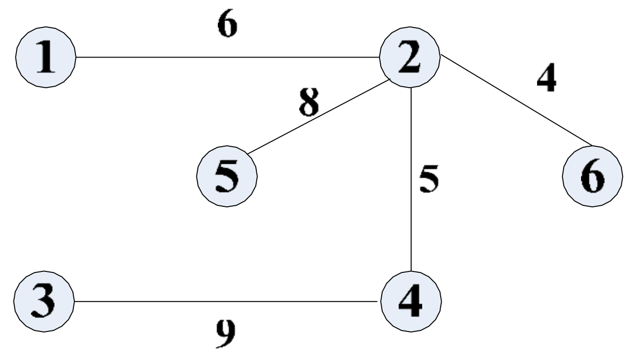
V-U={3}

U={1,2,3,4,5,6}

V-U={}

V={1,2,3,4,5,6}





最後，將頂點3加入到U集合中，此時集合U等於集合V，動作結束。

7-8 最短路徑(shortest path)

最短路徑(shortest path)問題是目前圖形結構中常見的典型問題之一。因為圖形中某頂點到達各頂點的路徑不是唯一，如果要從眾多的路徑中找出路徑最短者，則稱為最短路徑問題，可分為兩種形式：

1. 單點到其他各頂點之最短路徑。
2. 各個節點之間之最短路徑。

【舉例】單點到其他各頂點之最短路徑的問題

此問題的典型應用是由甲城市（頂點）到乙城市（頂點）之間的距離（權重值），計算由甲城市出發，經由多重交通網路的計算，到達乙城市的最短路徑，此種問題可行走的路徑，往往是有多條的，因此，我們必須要從這多條路徑中選擇最短路徑。

【求最短距離之作法】

步驟 1 ：

D[I] = A[F,I] (I = 1 , …,N)

S = { F }

V = { 1 , 2 , 3 , ... , N }

其中：D為 N 個位置的陣列，用來儲存某一頂點到其他頂點的最短距離。

F 表示由某一起始點開始。

A[F,I] 是表示F 點到I 點的距離。

V 是網路中所有頂點的集合。

S 也是頂點的集合。

步驟 2 ：

從 V-S 集合中找一頂點t 使得 D[t] 是最小值，並將頂點t 放入 S集合，一直到V-S 是空集合為止。

步驟 3 ：

根據下面的公式調整D陣列中的值。

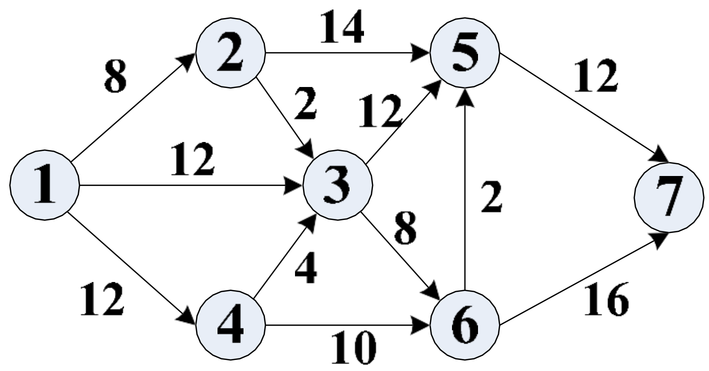
D[I] = min(D[I],D[t]+A[t,I])

此處I 是指t 的相鄰各頂點。

繼續回到步驟 2 執行。

【實例】

求下圖的最短距離？



【解答】

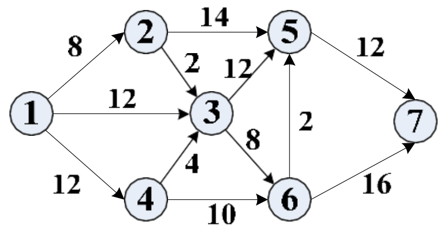
步驟 1： F = 1 ; S = { 1 } ; V = { 1 , 2 , 3 , 4 , 5 , 6 , 7 }

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0	8	12	12	∞	∞	∞

說明：

(1)假設我們從起始頂點1開始，頂點1到頂點2的距離為8，我們就可以把陣列 D[2]寫成8 ，而D[3]、D[4] 的值也是由頂點 1 到頂點 3 和頂點 4的距離，但是由於頂點 1無法由直接到達頂點5、頂點6及頂點7，因此，我們把 D[5]、D[6]、D[7]的值設定為∞ 。

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0	8	12	12	∞	∞	∞



(2) 由上面的陣列中，**頂點1**到**頂點2**的距離最短(即**D[2]**)，因此，將頂點 **2** 加入到 **S** 的集合中，因此，**S = { 1 , 2 }** ，**V-S = { 3 , 4 , 5, 6 , 7 }**

