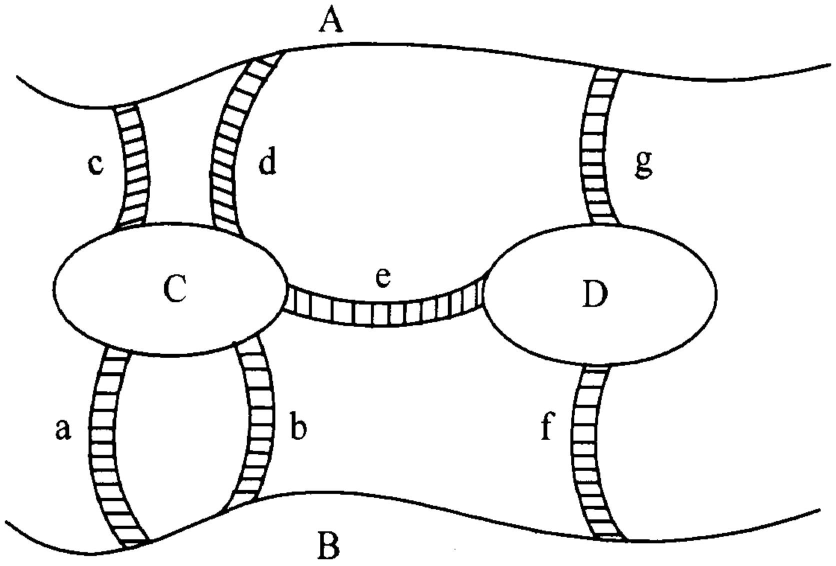
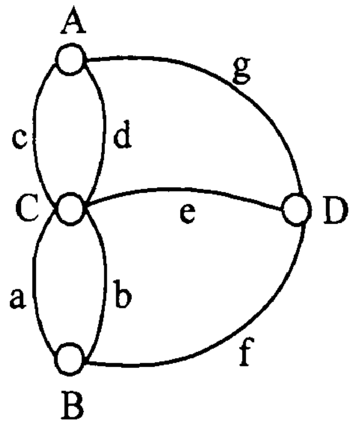
Chapter 06 圖形結構 (Graphs)

圖形結構

- 尤拉問題：是否可以從某城市開始走，然後走遍全部的橋，再回到原先的起始城市？

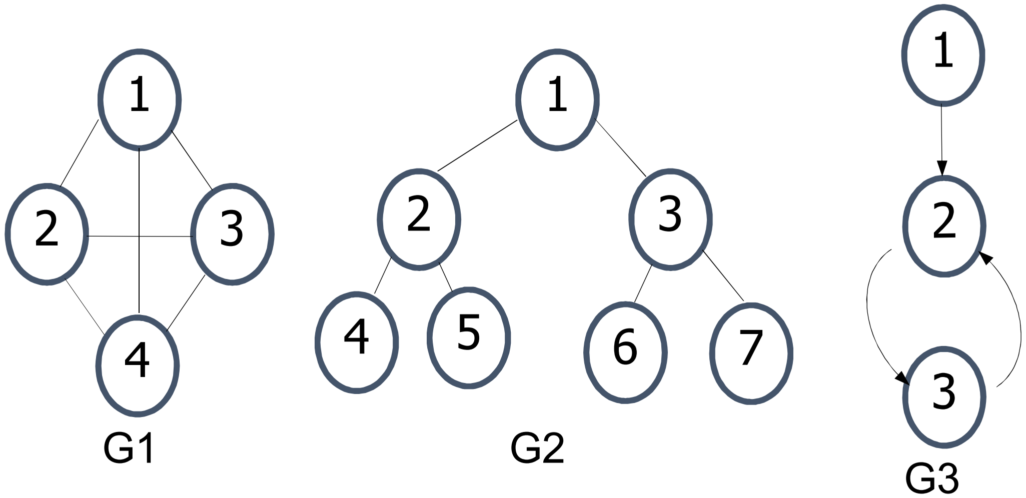




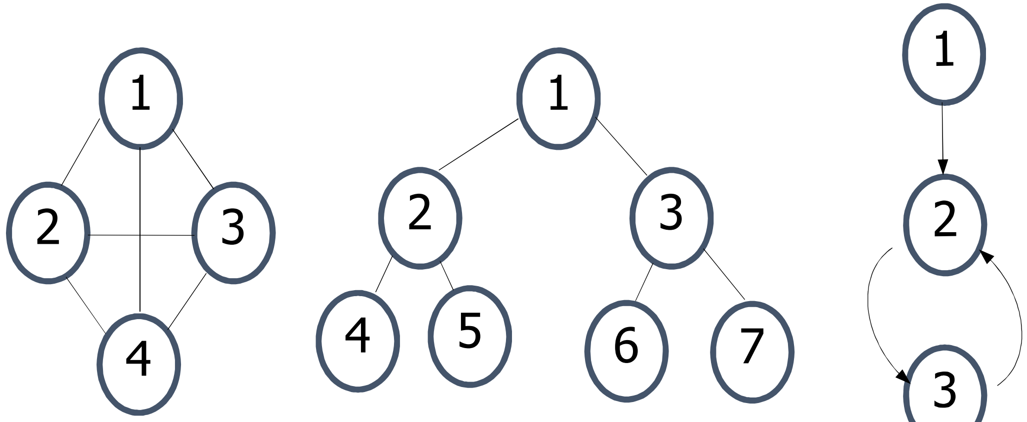
- 圖形結構
  - 圓圈為「頂點」(vertex)。
  - 連線為「分支度」(degree)；Ex：節點c的分支度為5。
- 假使尤拉問題要能成立的話，必須每個頂點具備偶數的分支度方可，此稱為尤拉循環(Eulerian cycle) 。

圖形的一些專有名詞(1)

- 無方向圖形(undirected graph)在邊上沒有箭頭者稱之，如: G1, G2。
- 有方向圖形(directed graph)：在邊上有箭頭者稱之，如: G3。



圖形的一些專有名詞(2)

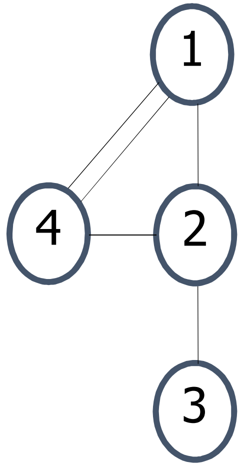


G1 G2 G3

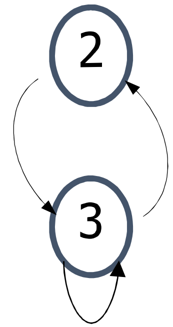
- 頂點(vertex)：上圖的圓圈稱之。
  - V(G1) = {1,2,3,4},
  - V(G2) = {1,2,3,4,5,6,7},
  - V(G3) = {1,2,3}
- 邊(edge)：上圖每個頂點之間的連線稱之。
  - E(G1) = {(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)}
  - E(G2) = {(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)}
  - E(G3) = {<1,2>,<2,3>,<3,2>}
- 圖形(graph)：是由所有頂點和所有邊組合而成的，以 G=(V, E)表示。

