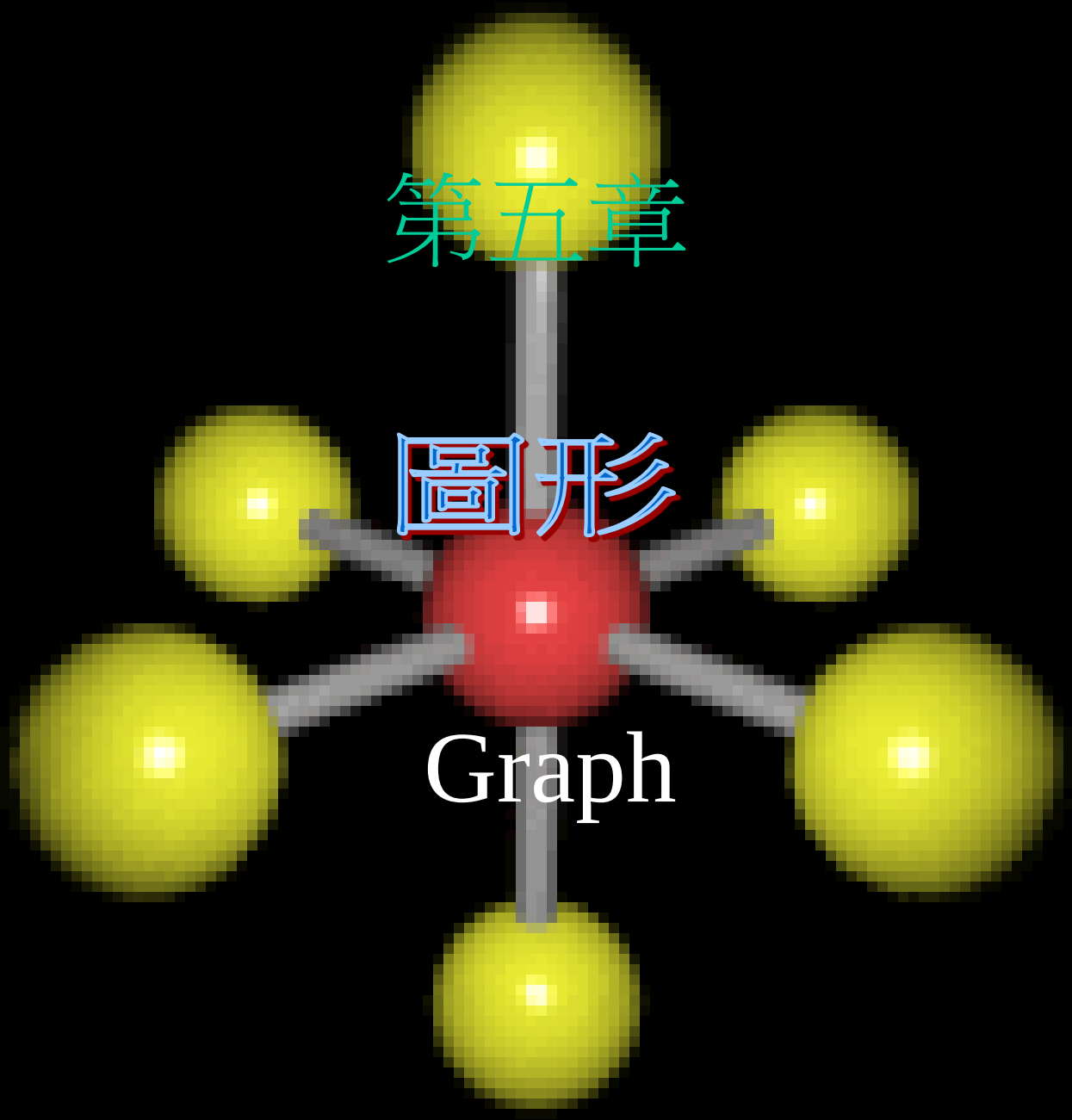


第五章

圖形

Graph

本章內容

5-1 圖形及定義

5-2 表示圖形的資料結構
鄰接矩陣、鄰接串列

5-3 圖形的走訪
廣度優先、深度優先

5-4 展開樹 (Spanning Tree)
Kruskal演算法、Prim演算法

5-5 最短路徑 (Shortest Path)
Dijkstra演算法、Floyd演算法

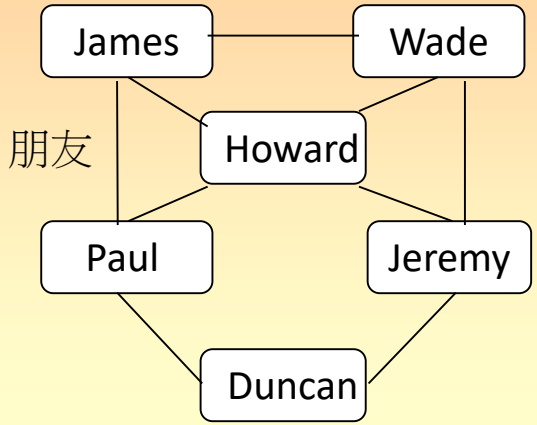
5-6 拓樸排序 (Topological Sorting)

5-7 關鍵路徑 (Critical Path)

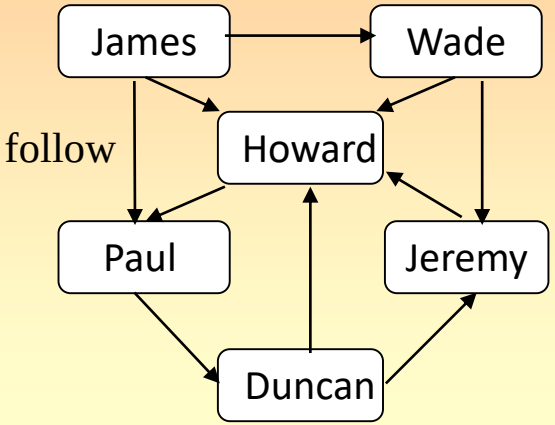
遊戲中的**地圖結合實境**，當你到處趴趴走，走到某個點的時候，與這個點相關的怪物、寶物或訊息就會跟著出現。遊戲程式設計者是如何在遊戲進行中管理地圖的相關資訊？

答案是使用**「圖形」的資料結構和演算法**，就能處理地圖資訊中的地點、道路或位置 座標，在遊戲進行的過程中，取出適當的資訊，這樣就可以讓你進入實境。

5-1 圖形及定義



社群網路(一)



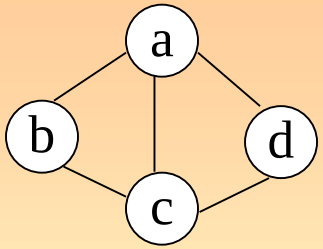
社群網路(二)

圖形 ：通信網路圖、電路圖、社群網路、交通路線圖…等

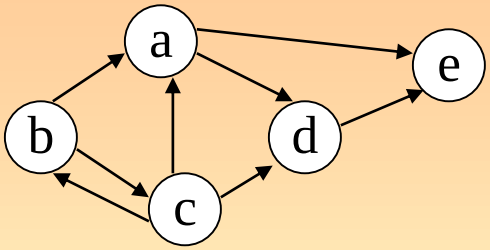
圖形 = 頂點 + 邊

Graph = (set of **V**ertices, set of **E**dges)

G = (V, E)



(a) G1 (無向圖)



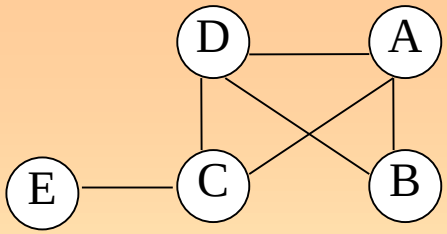
(b) G2 (有向圖)

G = (V, E)

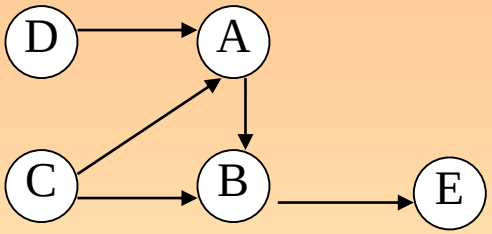
V(G1) = { a, b, c, d }（圖形 G1 中所有頂點所成的集合）
V(G2) = { a, b, c, d , e }

E(G1) = {(a,b), (a,c), (a,d), (b,c), (c,d) }（圖形 G1 中所有邊所成的集合）
（每個無向邊用一個無序對來表示）

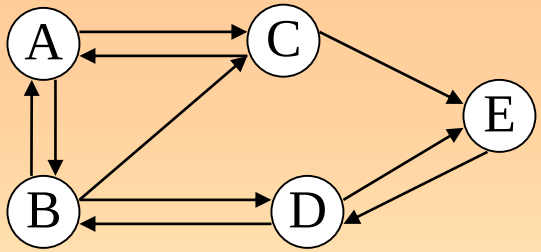
E(G2) = { <a,d>, <a,e>, <b,a>, <b,c>, <c,a>, <c,b>, <c,d>, <d,e> }
（每個有向邊用一個有序對來表示）



(a) G1 (無向圖)



(b) G2 (有向圖)



(c) G3 (有向圖)

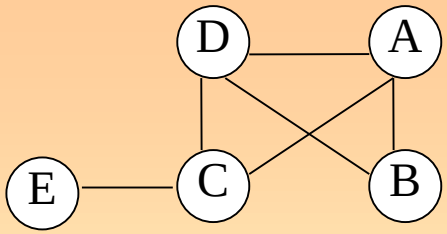
- **相鄰**(adjacency)：有邊相連的兩頂點。在G1中，頂點 A 與頂點 B 相鄰(adjacent)；在G2中，頂點 A相鄰到頂點 B 。

- **路徑**(path)：連續的邊。在G1中，E→C→D→B是一條路徑；
  環路(cyclte)：頭尾相接的路徑。在G1中， C→D→A→C 是一個環路。

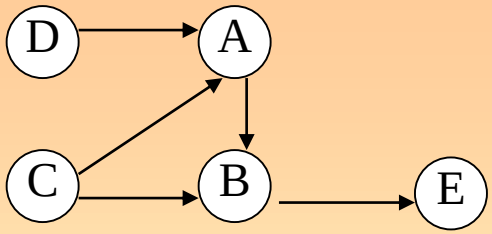
- **連通**(connect)：有路徑相通的兩頂點。在G1中，頂點 B 與頂點C 相連通(但不相鄰)；在G2中，頂點 D連通到頂點 E 。

- **連通圖**(connect graph)：任兩個頂點都連通。 G1是連通圖

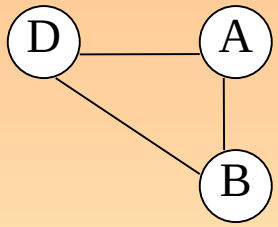
- **強連通圖**(strongly connect graph)： 有向圖中任兩個頂點都互相連通。G2不是強連通圖，例如 A 無法連通到 D。 G3是強連通圖



(a) G1 (無向圖)



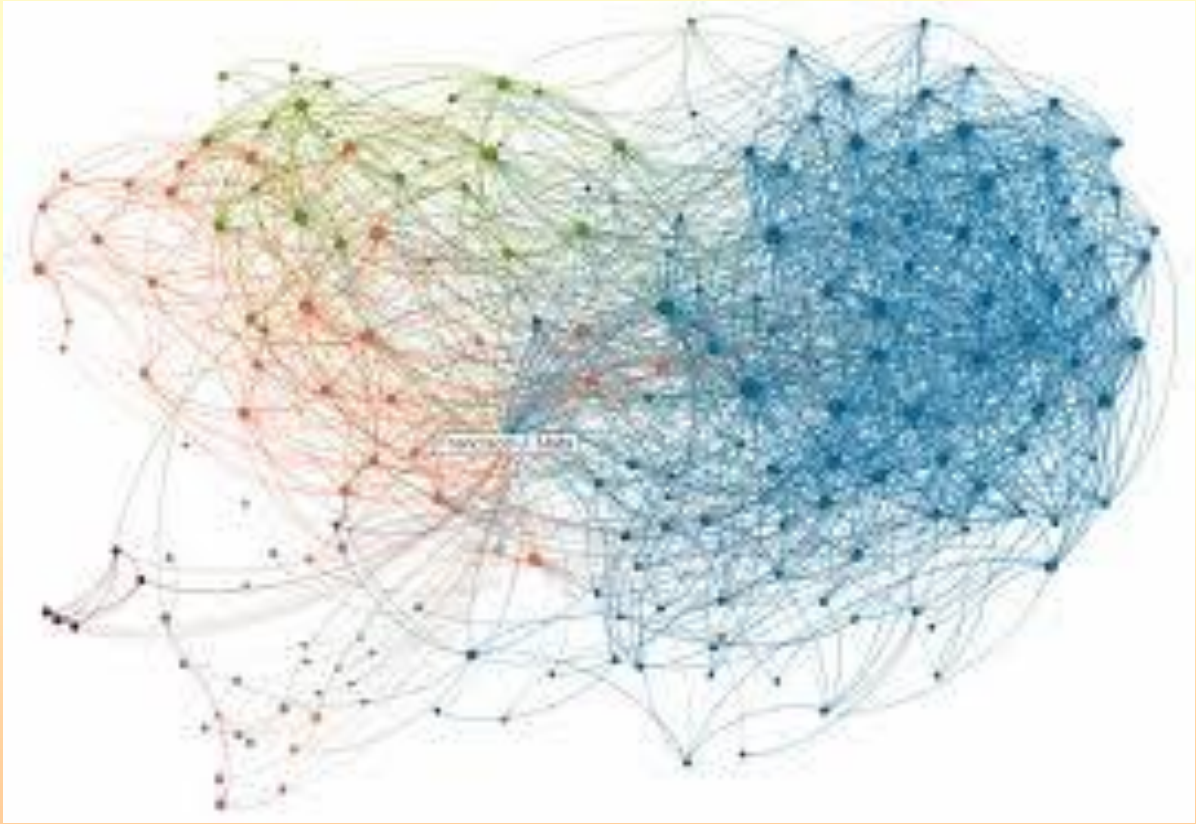
(b) G2 (有向圖)