(3) 頂點 **2** 的相鄰頂點為 **3 , 5** ，則：

D[3] = min (D[3] , D[2]+A[2,3]) = min (12 , 8+2) = 10

說明：此時，**頂點1**到**頂點3**可透過頂點**2**，其距離就變成**10**。

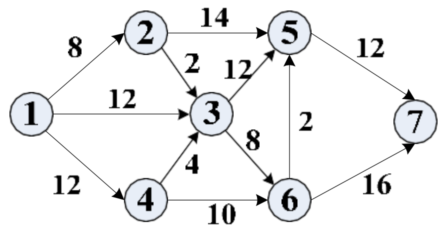
D[5] = min (D[5] , D[2]+A[2,5]) = min (∞ , 8+14) = 22

說明：此時，**頂點1**到**頂點5**可透過頂點**2**，其距離就變成**22**。

此時，陣列中的內容如下：

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0	8	10	12	22	∞	∞

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0	8	10	12	22	∞	∞



步驟2：

(1) 由上面的陣列中，頂點**1**到頂點**3**的距離最短(即**D[3]**)，因此，將頂點**3** 加入到 **S** 的集合中，因此，**S = { 1 , 2 , 3}** ，**V-S = { 4 , 5, 6 , 7 }**

(2) 頂點 **3** 的相鄰頂點為**5 , 6** ，則：

D[5] = min (D[5] , D[3]+A[3,5]) = min (22 , 10 + 12) = 22

說明：此時，頂點**1**到頂點**5**可透過頂點**2**與頂點**3**，其距離就變成**22**。

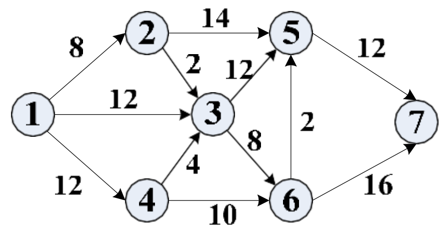
D[6] = min (D[6] , D[3]+A[3,6]) = min (∞ , 10 + 8) = 18

說明：此時，頂點**1**到頂點**6**可透過頂點**2**與頂點**3**，其距離就變成**18**。

此時，陣列中的內容如下：

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0	8	10	12	22	18	∞

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0	8	10	12	22	18	∞



步驟3：

(1) 由上面的陣列中，頂點1到頂點4的距離最短(即D[4])，因此，將頂點4 加入到 **S** 的集合中，因此，**S = { 1 , 2 , 3 , 4}** ，**V-S = { 5, 6 , 7 }**

(2) 頂點 **4** 的相鄰頂點為**3 , 6** ，則：

D[3] = min (D[3] , D[4]+A[4,3]) = min (10 , 12 + 4) = 10

說明：此時，頂點**1**到頂點**3**，其距離就變成**10**。

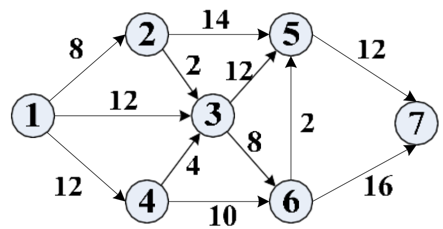
D[6] = min (D[6] , D[4]+A[4,6]) = min (18 , 12 + 10) = 18

說明：此時，頂點**1**到頂點**6**，可透過頂點**2**與頂點**3**，其距離就變成**18**。

所以陣列內容如下：

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0	8	10	12	22	18	∞

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0	8	10	12	22	18	∞



步驟4：

(1) 由上面的陣列中，頂點**1**到頂點**6**的距離最短(即**D[6]**)，因此，將頂點**6** 加入到 **S** 的集合中，因此，**S = { 1 , 2 , 3 , 4, 6}** ，**V-S = { 5, 7 }**

(2) 頂點 **6** 的相鄰頂點為**5 , 7**，則：

D[5] = min (D[5] , D[6]+A[6,5]) = min (22 , 18 + 2) = 20

說明：此時，頂點**1**到頂點**5**，可透過頂點**2**與頂點**3**，其距離就變成**20**。

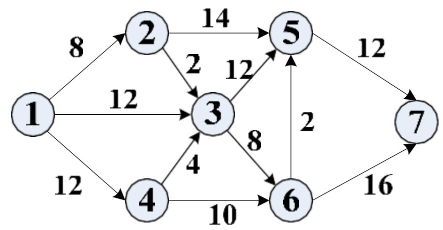
D[7] = min (D[7] , D[6]+A[6,7]) = min (∞ , 18 + 16) = 34