圖形的一些專有名詞(3)

- 限制
  - 多邊圖形（mutigraph）：假使兩個頂點間，有多條相同的邊此稱之為多重圖形，而不是圖形。如：G4。
  - 自身邊圖形(graph with self edges)：有self edges 的圖形稱之。
    - Self edge or self loop : 連到本身的邊稱之，如G3的<3,3>



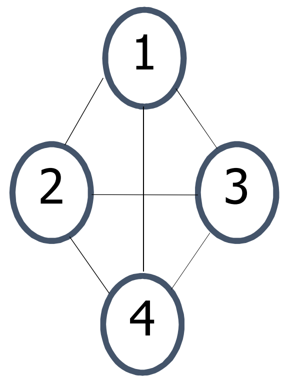
G4



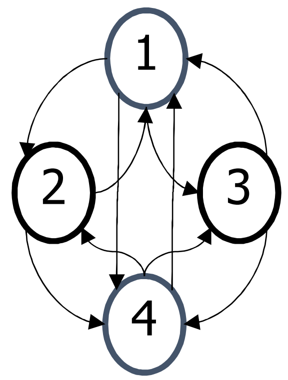
G3

圖形的一些專有名詞(4)

- 完整圖形（complete graph）：邊的個數最大者。在n個頂點的無方向圖形中，會有 n(n-1)/2個邊，如:G1。有方向圖形則有n(n-1)個邊，如G2圖。



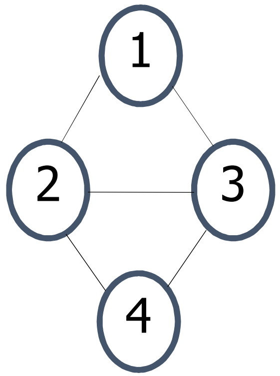
G1



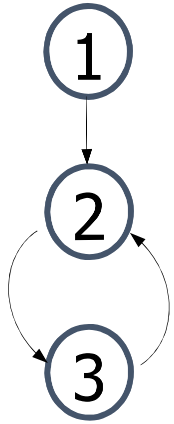
G2

圖形的一些專有名詞(5)

- 相鄰(adjacent)：在圖形的某一邊 (V_1, V_2) 中，我們稱頂點V_1與頂點V_2是相鄰的。如G1中頂點1相鄰的是2,3
- 連接(incident)：我們稱頂點V_1和頂點V_2是相鄰，而邊 (V_1, V_2) 是連接在頂點V_1與V_2頂點上。如G1中連接2這個頂點的邊有(1,2)、(2,3)、(3,4)
- 在有方向圖形中，如果$<V_1, V_2>$是一個有向邊，我們說V_1相鄰到(adjacent to) V_2 ，而稱V_2從V_1相鄰(adjacent from)。如:G3的 <1, 2>，2 adjacent from 1 而 1 adjacent to 2 。和2這個頂點相連的有<1,2>、<2,3>、<3,2>。



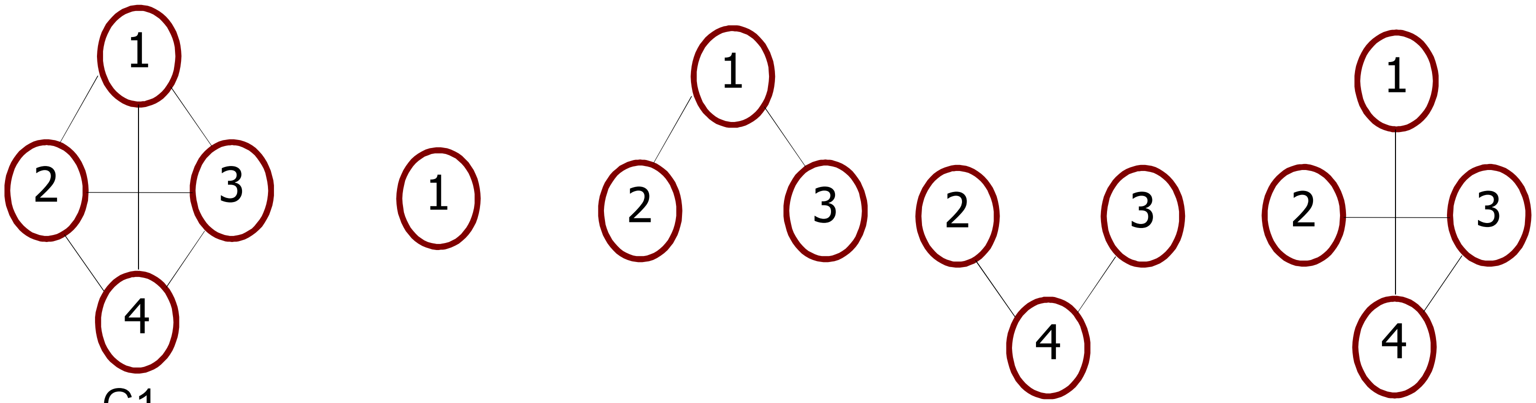
G1



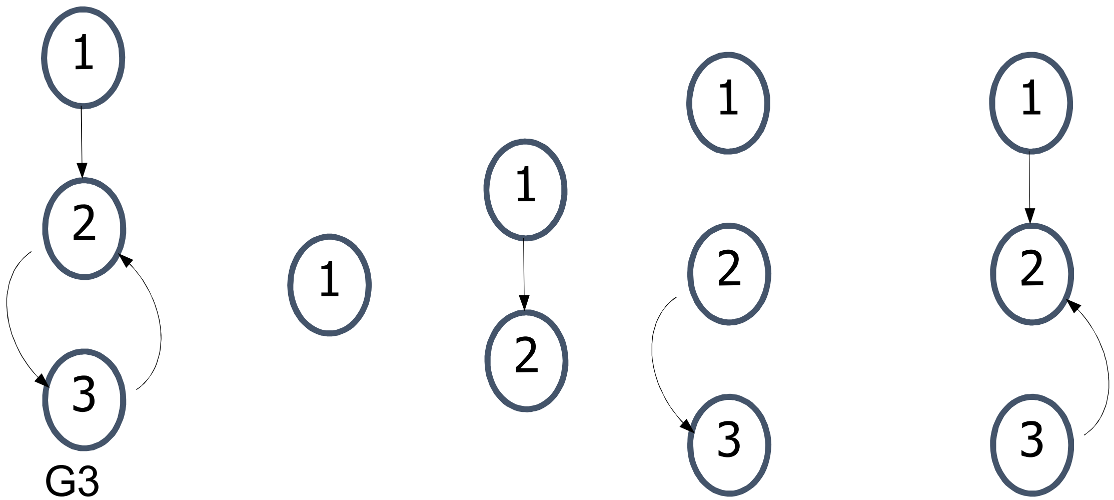
G3

圖形的一些專有名詞(6)

- 子圖(subgraph)：假使V(G’)是V(G)的部份集合及E(G’)是E(G)的部份集合，我們稱G'是G的子圖。
  - V(G’) $\in$ V(G) and E(G’) $\in$ E(G)