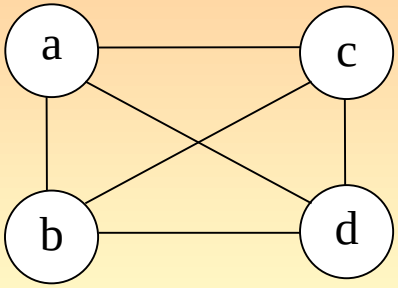
(c) G3 (無向圖)

- **子圖**(subgraph)：子圖的頂點與邊都被包含在原圖中。G3是G1 的子圖)。
- **分支度**(degree)：與頂點相接邊的數目。在G1中，頂點 B 的分支度是 2；頂點 E 的分支度是 1。
- **出分支度**(out-degree)：頂點往外有向邊的數目。在G2中，頂點 B 的出分支度是 1
- **入分支度**(in-degree)：頂點往內有向邊的數目。在G2中，頂點 B 的入分支度是 2。

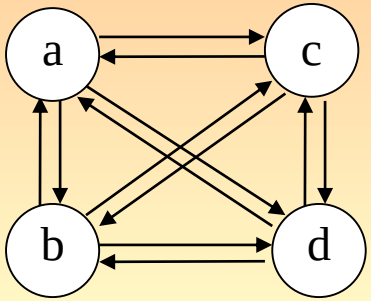
FB 使用圖形結構記載用戶資訊，每個用戶是一個頂點，兩點之間的連線 代表這兩個用戶是朋友關係，你認為這個圖是有向圖還是無向圖？



完全圖 (compltete graph)



(a) 此無向圖是一個完全圖。
4個頂點6個邊(= 4*3/2)。



(b) 此有向圖是一個完全圖。
4個頂點12個邊(= 4*3)。

- 無向完全圖：任何一對不同的頂點v_1和v_2間都是相鄰的，

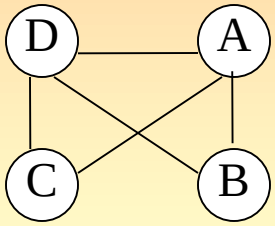
  n 個頂點 => $n(n\text{-}1)/2$個無向邊

- 有向完全圖：任何一對不同的頂點v_1和v_2，v_1都有邊到v_2，同時v_2都有邊到v_1

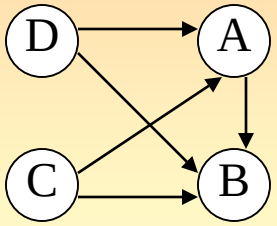
  n 個頂點 => $n(n\text{-}1)$個有無向邊

5-2 表示圖形的資料結構

鄰接矩陣 (adjacency matrix)



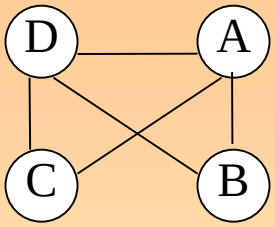
(a) G1 (無向圖)



(b) G2 (有向圖)

$$M = \begin{array}{c} \\ A(0) \\ B(1) \\ C(2) \\ D(3) \end{array} \begin{array}{c} \begin{array}{cccc} A(0) & B(1) & C(2) & D(3) \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array} \quad 4 \times 4$$

$$M = \begin{array}{c} \\ A(0) \\ B(1) \\ C(2) \\ D(3) \end{array} \begin{array}{c} \begin{array}{cccc} A(0) & B(1) & C(2) & D(3) \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \end{array} \quad 4 \times 4$$



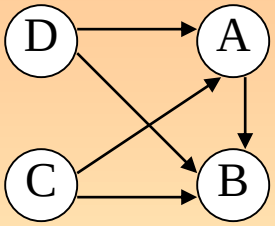
(a) G1 (無向圖)

M =

	A(0)	B(1)	C(2)	D(3)
A(0)	0	1	1	1
B(1)	1	0	0	1
C(2)	1	0	0	1
D(3)	1	1	1	0

4 × 4

1. 鄰接矩陣中元素的值只可能是0或1。
2. 主對角線上的元素都是0 (M[*i*][*i*]= 0)，因為頂點不可自成迴路。
3. M[*i*][*j*]為1代表頂點*i*和頂點*j*相鄰。例如頂點A和頂點B相鄰，所以 M [0][1] = 1。頂點A和頂點C也相鄰，所以M [0][2] = 1。
4. M[*i*][*j*]為0代表頂點*i*和頂點*j*不相鄰。例如頂點B沒有邊到頂點C（B、C不相鄰），所以M [1][2] = 0，以此類推。
5. 由於此圖是無向圖，頂點A有邊到頂點B代表同時頂點B有邊到頂點A（事實上是同一個邊），因此M [0][1] = 1，表示必定M [1][0] = 1。亦即對所有範圍內的*i*和*j*，M[*i*][*j*] = M[*j*][*i*]。因此無向圖的鄰接矩陣一定是個「對稱矩陣」(symmetric matrix)。
6. M中的非零項共有10個，恰好是邊數的2倍，原因在於每個邊各自造成矩陣兩個元素為1，因此都被重複算了兩次。
7. 第0列共有3個非零項，因為頂點A有3個邊相連，亦即頂點A的「分支度」為3。其餘各列的非零項數目，也分別代表對應頂點的分支度。由於M是對稱矩陣，所以第0列有3個非零項代表第0行也有3個非零項，都是說明頂點A的分支度為3。



(b) G2 (有向圖)

M =

	A(0)	B(1)	C(2)	D(3)
A(0)	0	1	0	0
B(1)	0	0	0	0
C(2)	1	1	0	0
D(3)	1	1	0	0

4 × 4

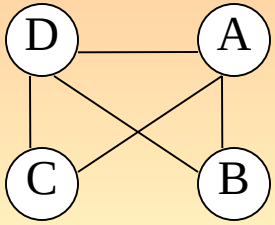
1. 有向圖的鄰接矩陣不一定是對稱矩陣（有無可能？）。

2. M[*i*][*j*]為1代表頂點*i*有邊到頂點*j*；但是M[*j*][*i*]不一定為1，因為頂點*j*不一定同時有邊到頂點*i*。因此整個矩陣的非零項的數目，恰好等於有向圖的邊數。

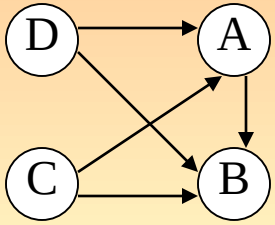
3. 第0列有1個非零項，代表頂點A有一個邊出去，亦即頂點A的「出分支度」為1。其餘各列的非零項數目，也分別代表各頂點的出分支度。

4. 第0行有2個非零項，代表頂點A有2個邊進來，亦即頂點A的「入分支度」為2。其餘各行的非零項數目，也分別代表各頂點的入分支度。

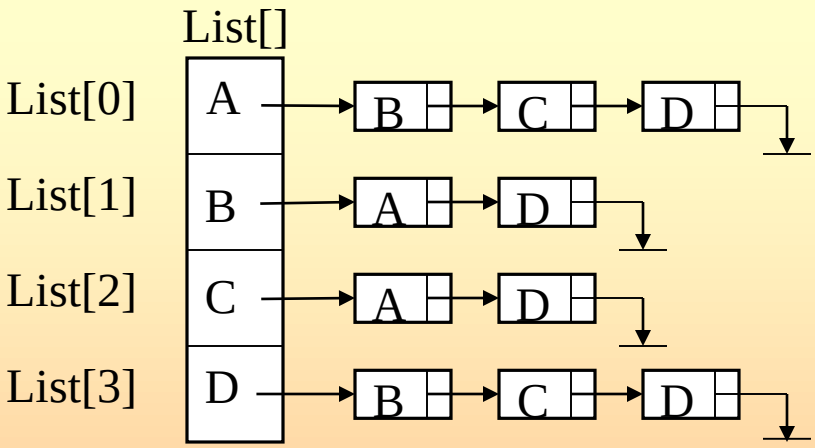
鄰接串列 (adjacency list)



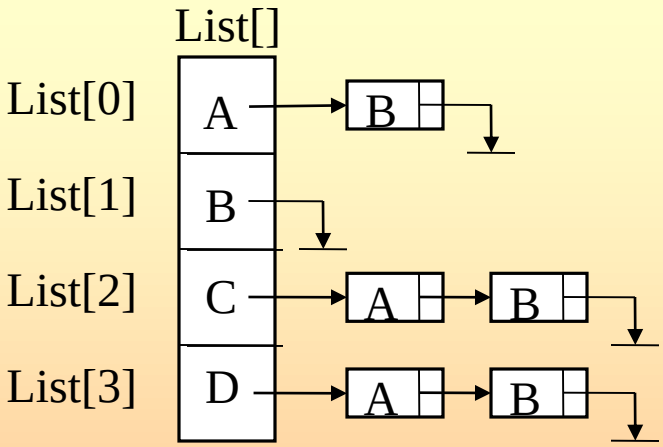
(a) G1 (無向圖)



(b) G2 (有向圖)

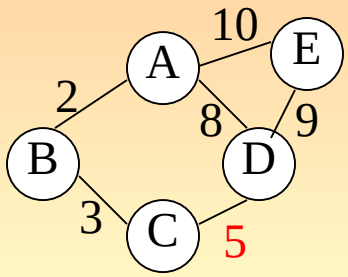


總節點數是邊數的 2 倍

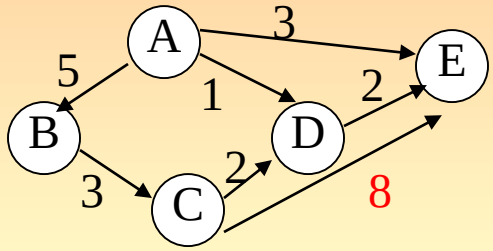


總節點數等於邊數

加權圖的鄰接矩陣



(a) G1 (無向圖)



(b) G2 (有向圖)

M1=

	A	B	C	D	E
A	0	2	∞	8	10
B	2	0	3	∞	∞
C	∞	3	0	5	∞
D	8	∞	5	0	9
E	10	∞	∞	9	0

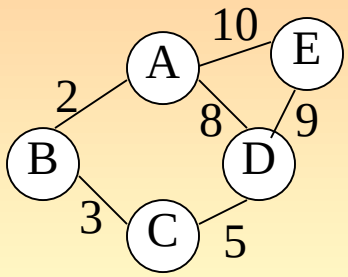
5 × 5

M2=

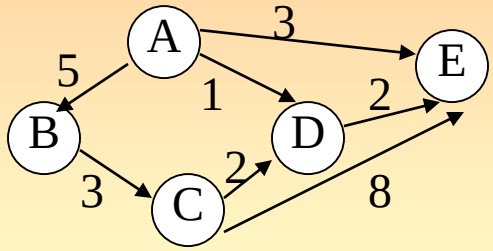
	A	B	C	D	E
A	0	5	∞	1	3
B	∞	0	3	∞	∞
C	∞	∞	0	2	8
D	∞	∞	∞	0	2
E	∞	∞	∞	∞	0

5 × 5

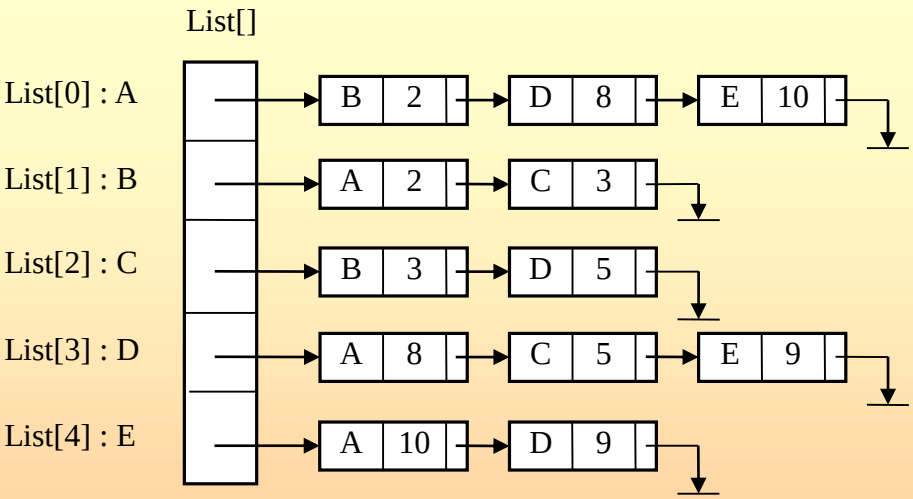
加權圖的鄰接串列



(a) G1 (無向圖)



(b) G2 (有向圖)