說明：此時，頂點**1**到頂點**7**，可透過頂點**2**與頂點**3**及頂點**6**，其距離就變成**30**。

所以陣列內容如下：

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0	8	10	12	20	18	34

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0	8	10	12	20	18	34



步驟5：

(1) 由上面的陣列中，頂點**1**到頂點**5**的距離最短(即**D[5]**)，因此，將頂點**5** 加入到 **S** 的集合中，因此，**S = { 1 , 2 , 3 , 4, 5, 6}** ，**V-S = { 7 }**

(2) 頂點 **5** 的相鄰頂點為 **7**，則：

D[7]=min (D[7] , D[5] + A[5,7]) = min (34 , 20 + 12) = 32

說明：此時，頂點**1**到頂點**7**，可透過頂點**2**、頂點**3**、頂點**6**及頂點**5**，其距離就變成**32**。

由於頂點 **7** 為最終頂點，將其加入 **S** 集合，此時，**S = { 1 , 2 , 3 , 4, 5, 6,7}** 因此，**V-S = {}** 。

最後陣列內容為：

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0	8	10	12	20	18	32

此陣列表示由頂點 **1** 到任一頂點的最短距離。

7-9 拓樸排序(Topological Sort)

【 AOV網路的引言】

在圖形結構中的工作(Activity)是指將一個計劃分成數個子計劃，而每一個子計劃完成時，即是整個計劃的完成，這個就稱為「工作」。

因此，如果我們將「工作」稱為工作網路上的「頂點」，而工作與工作之間的連線，代表著工作的優先順序時稱為工作網路上的「邊」。因此，這種以頂點來代表工作項目的網路稱為頂點工作網路(Activity On Vertex Network)，簡稱為AOV網路。

【定義】

若在AOV網路中，Vi 是 Vj 的前行者，則在線性的排列中，Vi 一定在 Vj 的前面，此種特性稱之為**拓樸排序 (Topological Sort)** 。