G1

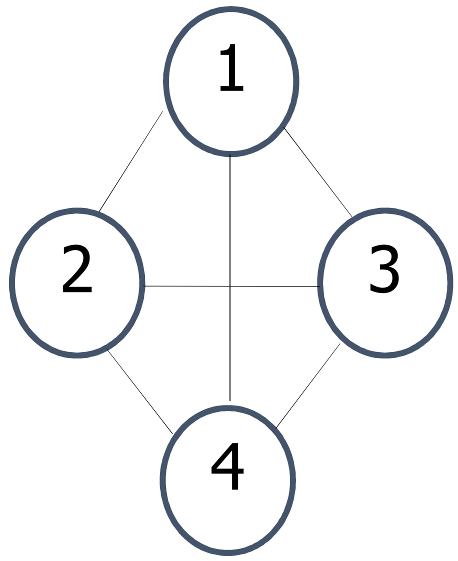


G3

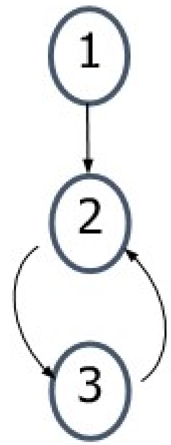
圖形的一些專有名詞(7)

- 路徑(path)：在圖形G中，從頂V_p到頂點V_q的路徑是指一系列的頂點V_p, V_{i1}, V_{i2},……, V_{in}, V_q ，其中(V_p,V_{i1}),(V_{i1},V_{i2}),……, (V_{in},V_q)是E(G)上的邊。
  - 如果G是有向圖，則該路徑為$<V_p,V_{i1}>$,$<V_{i1},V_{i2}>$,……, $<V_{in},V_q>$
- 長度(length)：一條路徑上的長度是指該路徑上所有邊的數目。

如: G1 中路徑 (1,2),(2,4),(4,1) 和(1,3),(3,4), (4,1)的長度都為3。G3 中路徑 <1,2>,<2,3>,<3,2>的長度為3。



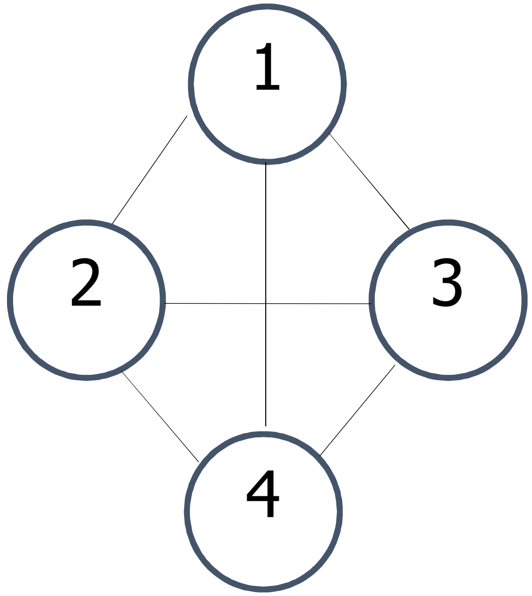
G1



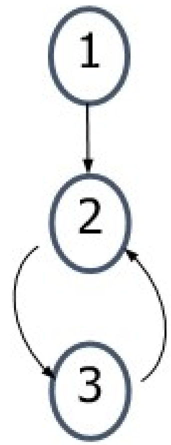
G3

圖形的一些專有名詞(8)

- 簡單路徑(simple path)：在一路徑上除了頭尾頂點之外，其餘的頂點皆是不相同的。 如: G1的路徑，(1,2),(2,4),(4,3)是簡單路徑而(1,2), (2,4), (4,2)則不是。G3的路徑，<1,2>,<2,3>是簡單路徑
- 循環(cycle)：是指一條頭尾頂點皆相同的簡單路徑。 G1的路徑 (1,2), (2,4), (4,1)是一個循環。
  - 有向循環：G3的路徑<2,3>,<3,2>是一個有向循環



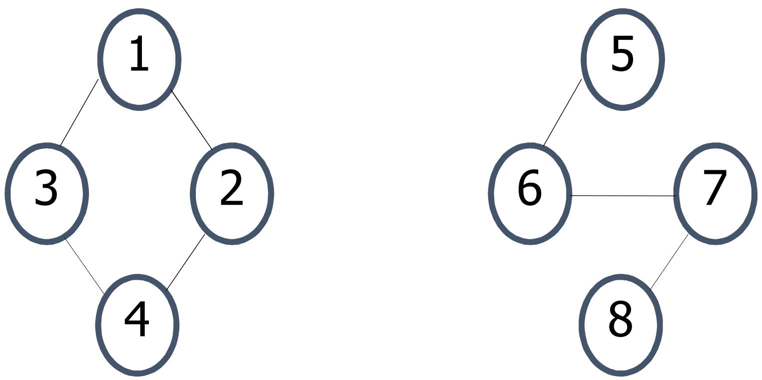
G1



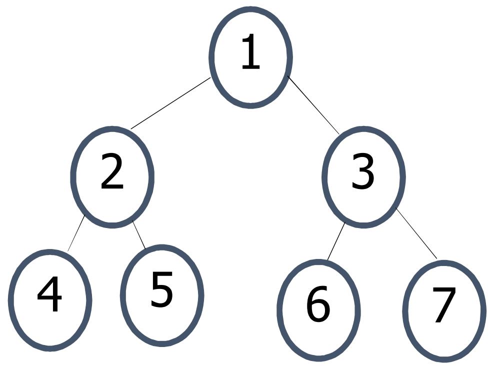
G3

圖形的一些專有名詞(9)

- 連通(connected)：在一個圖形G中，如果有一條路徑從V_1到V_2，那麼我們說V_1與V_2是連通的。若圖形中每一對的頂點均可找到一條路徑相連那該圖形為連通的。
  - 圖G5不是連通的。G2是連通的。



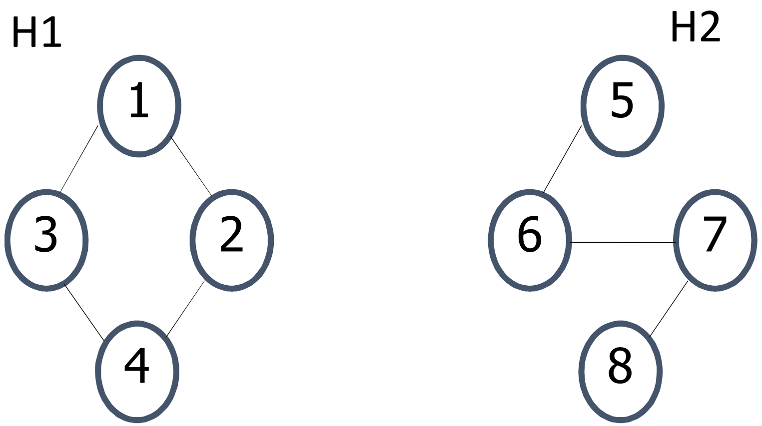
G5



G2

圖形的一些專有名詞(10)