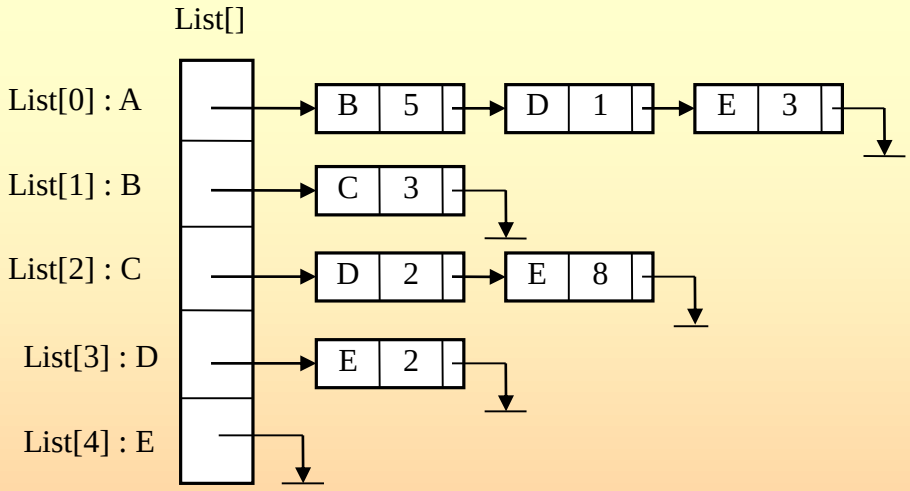
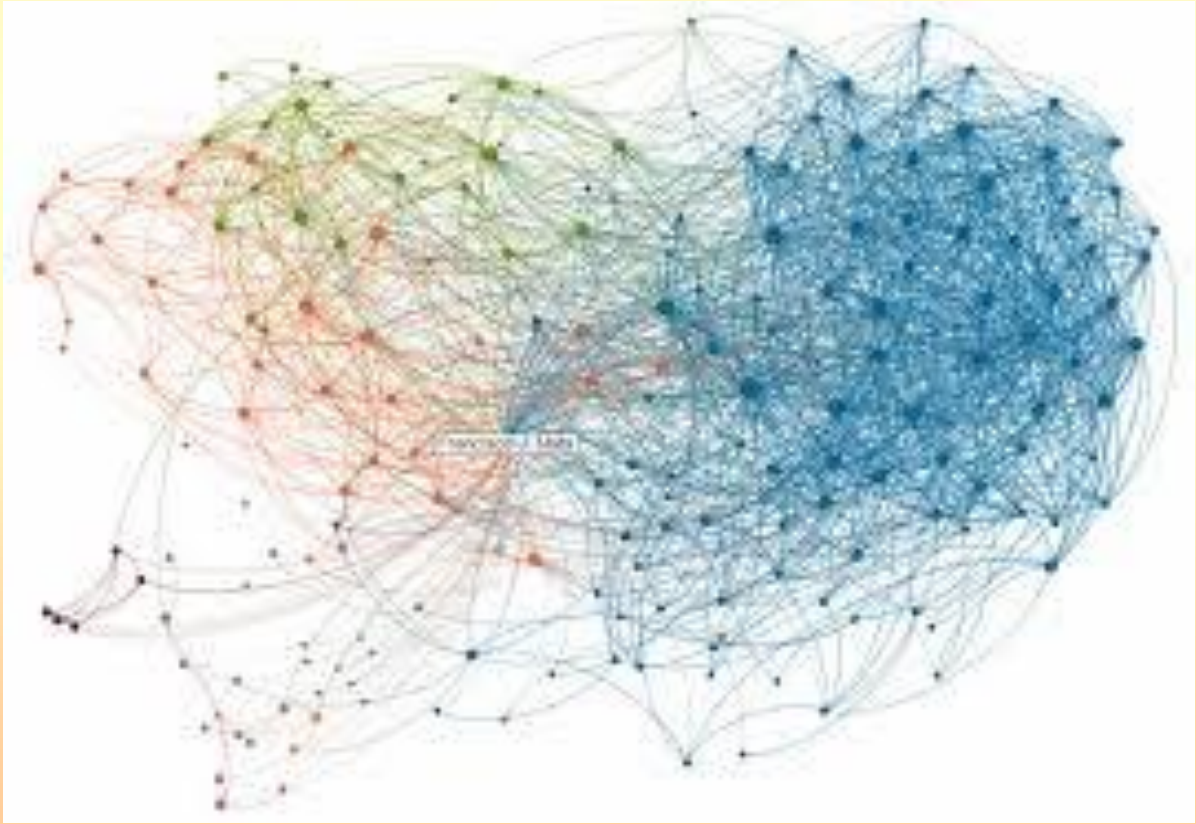


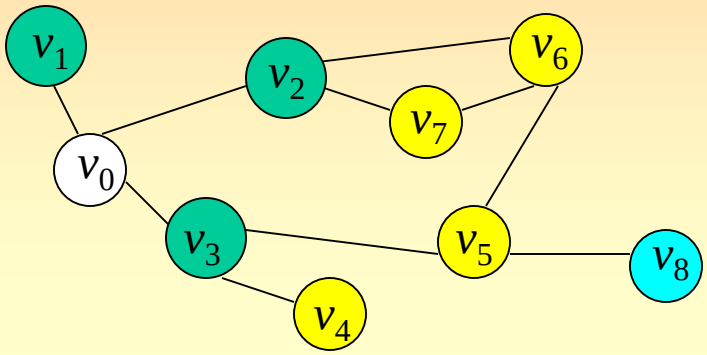
圖 5.7 (b) 圖 5.5(b) 的加權鄰接串列

FB 使用圖形結構記載用戶資訊，每個用戶是一個頂點，兩點之間的連線代表這兩個用戶是朋友關係，你認為 FB 會使用鄰接矩陣還是鄰接串列來儲存？



5-3 圖形的走訪

廣度優先走訪 (Breadth First Search, BFS)



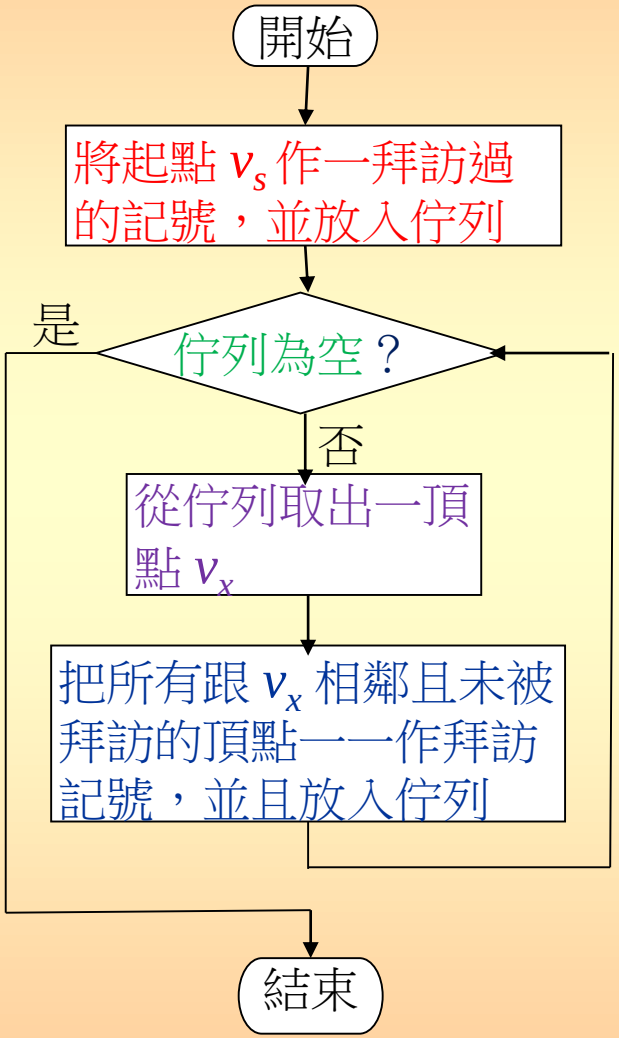
＊方法：從指定的起點 v_s 開始，拜訪和起點 v_s 相鄰的所有頂點。再拜訪更外一層的頂點，也就是與起點 v_s 相鄰頂點的相鄰頂點。直到所有連通頂點都拜訪過為止

＊如果以 v_0 為起點，其中一組 BFS 拜訪順序為：

v_0, v_1, v_2, v_3, v_6, v_7, v_5, v_4, v_8

第一層　第二層　第三層

BFS 演算法流程圖



BFS 演算法虛擬碼

演算法：BFS廣度優先走訪 (圖形G，起點 v_s)

拜訪v_s，並將v_s放入佇列

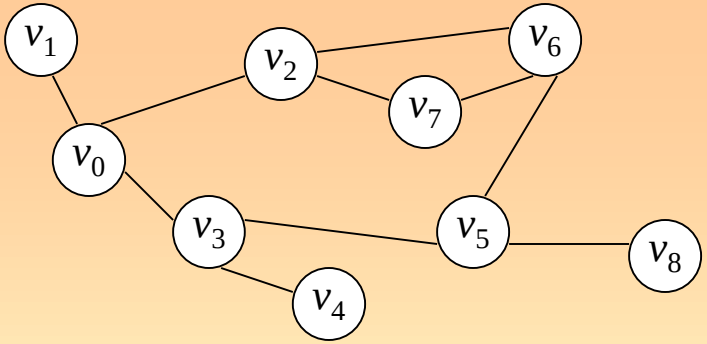
當佇列非空時，重複迴圈

從佇列取出一頂點v_x

逐一拜訪所有與v_x相鄰且未被拜訪的頂點v_y，並將v_y放入佇列

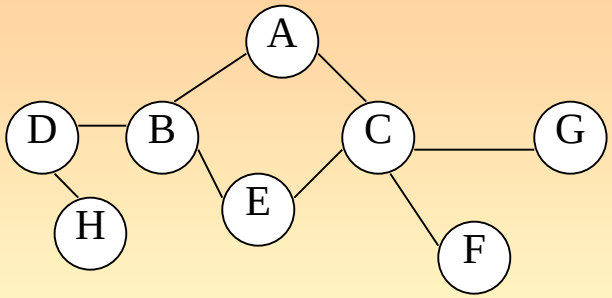
迴圈結束

演算法結束

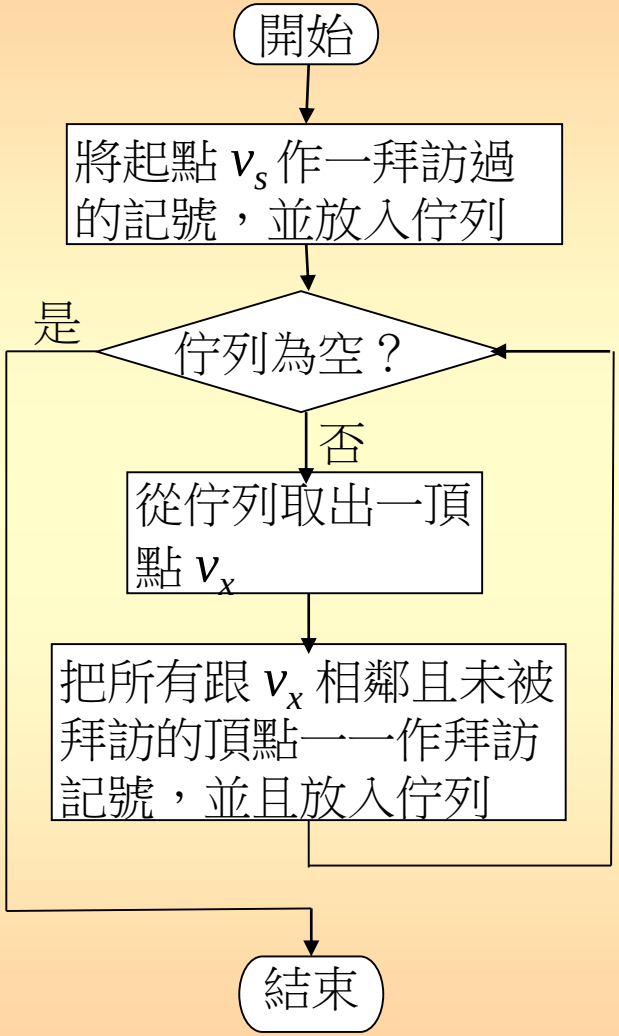


動作說明	佇列 (queue)	已走訪頂點
0. 初始狀態	(front) (rear)	無
1. 走訪v_0, Enqueue(v_0)	v_0	$\boldsymbol{v_0}$
2. Dequeue得v_0, 將所有與v_0相鄰且未被拜訪的頂點 (v_1, v_2, v_3) 一一拜訪且Enqueue	v_1, v_2, v_3	v_0, $\boldsymbol{v_1}$, $\boldsymbol{v_2}$, $\boldsymbol{v_3}$
3. Dequeue得v_1, 沒有任何頂點與v_1相鄰且未被拜訪的	v_2, v_3	v_0, v_1, v_2, v_3
4. Dequeue得v_2, 將所有與v_2相鄰且未被拜訪的頂點 (v_6, v_7) 一一拜訪且Enqueue	v_3, v_6, v_7	v_0, v_1, v_2, v_3, $\boldsymbol{v_6}$, $\boldsymbol{v_7}$
5. Dequeue得v_3, 將所有與v_3相鄰且未被拜訪的頂點 (v_4, v_5) 一一拜訪且Enqueue	v_6, v_7, v_4, v_5	v_0, v_1, v_2, v_3, v_6, v_7, $\boldsymbol{v_4}$, $\boldsymbol{v_5}$
6. Dequeue得v_6, 沒有任何頂點與v_6相鄰且未被拜訪的	v_7, v_4, v_5	v_0, v_1, v_2, v_3, v_6, v_7, v_4, v_5
7. Dequeue得v_7, 沒有任何頂點與v_7相鄰且未被拜訪的	v_4, v_5	v_0, v_1, v_2, v_3, v_6, v_7, v_4, v_5
8. Dequeue得v_4, 沒有任何頂點與v_4相鄰且未被拜訪的	v_5	v_0, v_1, v_2, v_3, v_6, v_7, v_4, v_5
9. Dequeue得v_5, 將所有與v_5相鄰且未被拜訪的頂點 (v_8) 一一拜訪且Enqueue	v_8	v_0,v_1,v_2,v_3,v_6,v_7,v_4,v_5,$\boldsymbol{v_8}$
10. Dequeue得v_8, 沒有任何頂點與v_8相鄰且未被拜訪的	空	v_0,v_1,v_2,v_3,v_6,v_7,v_4,v_5,v_8
11. 佇列已空, 停止		

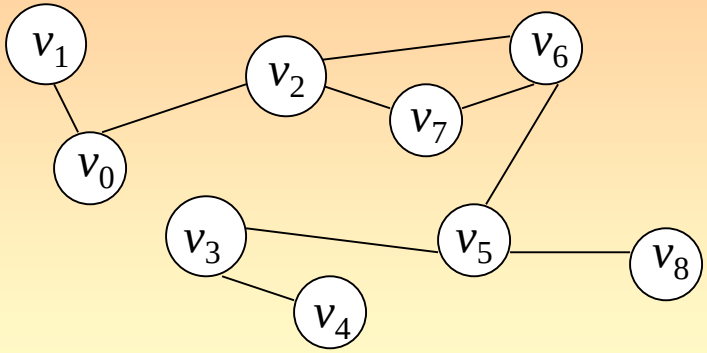
例 5.4 BFS



順序	佇列內容	已拜訪節點
1	A	A
2	BC	ABC
3	CDE	ABCDE
4	DEFG	ABCDEFG
5	EFGH	ABCDEFGH
6	FGH	ABCDEFGH
7	GH	ABCDEFGH
8	H	ABCDEFGH
9	空	ABCDEFGH