尋找AOV網路之拓樸排序的過程如下：

步驟 1 :在AOV網路中任意挑選一個沒有前行者的頂點。

步驟2 ： 輸出此頂點，並將該頂點所連接的邊刪除。

重覆步驟 1 及步驟 2，一直到全部的頂點皆輸出為止。

【應用】

1. 大專院校的選課資訊系統先修或擋修等限制

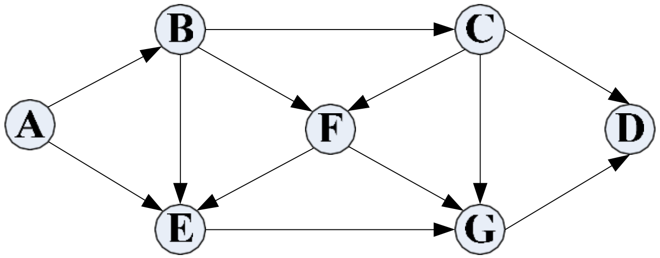
2. 政府部門的公文有層層呈報和會簽的流程

3. 資訊系統開發

4. 專案管理

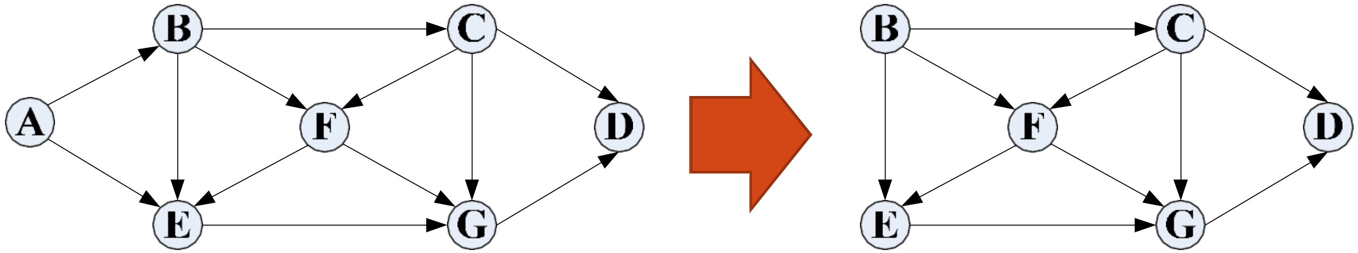
【範例】

求下面AOV網路之拓樸排序



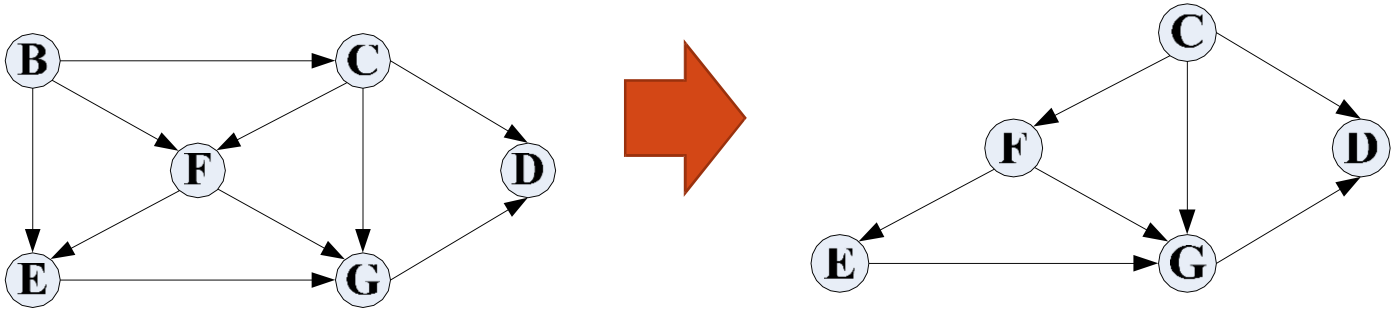
【解答】

步驟一：輸出A ，並刪除< A,B > 與 < A,E > 兩個邊



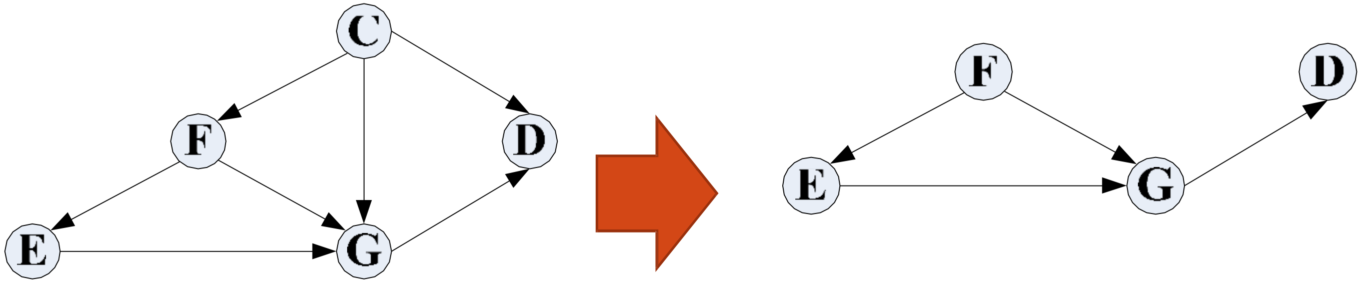
目前輸出的頂點： A

步驟二：輸出B，並刪除 < B,C>、<B,E>及<B,F> 三個邊



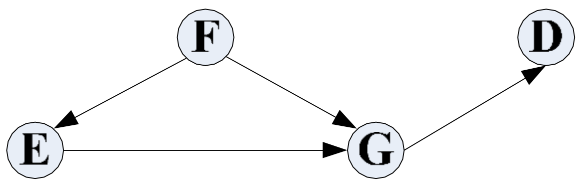
目前輸出的頂點： AB

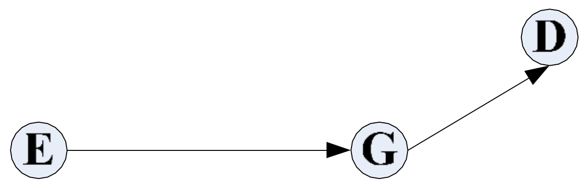
步驟三：輸出 C ，並刪除 < C,D>、<C,F>及<C,G> 三個邊



目前輸出的頂點： ABC

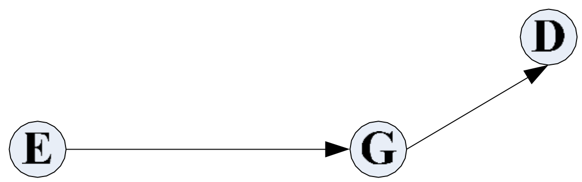
步驟四：輸出 **F**，並刪除 **< F,E >** 與 **< F,G >** 兩個邊

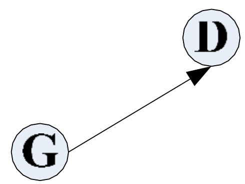




目前輸出的頂點： ABCF

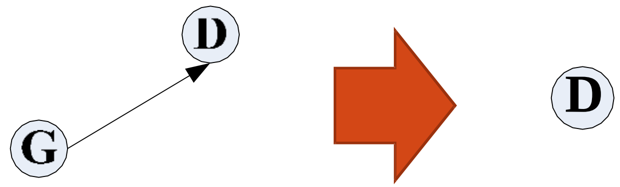
步驟五：輸出 **E** ，並刪除 **< E,G >**





目前輸出的頂點： ABCFE

步驟六：輸出 **G** ，並刪除 **< G,D >**



目前輸出的頂點： ABCFEG

步驟七：輸出 **D**

其所得的資料輸出順序為 **A , B , C , F , E , G , D**

以上的輸出順序即為拓樸排序