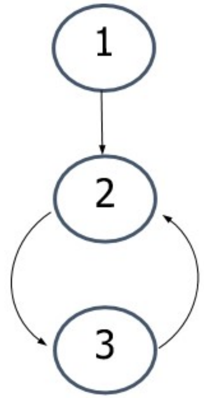
- 連通單元(connected component)：或稱單元(component)H是指該圖形中最大的連通子圖(maximum connected subgraph)，其中最大的意義是指在圖G中沒有任何一個子圖是連通且包含H。如圖G5有兩個單元H1和H2。



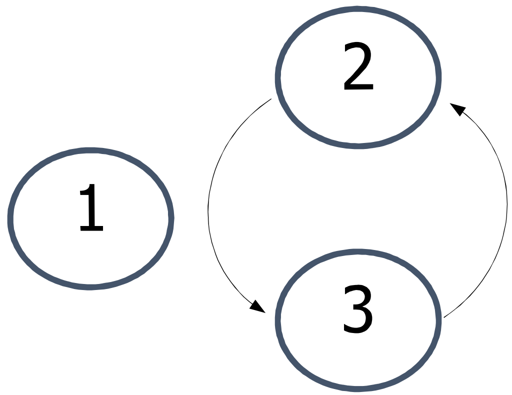
G5

圖形的一些專有名詞(11)

- 緊密連通(strongly connected)：在一有方向圖形中如果V(G)中的每一對不同頂點V_i, V_j 各有一條從V_i到V_j及從V_j到V_i的有方向路徑者稱之。
  - 右圖G3不是緊密連通，因為G3沒有V_2到V_1的路徑。
- 緊密連通單元(strongly connected component）：是指一個緊密連通最大子圖。如G3有兩個緊密連通單元。



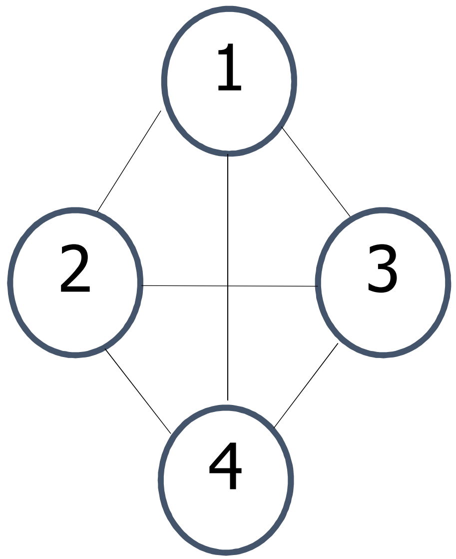
G3



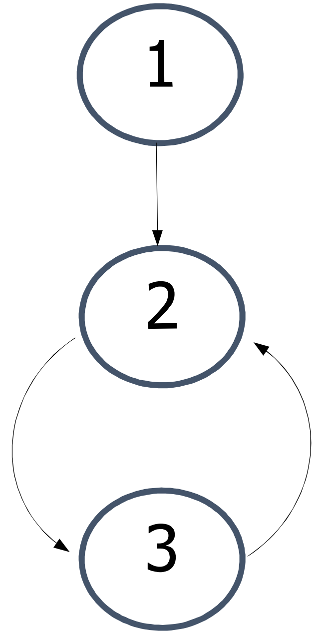
圖形的一些專有名詞(12)

- 分支度(degree)：附著在頂點的邊數。如V1的分支度為3
- 若G為有向圖
  - 入分支度(in-degree)：頂點V的入分支度是指以V為終點(即箭頭指向V)的邊數。
  - 出分支度(out-degree)：頂點V的出分支度是以V為起點的邊數。
  - 如: G3中頂點2的入分支度為2 而出分支度為1，分支度為3。
- 一個有n個頂點、e個邊的圖形中，d_i為頂點i的分支度，則圖形中的邊數：

$$e = (\sum_{i=1}^{n} d_i)/2$$



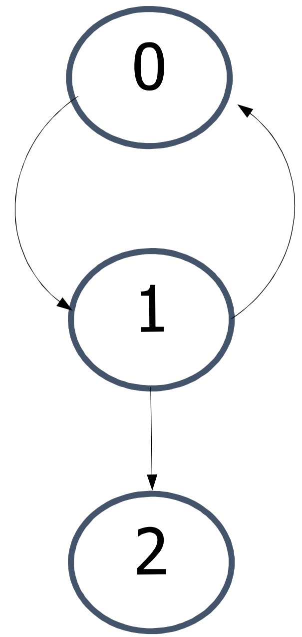
G1



G3

作業06-01

- V(G) = ?
- E(G) = ?
- G為complete graph?
- 頂點0相鄰到(adjacent to)那些點?
- 頂點0從那些點相鄰(adjacent from)
- 請畫出兩個G的子圖(subgraph)?
- 請寫出G中長度(length)為2的路徑(path)?
- 請寫出G中一條簡單路徑(simple path)?
- 請寫出G中一條循環(cycle)路徑?
- G是否連通?
- 頂點1的degree?
- 頂點1的in-degree?
- 頂點1的out-degree?



G

Graph ADT

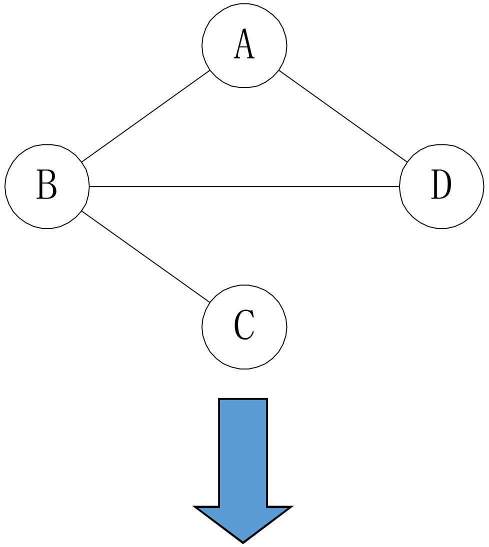
```
public abstract class Graph {
    int n; //number of vertices
    int e; //number of edges
    boolean isEmpty(){return n==0;};
    int numberOfVertices(){return n;}
    int numberOfEdges(){return e;}
    abstract int degree(int u);
    //return number of edges incident to vertex u
    abstract boolean existEdge(int u, int v);
    //return true if graph has the edge (u,v)
    abstract void insertVertex(int v);
    //insert vertex v into graph; v has no incident edges
    abstract void insertEdge(int u, int v);
    //insert edge (u,v) into graph
    abstract void deleteVertex(int v);
    //delete vertex v and all edges incident to it
    abstract void deleteEdge(int u,int v);
    //delete edge (u,v) from the graph
}
```

圖形的表示法

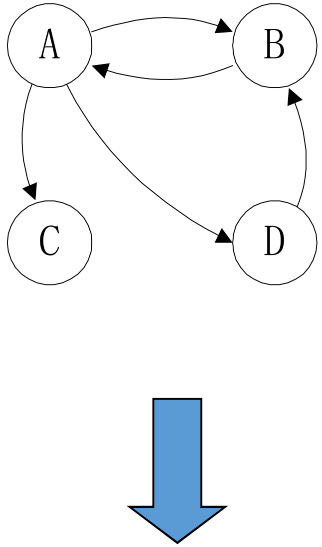
- 圖形的資料結構表示法常用的有下列二種：
  - 相鄰矩陣(adjacent matrix)；
  - 相鄰串列(adjacent list)。