深度優先走訪 (Depth First Search, DFS)



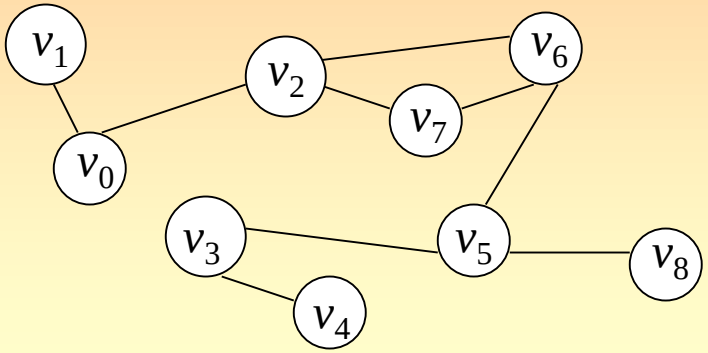
＊如果以 v_0 為起點，其中一組 DFS 拜訪順序為：

$v_0, v_1, v_2, v_6, v_7, v_5, v_8, v_3, v_4$

＊方法：從起點 v_s 開始，接著選擇和 v_s 相鄰的任一頂點 v_x ，並由 v_x **繼續**做深度優先拜訪。

當到達某個頂點 v_u 時，若 v_u 所有相鄰的頂點都已經被拜訪過，而無法繼續前進深入時，則**退回**到 v_u 的上個拜訪頂點，繼續作深度優先拜訪。

DFS遞迴版



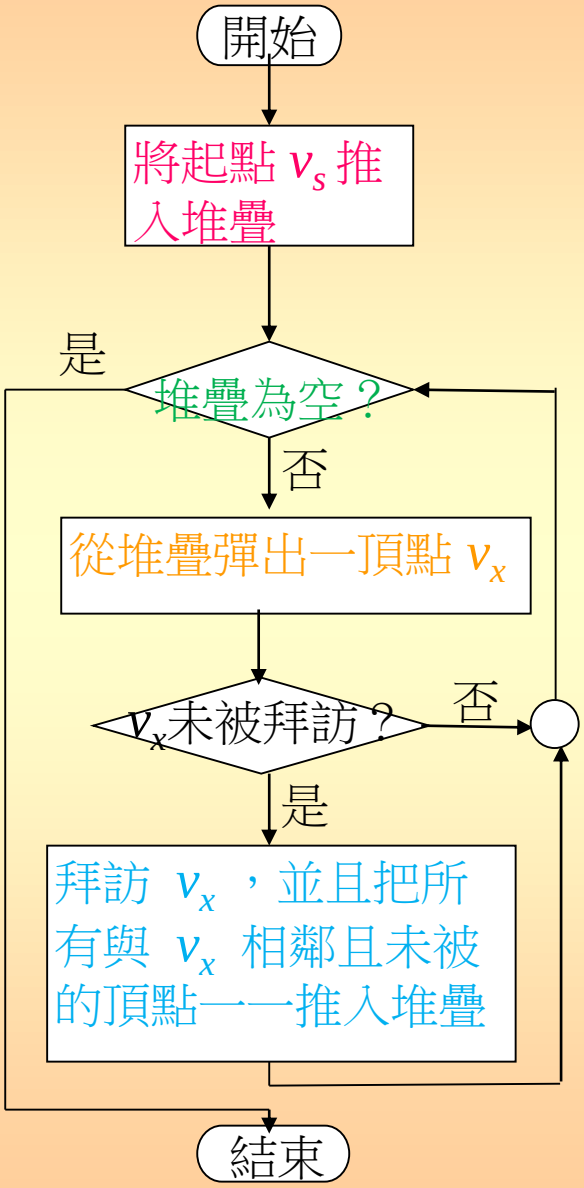
呼叫DFS(v_0) (進入函式v_0)
拜訪v_0 ----------(1)
呼叫DFS(v_1) (進入函式v_1)
拜訪v_1 ----------(2)
呼叫DFS(v_2) (進入函式v_2)
拜訪v_2 ----------(3)
呼叫DFS(v_6) (進入函式v_6)
拜訪v_6 ----------(4)
呼叫DFS(v_5) (進入函式v_5)
拜訪v_5 ----------(5)
呼叫DFS(v_3) (進入函式v_3)
拜訪v_3 ----------(6)
呼叫DFS(v_4) (進入函式v_4)
拜訪v_4 ----------(7)
呼叫DFS(v_8) (進入函式v_8)
拜訪v_8 ----------(8)
呼叫DFS(v_7) (進入函式v_7)
拜訪v_7 ----------(9)

演算法：遞迴版DFS深度優先走訪 (圖形G，起點v_s)
拜訪v_s
對所有與v_s相鄰且未被拜訪的頂點v_x，重複迴圈
遞迴呼叫 DFS(v_x)
迴圈結束
演算法結束

在 DFS 遞迴演算法中，**遞迴呼叫**
相當於「深入」走訪，從呼叫返回
相當於 「回溯」到上一個頂點

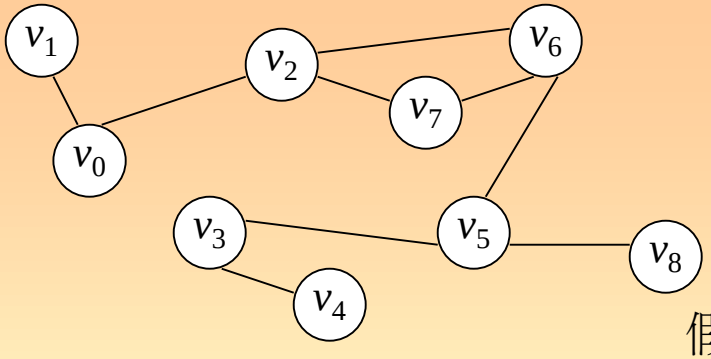
遞迴DFS走訪順序：
$v_0, v_1, v_2, v_6, v_5, v_3, v_4, v_8, v_7$。

DFS 演算法流程圖



DFS 演算法虛擬碼

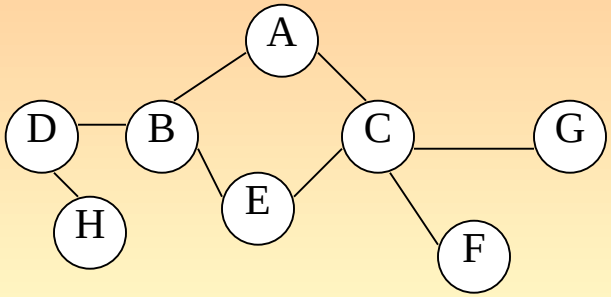
演算法：堆疊版深度優先走訪DFS (圖形G，起點v_s)
將v_s放入堆疊
當堆疊非空時，重複迴圈
從堆疊取出一頂點v_x
若v_x未被拜訪則
拜訪v_x，並將所有與v_x相鄰且未被拜訪的頂點放入堆疊
迴圈結束
演算法結束



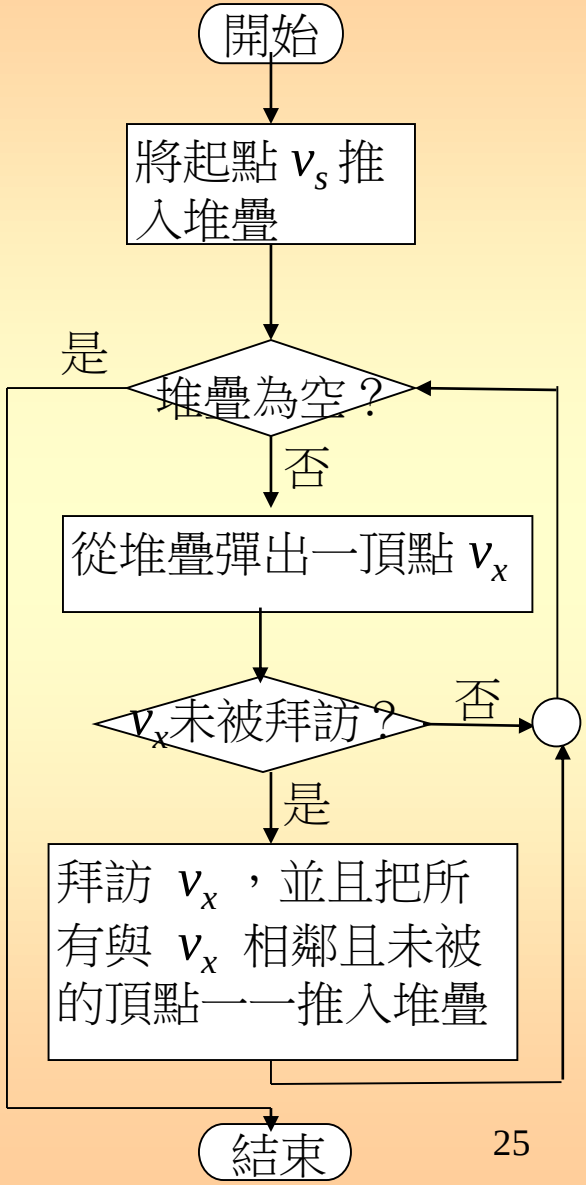
假設編號大的先進入堆疊

動作說明	堆疊 (stack)	已走訪頂點
0. 初始狀態	(top) (bottom)	
1. Push(v_0)	v_0	無
2. Pop得v_0, 拜訪v_0, 將所有與v_0相鄰且未被拜訪的頂點 (v_1, v_2) 一一Push	v_1, v_2	$\boldsymbol{v_0}$
3. Pop得v_1, 拜訪v_1, 沒有任何頂點與v_1相鄰且未被拜訪的	v_2	v_0, $\boldsymbol{v_1}$
4. Pop得v_2, 拜訪v_2, 將所有與v_2相鄰且未被拜訪的頂點 (v_6, v_7) 一一Push	v_6, v_7	v_0, v_1, $\boldsymbol{v_2}$
5. Pop得v_6, 拜訪v_6, 將所有與v_6相鄰且未被拜訪的頂點 (v_5, v_7) 一一Push	v_5, v_7, v_7	v_0, v_1, v_2, $\boldsymbol{v_6}$
6. Pop得v_5, 拜訪v_5, 將所有與v_5相鄰且未被拜訪的頂點 (v_3, v_8) 一一Push	v_3, v_8, v_7, v_7	v_0, v_1, v_2, v_6, $\boldsymbol{v_5}$
7. Pop得v_3, 拜訪v_3, 將所有與v_3相鄰且未被拜訪的頂點 (v_4) 一一Push	v_4, v_8, v_7, v_7	v_0, v_1, v_2, v_6, v_5, $\boldsymbol{v_3}$
8. Pop得v_4, 拜訪v_4, 沒有任何頂點與v_4相鄰且未被拜訪的	v_8, v_7, v_7	v_0, v_1, v_2, v_6, v_5, v_3, $\boldsymbol{v_4}$
9. Pop得v_8, 拜訪v_8, 沒有任何頂點與v_8相鄰且未被拜訪的	v_7, v_7	v_0,v_1,v_2,v_6,v_5,v_3,v_4,$\boldsymbol{v_8}$
10. Pop得v_7, 拜訪v_7, 沒有任何頂點與v_7相鄰且未被拜訪的	v_7	v_0,v_1,v_2,v_6,v_5,v_3,v_4,v_8,$\boldsymbol{v_7}$
11. Pop得v_7, v_7 已拜訪		v_0,v_1,v_2,v_6,v_5,v_3,v_4,v_8,v_7
12. 佇列已空, 停止		

例 5.5



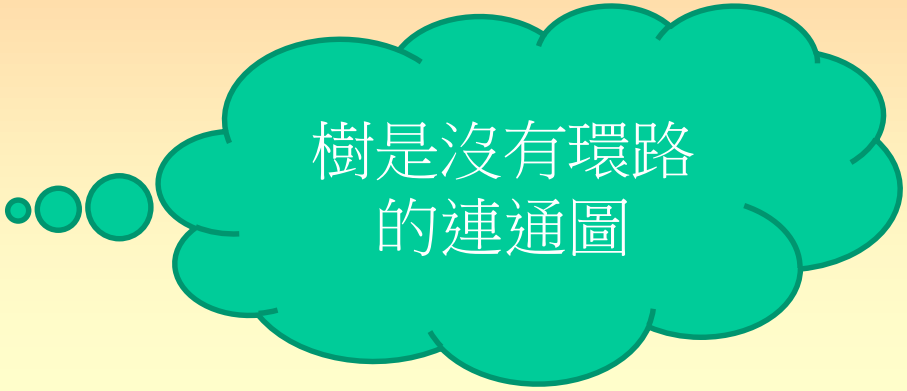
順序	堆疊內容	已拜訪節點
1	A	
2	CB	A
3	GFEB	AC
4	FEB	ACG
5	EB	ACGF
6	BB	ACGFE
7	DB	ACGFEB
8	HB	ACGFEBD
9	B	ACGFEBDH
10	空	ACGFEBDH

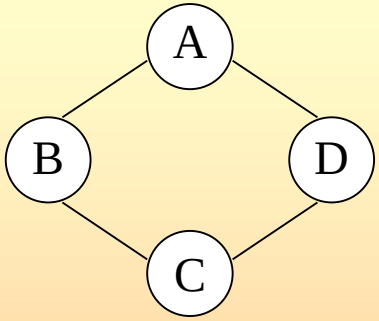


5-4 展開樹（Spanning Tree）

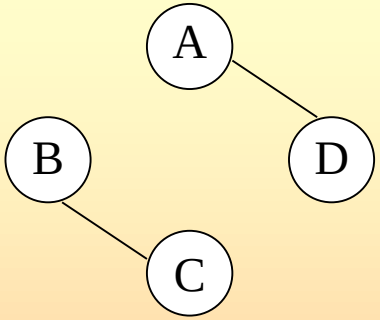
樹的定義：

1. 樹是一種圖形。
2. 樹是連通的。
3. 樹沒有環路。

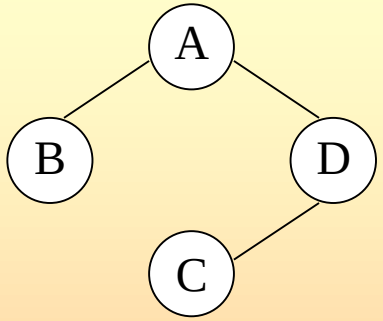




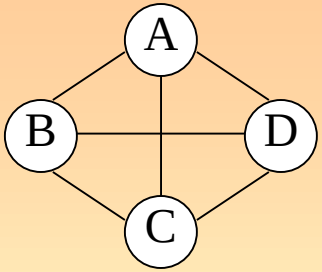
此圖有環路，不是樹



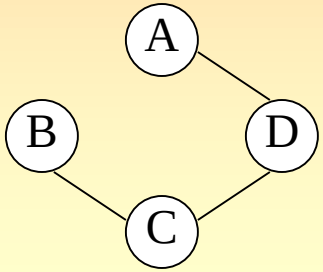
此圖不連通，不是樹



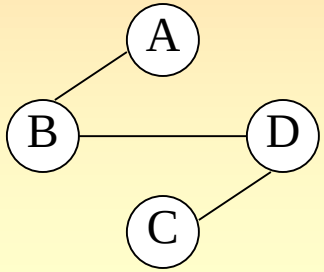
連通又沒有環路，是樹



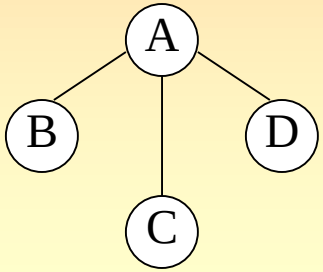
圖形G



展開樹 T1



展開樹 T2



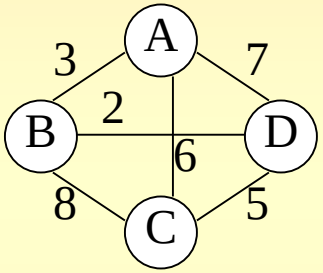
展開樹 T3

一個連通圖形 G 的**展開樹** T 定義為：

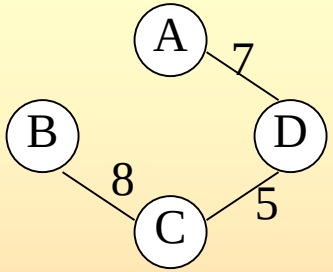
1. T 是 G 的子圖。
2. T 包含G 的所有**頂點**。(V(T) = V(G))
3. T是一棵樹。

Minimum Cost Spanning Tree

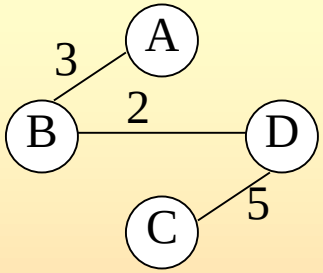
加權圖形 G 可能的眾多展開樹之中，會有一棵展開樹的加權和 (花費，cost) 是最小的，稱為**最少花費展開樹** (minimum cost spanning tree) 。最少花費展開樹可能多於一棵 。



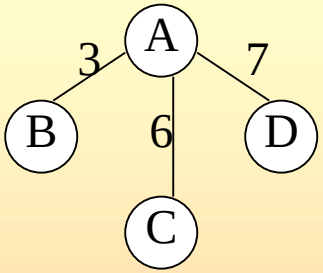
圖形G



展開樹T1
cost = 20

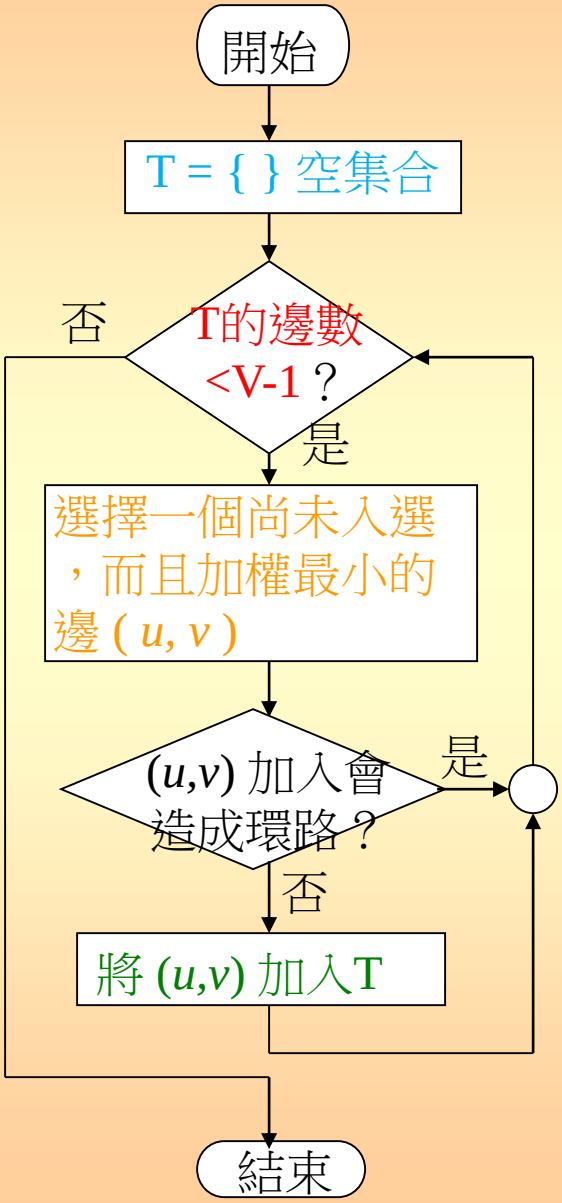


展開樹T2
cost = 10



展開樹T3
cost = 16

Kruskal 演算法流程圖



Kruskal 演算法虛擬碼

演算法：Kruskal法求出最少花費展開樹T (加權圖形G)

設定T為空集合

當 “T的邊數小於G的頂點數減1” 時，重複迴圈

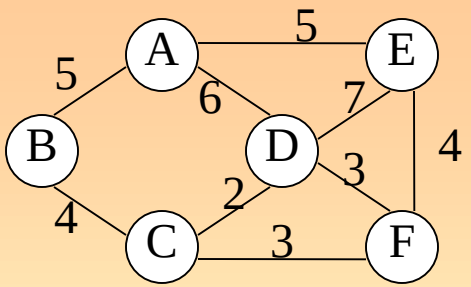
選擇一個尚未入選而且加權最小的邊 (*u*,*v*)

若 (*u*,*v*) 加入T不會造成環路則

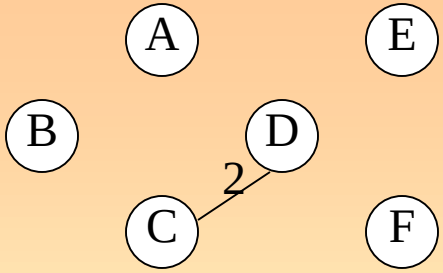
將 (*u*,*v*) 加入T

迴圈結束

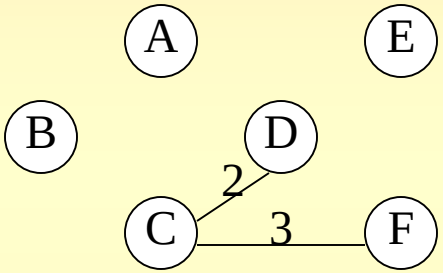
演算法結束



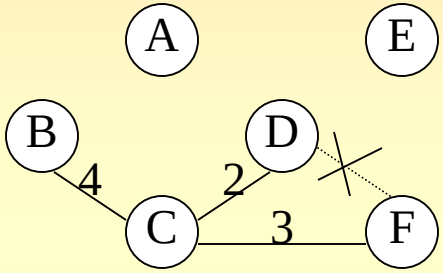
求此圖的最少花費展開樹



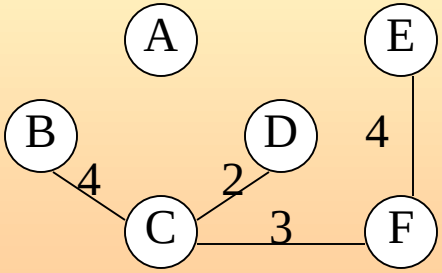
挑選邊 (C,D) 加入T



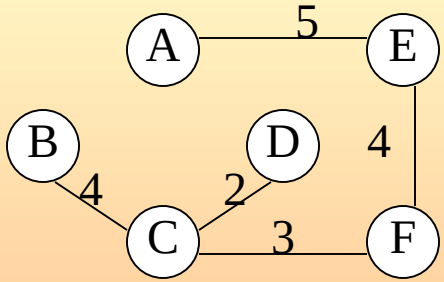
挑選邊 (C,F) 加入T。挑選邊 (D,F)亦可



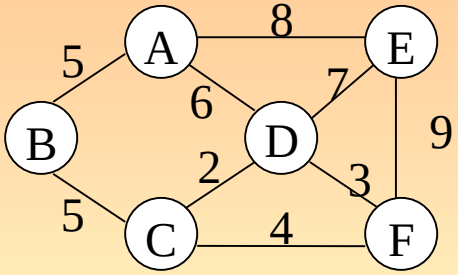
挑選邊 (B,C) 加入T。挑選邊 (E,F)亦可。
不可挑選邊 (D,F) 加入T，因為會造成環路



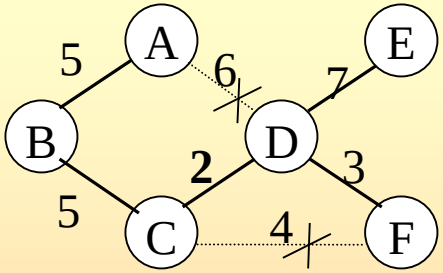
挑選邊 (E,F) 加入T



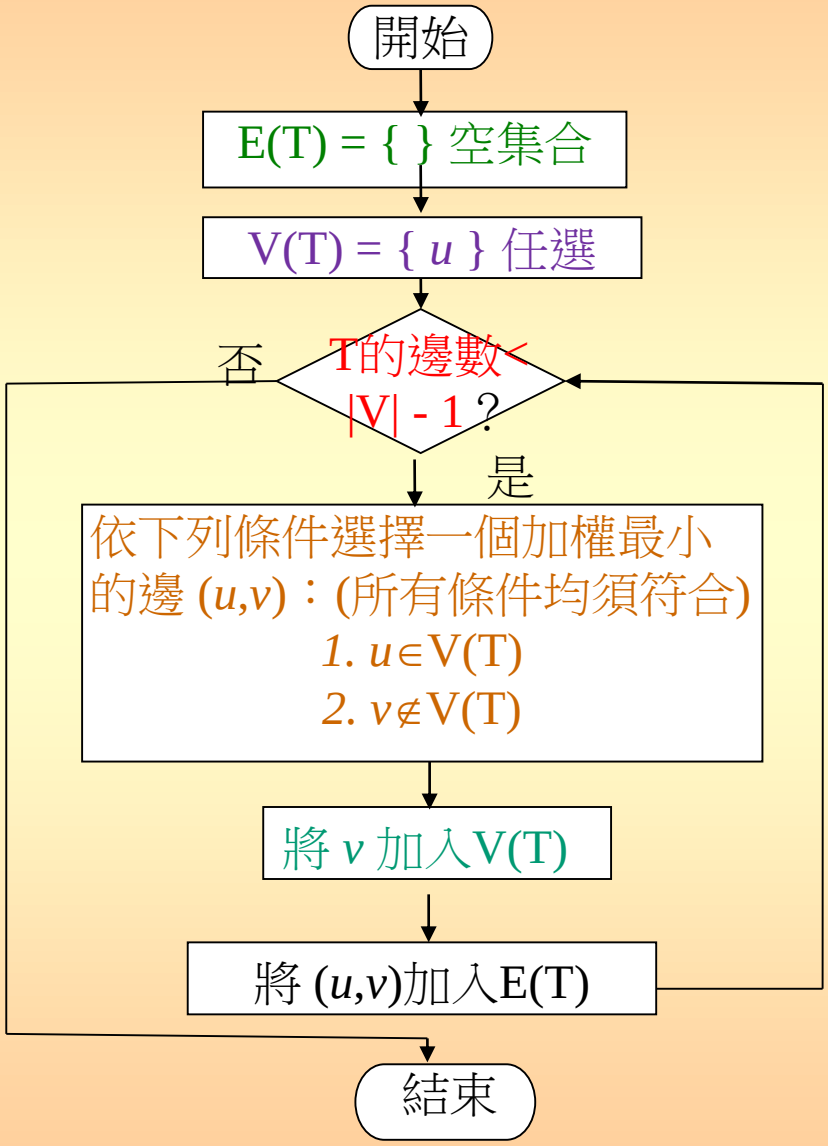
挑選邊 (A,E) 加入T。挑選邊 (A,B)亦可。此時邊數達頂點數減一，此樹即為最少花費展開樹



按照加權排序為 (C,D), (D,F), (C,F), (A,B), (B,C), (A,D), (D,E), (A,E), (E,F)



Prim 演算法流程圖



Prim 演算法虛擬碼

演算法：Prim法求出MCST (加權圖形G)

設定E(T) 為空集合

設定V(T) = { *u* } (任選)

當 “T的頂點數< G的頂點數”

時，重複迴圈

選擇一個加權最小的邊 (*u*,*v*)

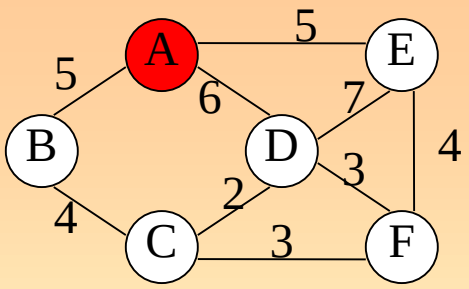
，*u*∈V(T) 且*v*∉V(T)

將 *v* 加入V(T)