圖形表示法 -相鄰矩陣

- 將圖形中的 n 個頂點，以一個 n × n 的二維矩陣來表示，其中每一元素V_{ij}，若$V_{ij} = 1$，表示圖形中 V_i 與 V_j 有一條邊為(V_i, V_j)，假若是有方向圖形的話，表示有一條邊為$<V_i, V_j>$。$V_{ij} = 0$表示頂點i與頂點j沒有邊存在 。



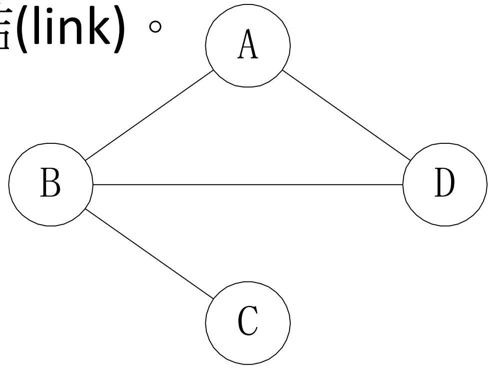
$$
\begin{array}{c}
 \\ A \\ B \\ C \\ D
\end{array}
\begin{array}{c}
\begin{array}{cccc} A & B & C & D \end{array} \\
\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}
\end{array}
$$

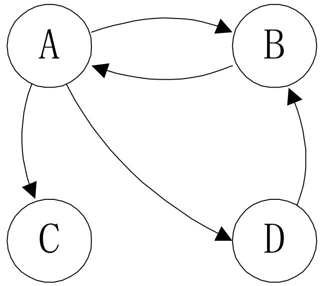


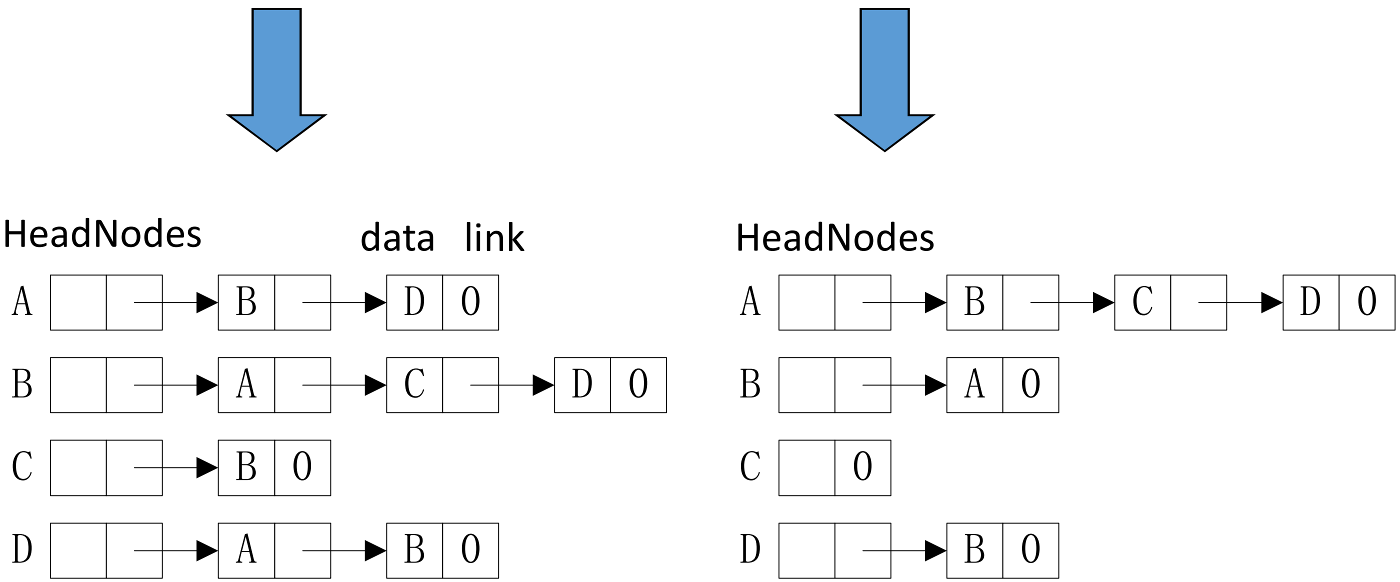
$$
\begin{array}{c}
 \\ A \\ B \\ C \\ D
\end{array}
\begin{array}{c}
\begin{array}{cccc} A & B & C & D \end{array} \\
\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}
\end{array}
$$

圖形表示法 - 相鄰串列

- 相鄰串列乃是將圖形中的每個頂點皆形成串列首，而在每個串列的節點，表示它們之間有邊存在。
- 圖形中每個頂點均對應一個串列，串列的節點包含資料(data)和鏈結(link)。