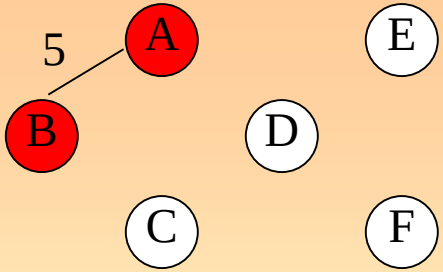
將 (*u*,*v*) 加入E(T)

迴圈結束

演算法結束

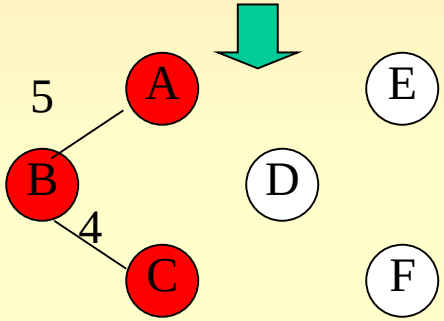


求此圖的最少花費展開樹

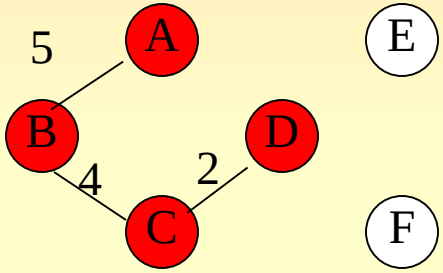


任選頂點A加入V(T)。

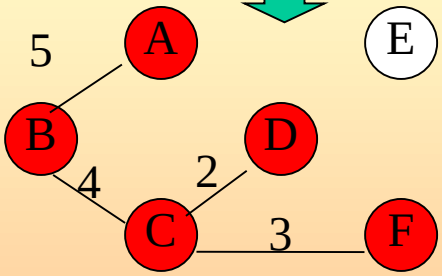
挑選邊 (A,B) 加入E(T) ，頂點B自然加入V(T)



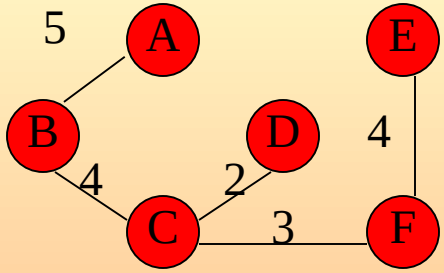
挑選邊 (B,C) 加入E(T)。
頂點C自然加入V(T)



挑選邊 (C,D) 加入E(T)。
頂點 D 自然加入V(T)

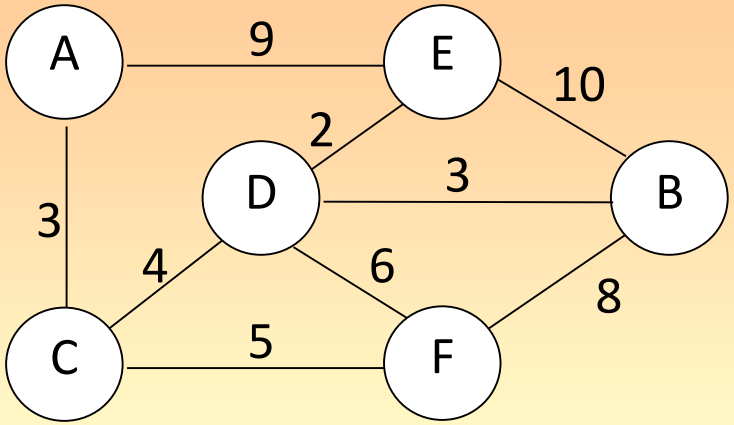


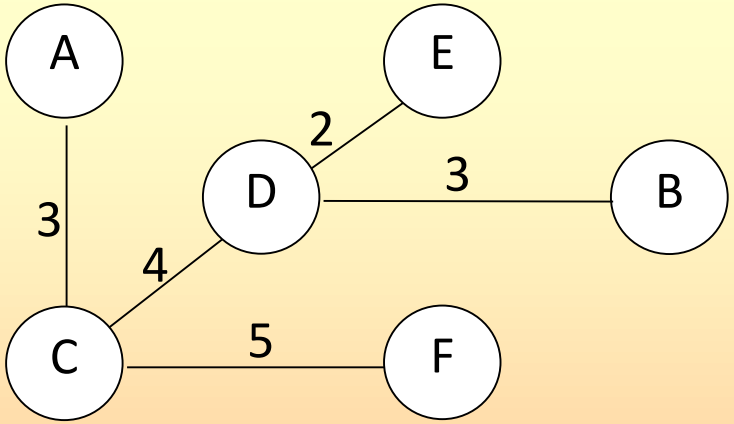
挑選邊 (C,F) 加入E(T)。
頂點 F 自然加入V(T)



挑選邊 (E, F) 加入T。頂點 F 自然加入V(T)

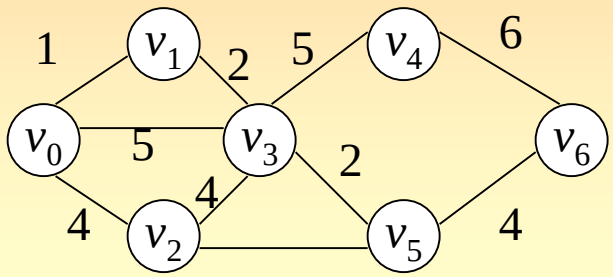
。此時包含所有頂點，此樹即為最少花費展開樹


A
9
E
10
2
D
3
B
3
4
6
8
C
5
F


A
E
2
D
3
B
3
4
C
5
F

5-5 最短路徑 (Shortest Path)

Dijkstra 演算法計算從某個起點至其餘各點的最短距離



$$\begin{pmatrix} 0 & 1 & 4 & 5 & \infty & \infty & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty & \infty \\ 4 & \infty & 0 & 4 & \infty & 3 & \infty \\ 5 & 2 & 4 & 0 & 5 & 2 & \infty \\ \infty & \infty & \infty & 5 & 0 & \infty & 6 \\ \infty & \infty & 3 & 2 & \infty & 0 & 4 \\ \infty & \infty & \infty & \infty & 6 & 4 & 0 \end{pmatrix}_{7 \times 7}$$

Dijkstra 演算法需要以下幾個資料結構：

1. Cost [V][V] ：圖形的加權鄰接矩陣，其中V為頂點數目

2. Dist [V] ：從起點到各點的最短距離。如果起點是V0，陣列Dist[] 初設為 Cost[0] (Cost[][] 陣列的第 0列)

3. Prior[V] ：各頂點最短路徑中的前一個頂點，初設每個元素都為V0 (起點)

4. Decided[V] ：各頂點最短路徑是否已被決定。初設除起點外每個元素均為 0

Dijkstra 演算法虛擬碼

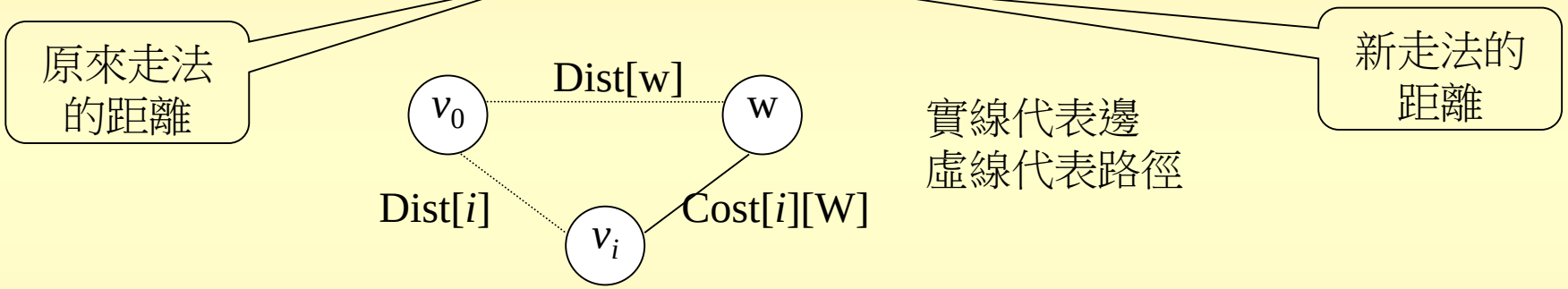
演算法：Dijkstra法求出由指定頂點v_s至其餘各點的最短距離(加權圖形G)
    設定v_s至其餘各點v_t的已知最短距離為邊(v_s,v_t)的加權
        (若v_s與v_t不相鄰則為∞)
    設定除v_s之外的各點為「最短距離未定點」
    當“尚有頂點的最短距離未定” 時，重複迴圈
        從未定頂點中選擇一個已知距離最小的點v_x
        將v_x設為「最短距離已定點」
        逐一測試所有未定點v_t是否會因為v_x而變近
            若有變近則更新v_t的已知最短距離
    迴圈結束
演算法結束

演算法步驟

1. 找到一個頂點 i，其 Decided[i] = 0 (最短距離尚未決定)且 Dist[i] (已知距離)最小，若Decided[i]均為 1 則跳至步驟 4.

2. 對每一個 Decided[w] = 0 的頂點 w ，依下述方式更新 Dist[w]

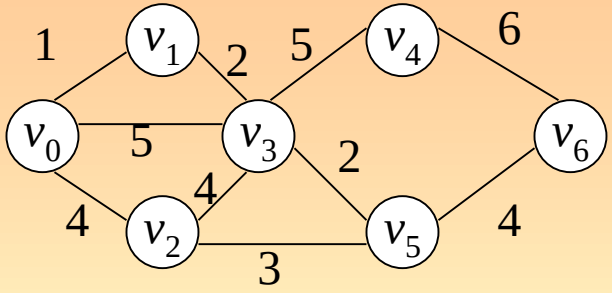
Dist[w] = min { Dist[w], Dist[i] + Cost[i][w] }



從 v_0 出發，如果經由 v_i 再到 w 比原來由 v_0 到 w 更近，則採用新的走法

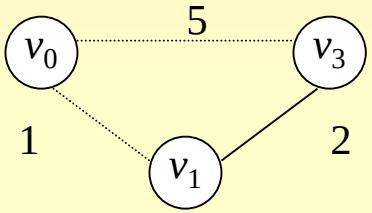
3. 跳至步驟 1.

4. 演算法結束。每個 Dist[k] 為起點至頂點 k 的最短距離



	[0]	[1]	[2]	[3]	[4]	[5]	[6]
Dist[]	0	1	4	5	∞	∞	∞
Prior[]	0	0	0	0	0	0	0
Decided[]	1	0	0	0	0	0	0

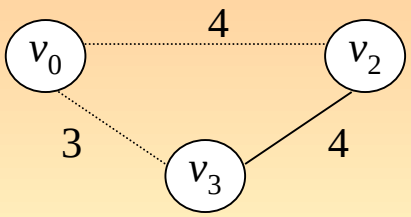
階段1 (選定 v_1為已定頂點，檢查與 v_1相鄰的未定點可否因 v_1而變近)



Dist[**3**] = min { Dist[**3**] , Dist [1] + Cost[1][**3**] }
= min { 5 , 1 + 2 } = 3

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
Dist[]	0	1	4	**3**	∞	∞	∞
Prior[]	0	0	0	**1**	0	0	0
Decided[]	1	**1**	0	0	0	0	0

階段2 (選定 v_3為已定頂點，檢查與 v_3相鄰的未定點可否因 v_3而變近)

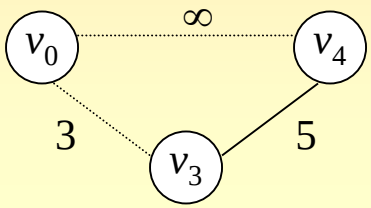


以 v_2 扮演W：

Dist[**2**] = min {Dist[**2**] , Dist[3]+Cost[3][**2**]}

= min {4 , 3 + 4}

= 4（不變 ）

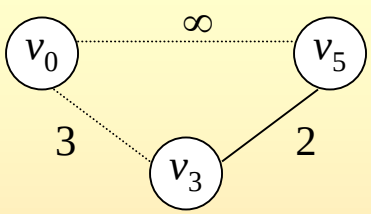


以 v_4 扮演W：

Dist[**4**] = min {Dist[**4**] , Dist[3]+Cost[3][**4**]}

= min {∞ , 3 + 5}

= 8（v_3使v_4變近，v_4的前一站設為v_3， Prior[4] 設為3）



以 v_5 扮演W：

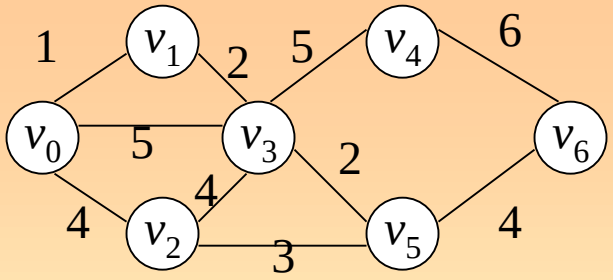
Dist[**5**] = min {Dist[**5**] , Dist[3]+Cost[3][**5**]}

= min {∞ , 3 + 2}

= 5（v_3使v_5變近，v_5的前一站設為v_3， Prior[5] 設為3）

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
Dist[]	0	1	4	3	**8**	**5**	∞
Prior[]	0	0	0	1	**3**	**3**	0
Decided[]	1	1	0	**1**	0	0	0

階段 3 選定v_2
階段 4 選定v_5
階段 5 選定v_4
階段 6 選定v_6



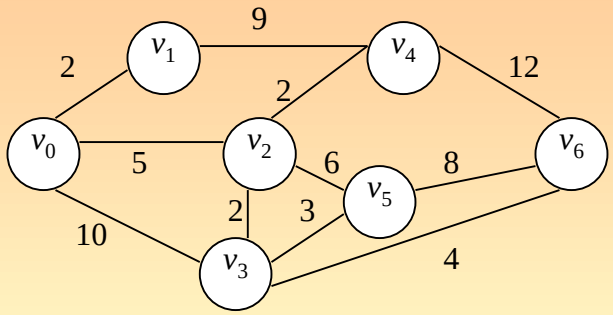
最後可得：

方格內為 Dist[] / Prior[]

Decided vertices	v_1	v_2	v_3	v_4	v_5	v_6
{ v_0}	**1/0**	4/0	5/0	∞/0	∞/0	∞/0
{ v_0, v_1}	|	4/0	**3/1**	∞/0	∞/0	∞/0
{ v_0, v_1, v_3}	|	**4/0**	|	8/3	5/3	∞/0
{ v_0, v_1, v_3, v_2 }	|	|	|	8/3	**5/3**	∞/0
{ v_0, v_1, v_3, v_2, v_5 }	|	|	|	**8/3**	|	9/5
{v_0, v_1,v_3,v_2,v_5, v_4 }	|	|	|	|	|	**9/5**

由v_4那一欄的值 (8/3) 可以知道，從v_0到v_4的最短距離是8，而它的前一站是v_3。

例 5.8 求出下圖中，從頂點v_0到各頂點的最短路徑及最短距離。

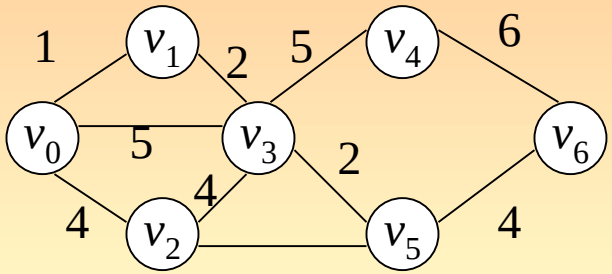


作表可得：

Decided vertices	v_1	v_2	v_3	v_4	v_5	v_6
{ v_0}	**2/0**	5/0	10/0	∞/0	∞/0	∞/0
{ v_0, v_1}	|	**5/0**	10/0	11/1	∞/0	∞/0
{ v_0, v_1, v_2}	|	|	**7/2**	7/2	11/2	∞/0
{ v_0, v_1, v_2, v_3 }	|	|	|	**7/2**	10/3	11/3
{ v_0, v_1, v_2, v_3, v_4 }	|	|	|	|	**10/3**	11/3
{v_0, v_1,v_2,v_3,v_4, v_5 }	|	|	|	|	|	**11/3**

由 v_6 欄的最終值 (11 / 3) 可以看出：
v_0 到 v_6 的最短距離為：11，
v_0 到 v_6 的最短路徑由後往前推為：$v_6 \leftarrow v_3 \leftarrow v_2 \leftarrow v_0$，
亦即 $v_0 \rightarrow v_2 \rightarrow v_3 \rightarrow v_6$。

Floyd演算法計算每一對頂點間的最短距離



A^{-1}=

	v_0	v_1	v_2	v_3	v_4	v_5	v_6
v_0	0	1	4	5	∞	∞	∞
v_1	1	0	∞	2	∞	∞	∞
v_2	4	∞	0	4	∞	3	∞
v_3	5	2	4	0	5	2	∞
v_4	∞	∞	∞	5	0	∞	6
v_5	∞	∞	3	2	∞	0	4
v_6	∞	∞	∞	∞	6	4	0

7×7

將右邊矩陣經下式計算得到中間矩陣：

$$A^0[i][j] = \min \{ A^{-1}[i][j] , A^{-1}[i][\mathbf{0}] + A^{-1}[\mathbf{0}][j] \}$$

也就是若原來 v_i 到 v_j 的距離比經由 v_0 的距離更遠，則採用經由 v_0 的新距離，否則不變。

v_1 到 v_2 的走法經由 v_0 變近了

A^0=

	v_0	v_1	v_2	v_3	v_4	v_5	v_6
v_0	0	1	4	5	∞	∞	∞
v_1	1	0	**5**	2	∞	∞	∞
v_2	4	**5**	0	4	∞	3	∞
v_3	5	2	4	0	5	2	∞
v_4	∞	∞	∞	5	0	∞	6
v_5	∞	∞	3	2	∞	0	4
v_6	∞	∞	∞	∞	6	4	0

7×7

k 從 0 到 6 套用下列通式，可得結果矩陣

$$A^{k}[i][j] = \min \{ A^{k-1}[i][j] , A^{k-1}[i][\boldsymbol{k}] + A^{k-1}[\boldsymbol{k}][j] \}$$

也就是若原來v_i到v_j的距離比經由v_k的距離更遠，則採用經由v_k的新距離，否則不變。

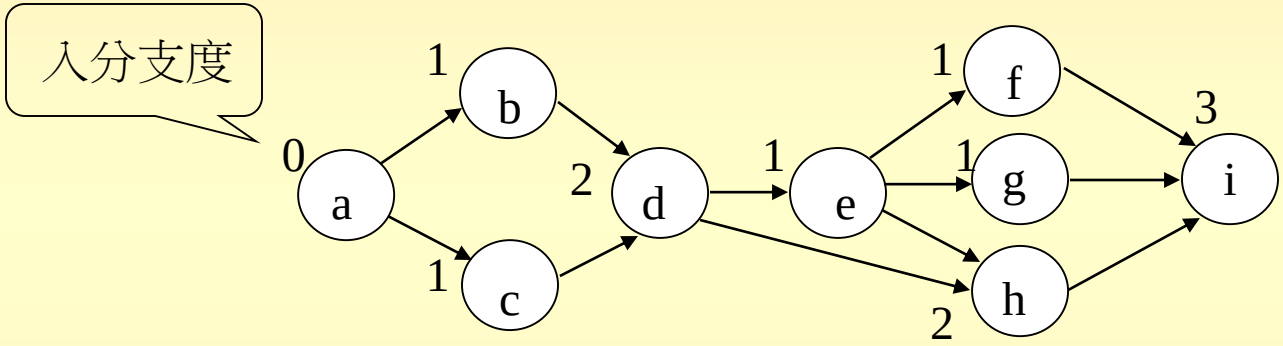
$A^6=$

	v_0	v_1	v_2	v_3	v_4	v_5	v_6
v_0	0	1	4	3	8	5	9
v_1	1	0	5	2	7	4	8
v_2	4	5	0	4	9	3	7
v_3	3	2	4	0	5	2	6
v_4	8	7	9	5	0	7	6
v_5	5	4	3	2	7	0	4
v_6	9	8	7	6	6	4	0

7×7

5-6 拓樸排序 (Topological Sorting)

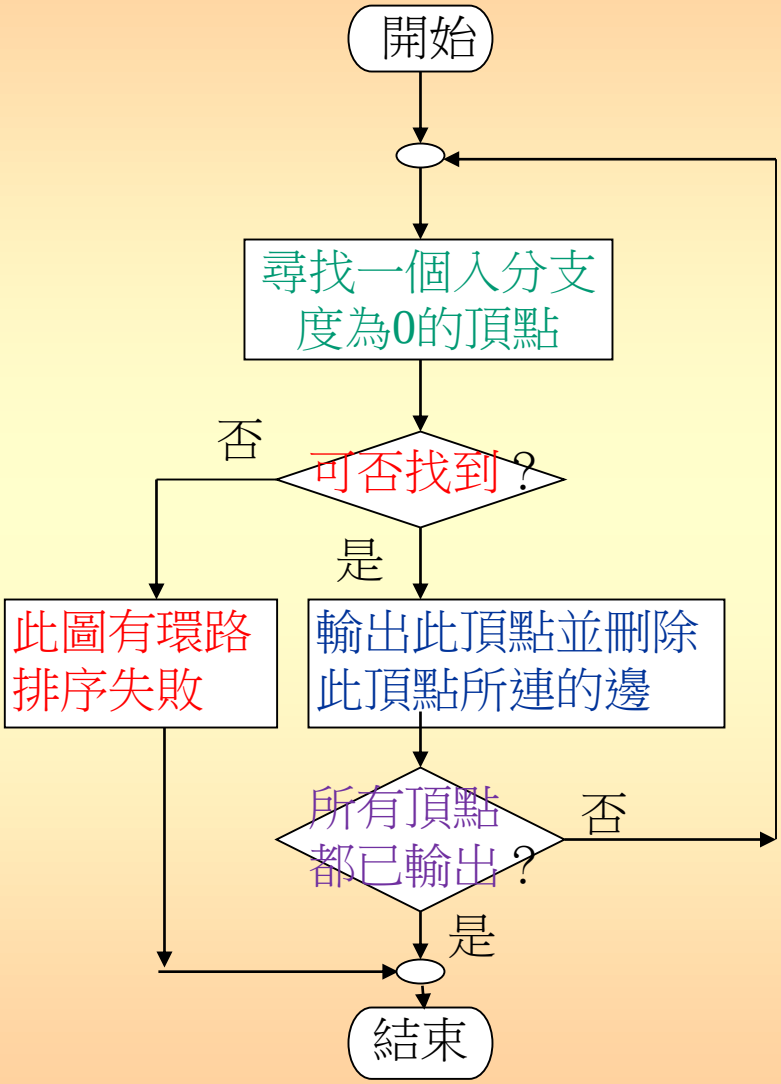
如果用有向圖形來表示事件發生的**先後順序限制**，**頂點**表示工作項目，以邊表示工作之間的先後關係，我們稱這種有向圖形為
頂點工作網路 (Activity On Vertex Network, AOV網路)



一開始就可以執行的工作，就是完全沒有**前行者** (入分支度為 0) 的工作，例如上圖的頂點 a 。

頂點 a 執行後，它的立即後繼者 b, c 就可以接著執行了（a 對它們的限制已經解除了），以這個方式繼續執行下去即可得到此AOV的拓樸順序

拓樸排序演算法流程圖



拓樸排序演算法虛擬碼

演算法：拓樸排序演算法輸出頂點順序(AOV網路)