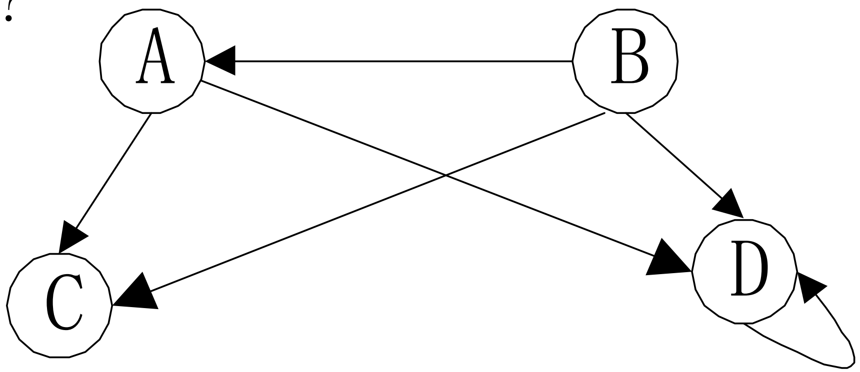






作業06-02

- 試用鄰接矩陣(Adjacency Matrix)及鄰接串列(Adjacency List)表示下列圖形？



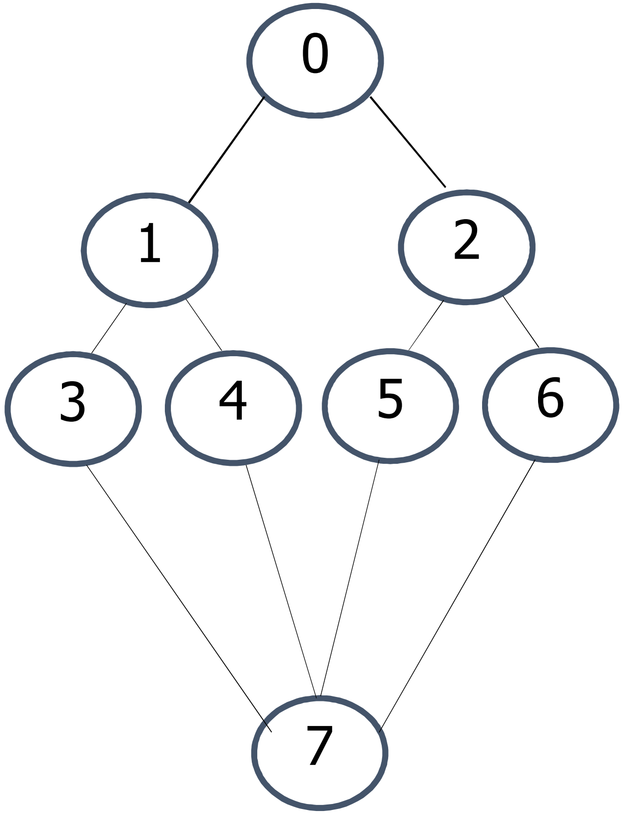
圖形追蹤

- 圖形的追蹤是從無方向圖形的某一頂點開始，去拜訪圖形中連通(connected)到該頂點的所有的頂點。
- 圖形的追蹤有兩種方法：
  - 深先搜尋(depth first search)；
  - 寬先搜尋(breadth first search)。

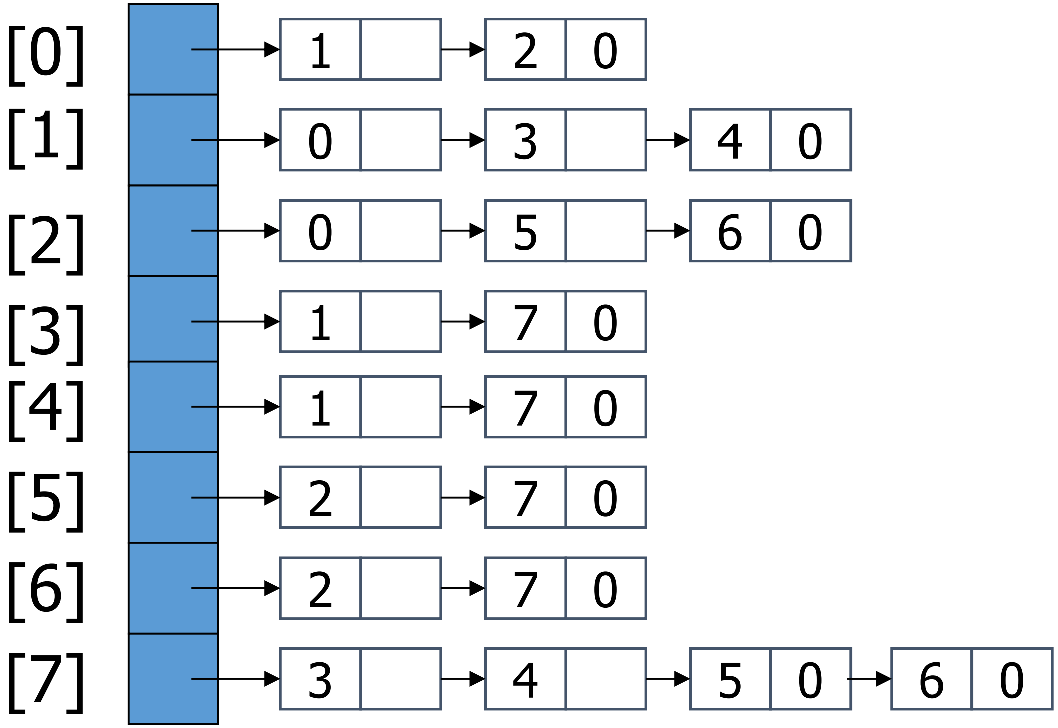
圖形追蹤 -深先搜尋(1)

- 先拜訪起始點V；
- 然後選擇與V相鄰而未被拜訪的頂點W，以W為起始點做縱向優先搜尋；
- 假使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，其尚有未被拜訪過的相鄰頂點，繼續做縱向優先搜尋；
- 假若從任何已走過的頂點，都無法再找到未被走過的相鄰頂點時，此時搜尋就結束了。

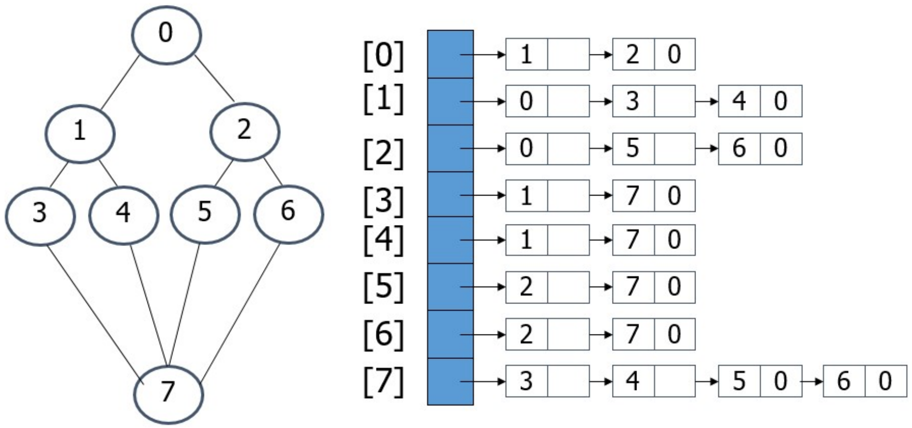
圖形追蹤 -深先搜尋(2)



0,1,3,7,4,5,2,6

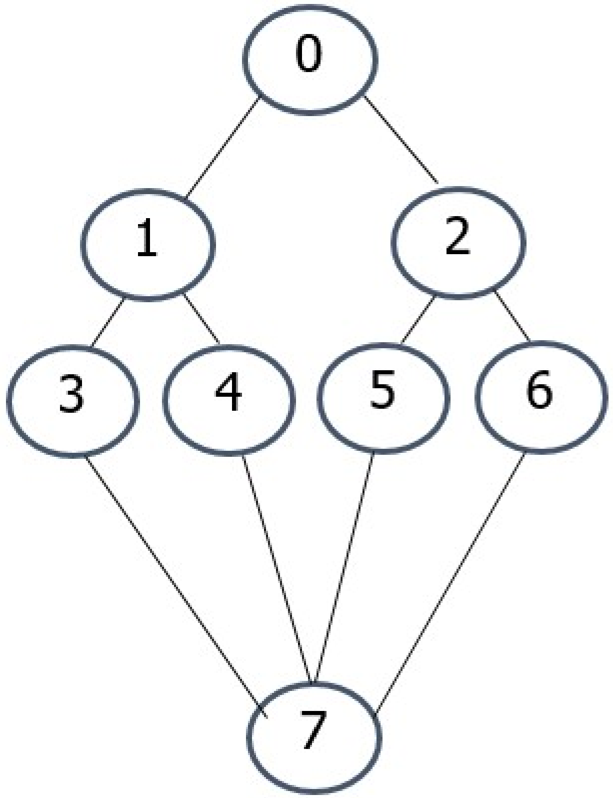


```java
public class Graph{
    int n; //number of vertices
    boolean[] visited;
    public HeadNode[] headNodes;
    Graph(int nv){
        this.n=nv;
        this.headNodes = new HeadNode[nv];
        visited = new boolean[nv];
    }
    void DFS(){
        DFS(0);
        System.out.println("");
    }
    void DFS(int v){
        visited[v] = true;
        System.out.print(v+",");
        ChainNode curNode = headNodes[v].head.next;
        while(curNode!=null){
        if(!visited[curNode.data])
            DFS(curNode.data);
        curNode = curNode.next;
    }
}
```