重複迴圈

尋找一個入分支度為0的頂點

如果找不到則

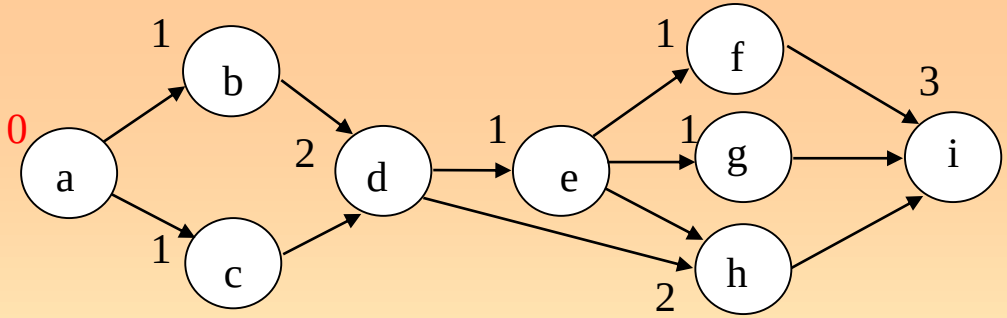
此圖有環路，排序失敗

否則

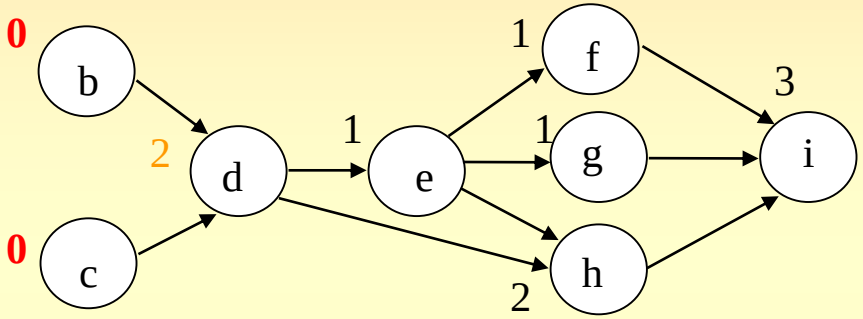
輸出此頂點，並刪除此頂點所連的所有邊

當所有頂點都已輸出時，迴圈結束

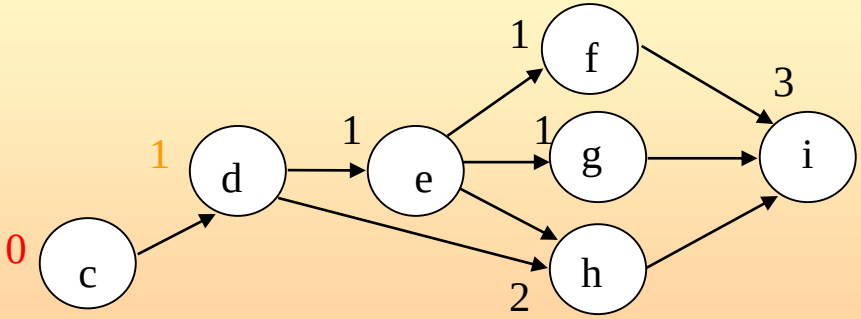
演算法結束



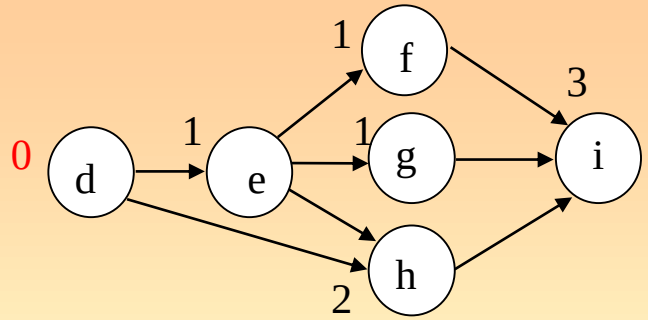
輸出： a（將b、c的入分支度減1）



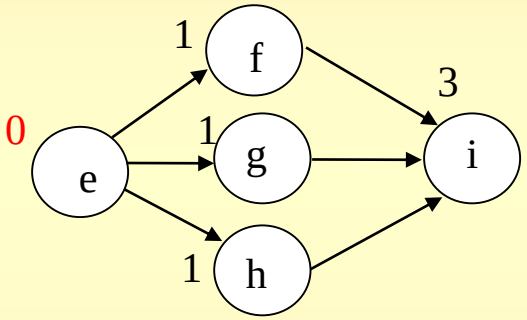
輸出： b（將d的入分支度減1，在此輸出 c 亦可）



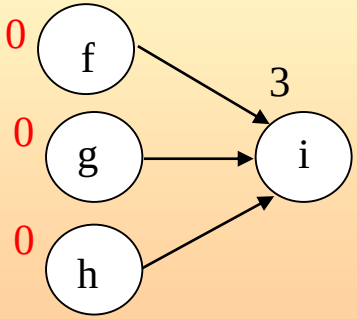
輸出： c（將d的入分支度減1）



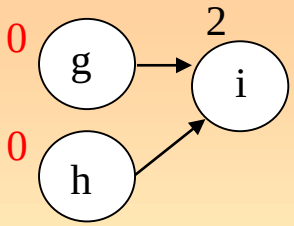
輸出： d（將e、h的入分支度減1）



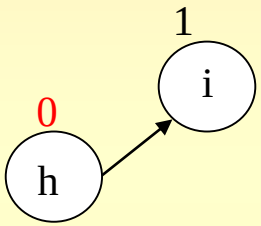
輸出 e（將f、g、h的入分支度減1）



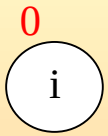
輸出 f（輸出 g, h亦可，不管是哪一個，都將i的入分支度減1）



輸出 g（輸出 h亦可，將i的入分支度減1）



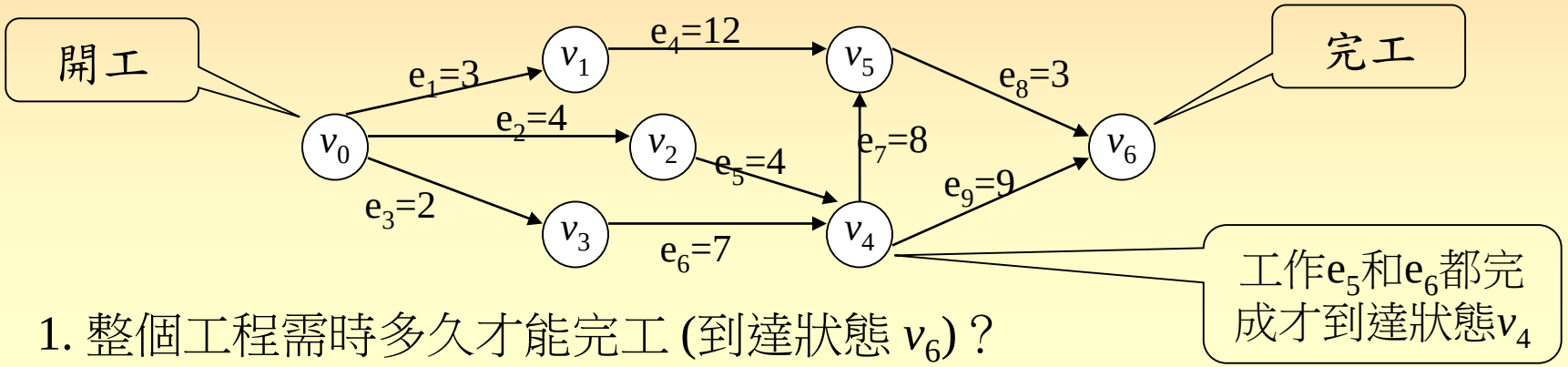
輸出 h（將i的入分支度減1）



輸出 i，結束排序

5-7 關鍵路徑 (Critical Path)

邊工作網路 (Activity On Edge Network, AOE網路)，
以頂點代表狀態，而邊才代表工作。



1. 整個工程需時多久才能完工 (到達狀態 v_6)？
2. 什麼情況下會提前或延後工期？

需要計算下列四種值

etv ：earliest time of vertex，頂點最早時間。

ltv ：latest time of vertex，頂點最晚時間，超出此時間將會延誤整個工期。

ete ：earliest time of edge，邊（工作）最早時間。

lte ：latest time of edge，工作最晚時間，也就是每個工作最晚**必須**開始進行的時間。超出此時間將會延誤整個工期。

關鍵路徑虛擬碼

演算法：計算一個AOE網路的關鍵路徑

- 計算etv

  設定「源點」(沒有前行者的頂點)的etv為0

  按照頂點的拓樸排序順序，逐一計算每個頂點v_i的etv

  etv[v_i] = 最大值{ v_i所有前行者v_j的etv[v_j] + (v_i, v_j)的邊長}

- 計算ltv

  設定「終點」(沒有後繼者的頂點)的ltv等於其etv

  按照頂點的反拓樸順序，逐一計算頂點v_i的ltv

  ltv[v_i] = 最小值{ v_i所有後繼者v_j的ltv[v_j] – (v_i, v_j)的邊長}

- 計算lte與ete

  對每一個邊 e_k = (v_i, v_j)：

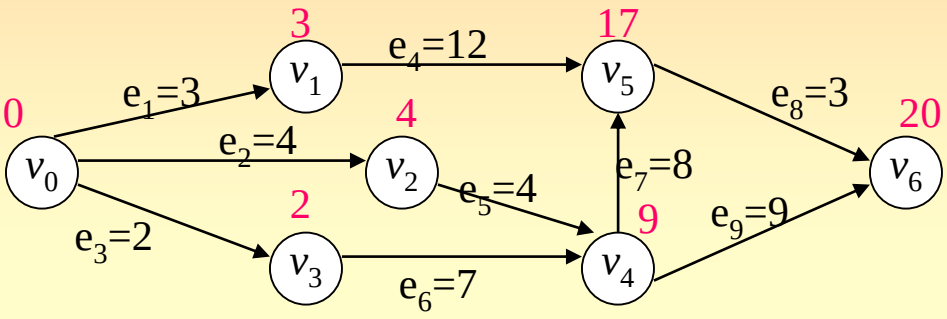
  ete[e_k] = etv[v_i] 且 lte[e_k] = ltv[v_j] – e_k的邊長

- 所有lte等於ete的邊即會構成關鍵路徑

演算法結束

1. 首先，我們按照拓樸順序一個一個決定頂點 v 的 etv。取 v 所有前行頂點的 (etv + 邊長) 的最大值。

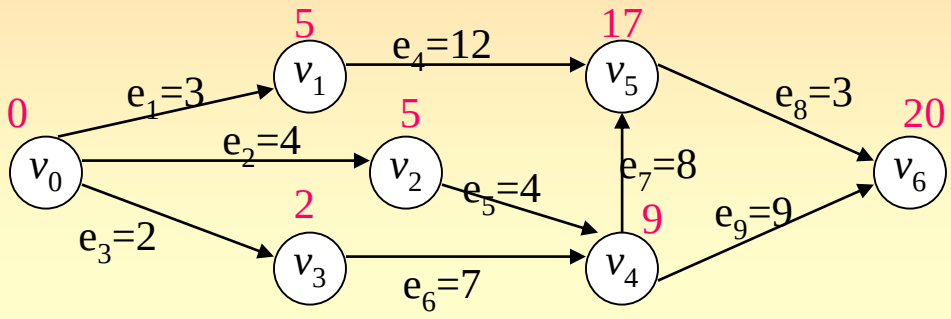
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
etv[]	0	3	4	2	9	17	20



1. etv[0] = 0，v_0 在第 0 年可開始，一宣告開工即進入此狀態。
2. etv[1] = etv[0] + $|e_1|$ = 0 + 3 = 3， v_0 是 v_1 的前行頂點，而且工作 e_1需時3年。
3. etv[2] = etv[0] + $|e_2|$ = 0 + 4 = 4， v_0 是 v_2 的前行頂點，因為工作e_2需時4年。
4. etv[3] = etv[0] + $|e_3|$ = 0 + 2 = 2， v_0 是 v_3 的前行頂點，因為工作e_3需時2年。
5. etv[4] = max { etv[2] + $|e_5|$, etv[3] + $|e_6|$ } = max {4+4, 2+7 } = 9，雖然第4年後狀態v_2成立，可以開始進行工作e_5，並於第8年可完成。但是另一隊人馬在第2年從狀態v_3開始進行工作e_6，要到第9年始可完成。必須等到兩個工作都完成後，狀態v_4才算成立，取最大值的原因在此。
6. etv[5] = max { etv[1] + $|e_4|$, etv[4] + $|e_7|$ } = max { 3+12, 9+8 } = 17，v_5的入分支度為2，必須取兩者的最大值，v_5的旗子才能舉起。
7. etv[6] = max { etv[5] + $|e_8|$, etv[4] + $|e_9|$ } = max {17+3, 9+9 } = 20。

2. 頂點的最晚必須開始時間 (ltv)則必須按照反拓樸順序，由後往前推算

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
ltv[]	0	5	5	2	9	17	20



1.ltv[6] = etv[6] = 20，因為我們是以不延誤工期為原則，所以理所當然要把它的最晚時間與本身的最早時間設為相等。

2. ltv[5] = ltv[6] – |e_8| = 20–3 = 17，因為最晚20年必須到達v_6，而工作 e_8需時3年，因此最晚第17年必須由 v_5 出發，否則將會延誤工期。

3. ltv[4] = min { ltv[6] – |e_9|, ltv[5] – |e_7| } = min { 20 – 9, 17 – 8 } = 9。取最小值的原因在於要適應最早的工作，以免它延誤。

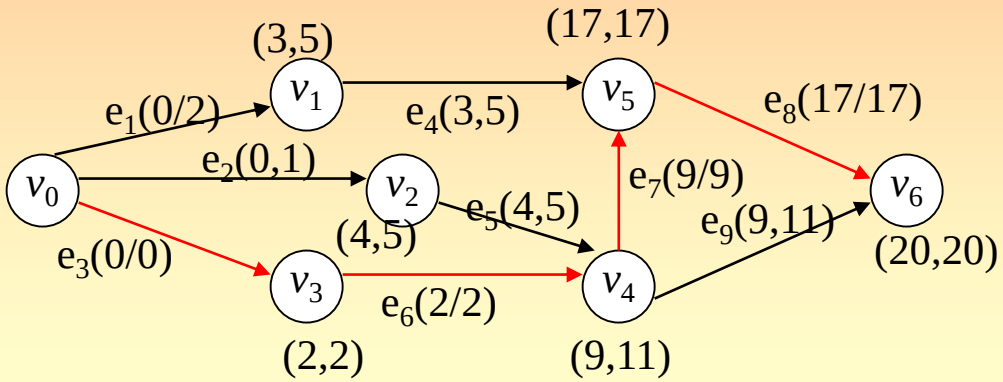
4. ltv[3] = ltv[4] – |e_6| = 9 – 7 = 2。

5. ltv[2] = ltv[4] – |e_5| = 9 – 4 = 5。

6. ltv[1] = ltv[5] – |e_4| = 17 – 12 = 5。

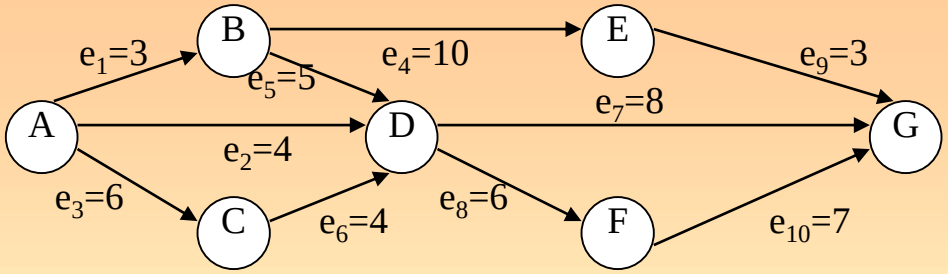
7. ltv[0] = min { ltv[1] – |e_1|, ltv[2] – |e_2|, ltv[3] – |e_3|} = min { 5 – 3, 5 – 4, 2 – 2 } = 0。

3. 推算出「工作最早時間」(ete) 和「工作最晚時間」(lte)。推算的原則是：一個工作的最早時間，取此工作(邊)之起點的最早時間，而最晚時間取終點的最晚時間減去邊長。



ete[1] = etv[0] = 0 (e_1的起點是v_0) lte[1] = ltv[1] - $|e_1|$ = 5 – 3 = 2 (e_1的終點是v_1)
ete[2] = etv[0] = 0 (e_2的起點是v_0) lte[2] = ltv[2] - $|e_2|$ = 5 – 4 = 1 (e_2的終點是v_2)
ete[3] = etv[0] = 0 (e_3的起點是v_0) lte[3] = ltv[3] - $|e_3|$ = 2 – 2 = 0 (e_3的終點是v_3)
ete[4] = etv[1] = 3 (e_4的起點是v_1) lte[4] = ltv[5] - $|e_4|$ = 17 – 12 = 5 (e_4的終點是v_5)
ete[5] = etv[2] = 4 (e_5的起點是v_2) lte[5] = ltv[4] - $|e_5|$ = 9 – 4 = 5 (e_5的終點是v_4)
ete[6] = etv[3] = 2 (e_6的起點是v_3) lte[6] = ltv[4] - $|e_6|$ = 9 – 7 = 2 (e_6的終點是v_4)
ete[7] = etv[4] = 9 (e_7的起點是v_4) lte[7] = ltv[5] - $|e_7|$ = 17 – 8 = 9 (e_7的終點是v_5)
ete[8] = etv[5] = 17 (e_8的起點是v_5) lte[8] = ltv[6] - $|e_8|$ = 20 – 3 = 17 (e_8的終點是v_6)
ete[9] = etv[4] = 9 (e_9的起點是v_4) lte[9] = ltv[6] - $|e_9|$ = 20 – 9 = 11 (e_9的終點是v_6)

- 有一條路徑，上面的邊其最晚時間 (lte) 都等於最早時間 (ete)，稱為「關鍵路徑」(critical path)。e_3, e_6, e_7, e_8 ($v_0 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6$)



先算出每個頂點的最早時間和最晚時間。

頂　點	A	B	C	D	E	F	G
最早時間	0	3	6	10	13	16	23
最晚時間	0	5	6	10	20	16	23

因此最快到達頂點G的時間為23。

再算出每個工作的最早時間和最晚時間。

工　作	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}
最早時間	0	0	0	3	3	6	10	10	13	16
最晚時間	2	6	0	10	5	6	15	10	20	16

最早時間和最晚時間相等的，即為關鍵路徑上的邊。
因此關鍵路徑為：
e_3, e_6, e_8, e_{10} ($A \rightarrow C \rightarrow D \rightarrow F \rightarrow G$)