圖形追蹤 -寬先搜尋(1)

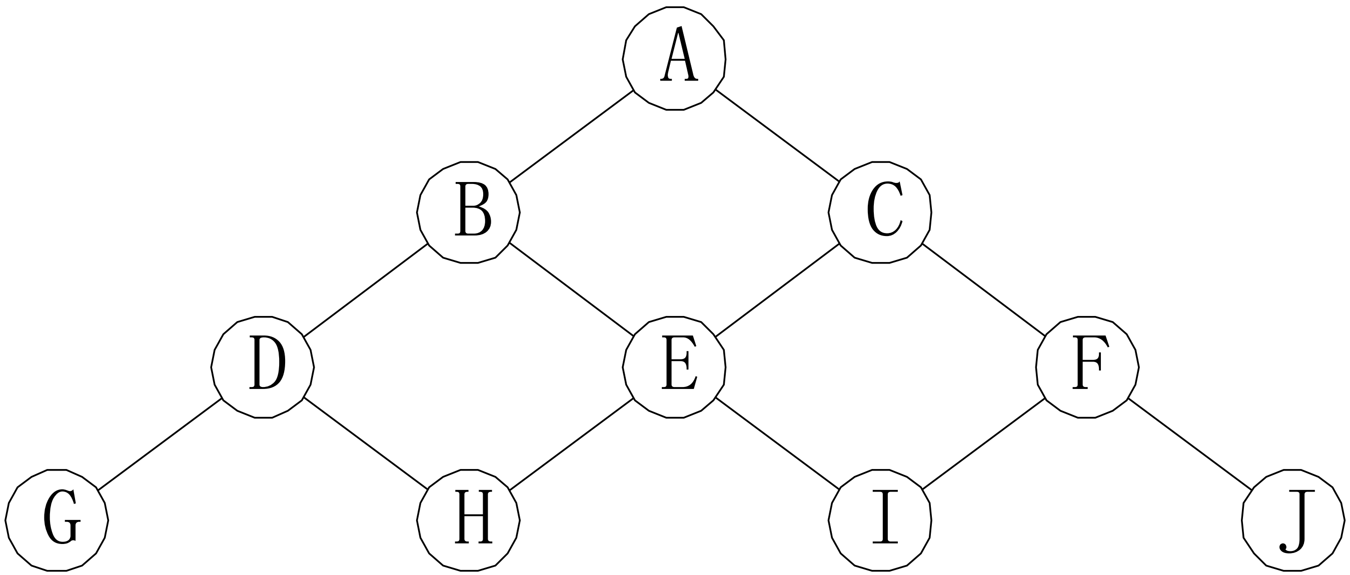
- 寬先搜尋先拜訪完所有的相鄰頂點，再去找尋下一層的其他頂點。
- 如右圖以寬先搜尋，其拜訪頂點的順序是0,1,2,3,4,5,6,7。
- 寬先搜尋以佇列來運作。



```java
public class Graph{
    int n; //number of vertices
    boolean[] visited;
    public HeadNode[] headNodes;
    Graph(int nv){
        this.n=nv;
        this.headNodes = new HeadNode[nv];
        visited = new boolean[nv];
    }
    void BFS(int v){
        visited[v] = true;
        System.out.print(v+",");
        Ch04MyIntQueue queue = new Ch04MyIntQueue();
        queue.push(new ChainNode(v));
        while(!queue.isEmpty()){
            v = queue.pop().data;
            ChainNode curNode = headNodes[v].head.next;
            while(curNode!=null){
                if(!visited[curNode.data]){
                    queue.push(new ChainNode(curNode.data));
                    visited[curNode.data] = true;
                    System.out.print(curNode.data+",");
                }
            curNode = curNode.next;
            }
        }
        System.out.println("");
    }
}
```

作業06-03

- 試以深先搜尋(Depth First Search)及寬先搜尋(Breadth First Search)拜訪下圖之每一個節點？



深先搜尋：A、B、D、G、H、E、C、F、I、J。
寬先搜尋：A、B、C、D、E、F、G、H、I、J。

作業06-04

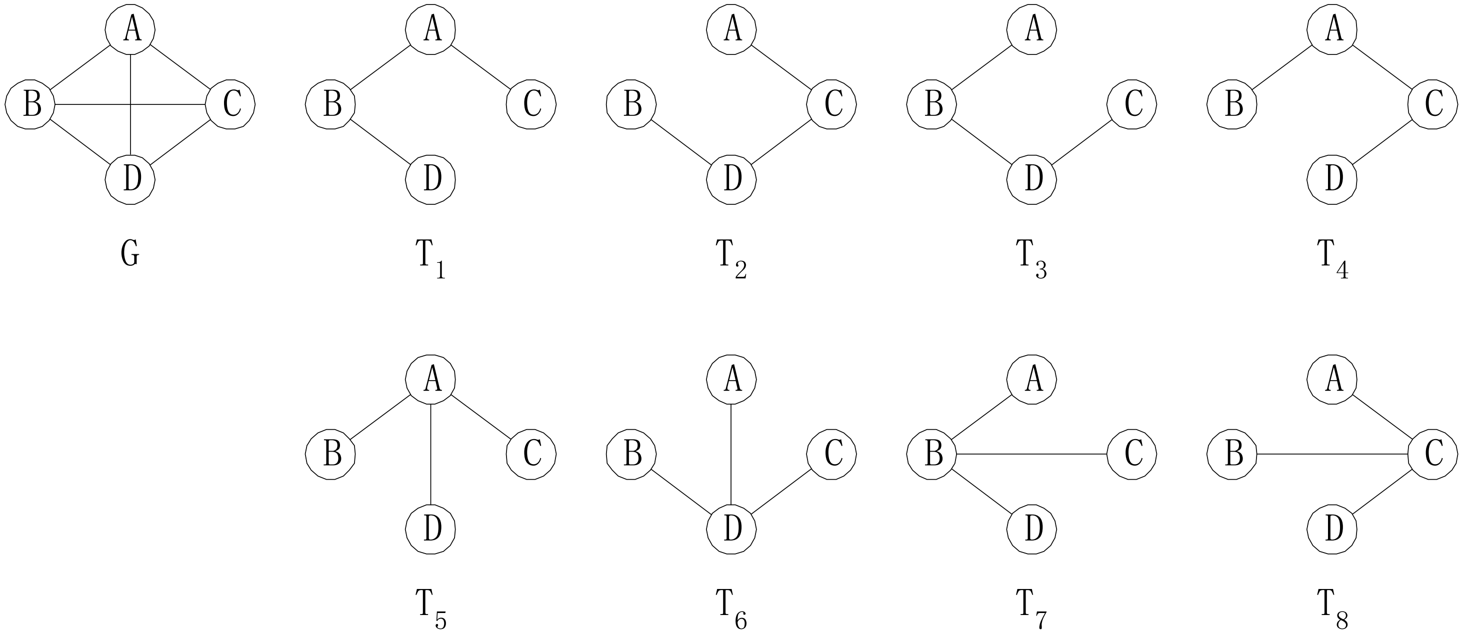
- 請實作深先搜尋演算法

作業06-05

- 請實作寬先搜尋演算法

擴張樹(Spanning Tree)

- 擴張樹
  - 一個包含 N 個頂點的無向相連圖，我們可以找出用圖中的 N-1 個邊來連接所有頂點的樹
  - 若再加入圖形中其餘的邊到擴張樹中必會形成迴路
  - 擴張樹中的任兩個頂點間都是相連的，也就是存在一條路徑可通，但此一路徑不一定是原圖形中該兩頂點之最短路徑。

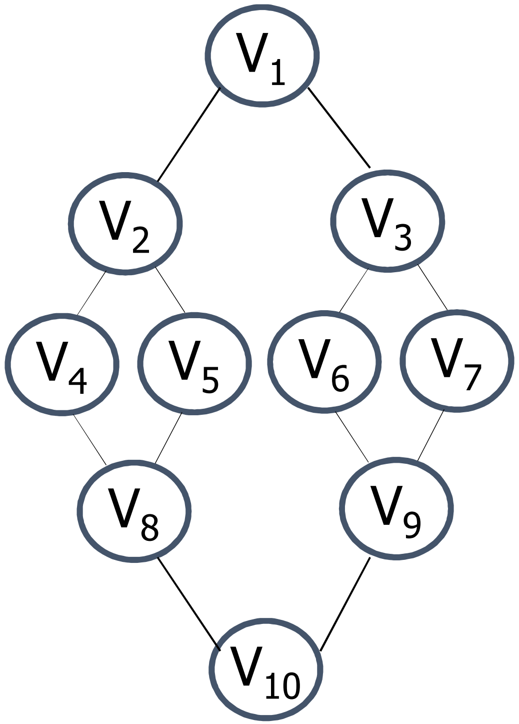


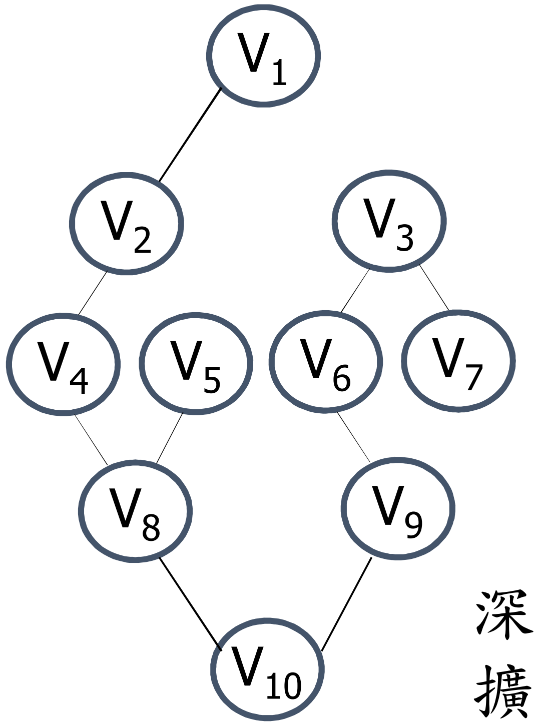
擴張樹

- 只要使用相鄰串列(adjacent list)，不論用何種追蹤方式，我們一定：
  - 可以拜訪所有的頂點。
  - 每個頂點只檢查一次
- 所以圖形的邊可以分成兩部份:
  - T：有拜訪過的邊；
  - N：沒拜訪過的邊。
- 上述的T和所有的頂點 (G) 即可形成一棵擴張樹。

深先搜尋擴張樹

- 假若使用深先搜尋的追蹤方式，則稱為深先搜尋擴張樹。

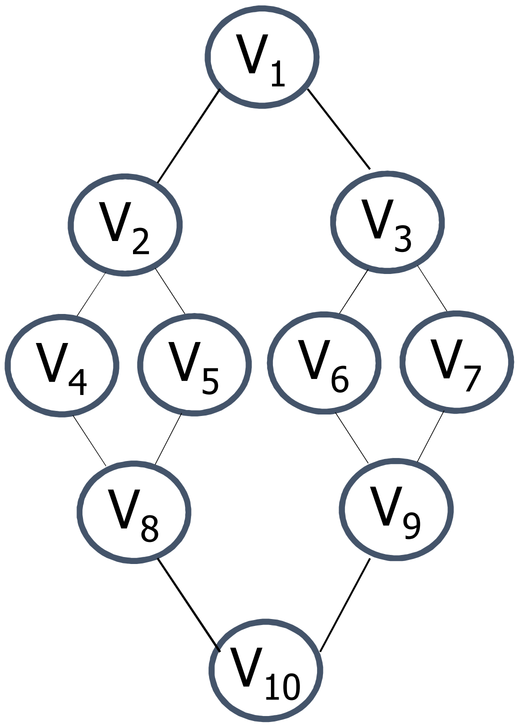


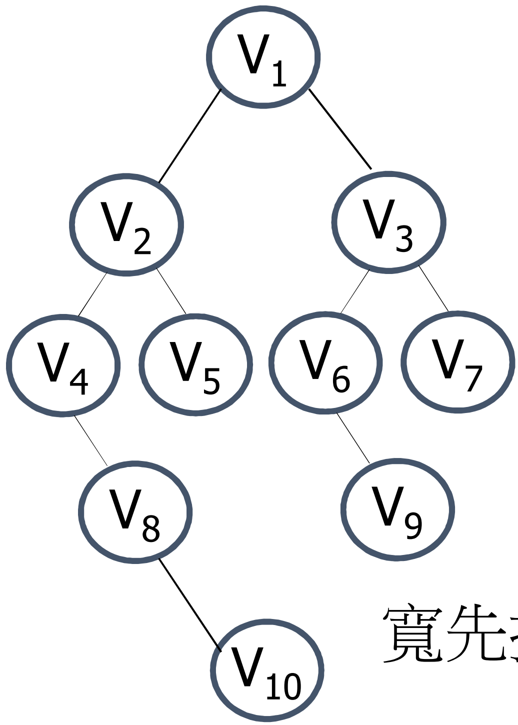


深先搜尋
擴張樹

寬先搜尋擴張樹

- 若使用寬先搜尋的追蹤方式，則稱為寬先搜尋擴張樹。





寬先搜尋擴張樹

擴張樹

- 假設圖形中的邊都有一個比重(weight)則該圖形稱為比重圖形(weighted graph)
- 比重可能為成本或距離則該圖形稱為網路(Network)
- 一棵擴張樹的成本該樹所有邊的比重的總和
- 而一個無方向的比重圖形可有很多擴張樹
- 找出最小成本擴張樹(Minimum-Cost Spanning Tree)是一個重要的問題

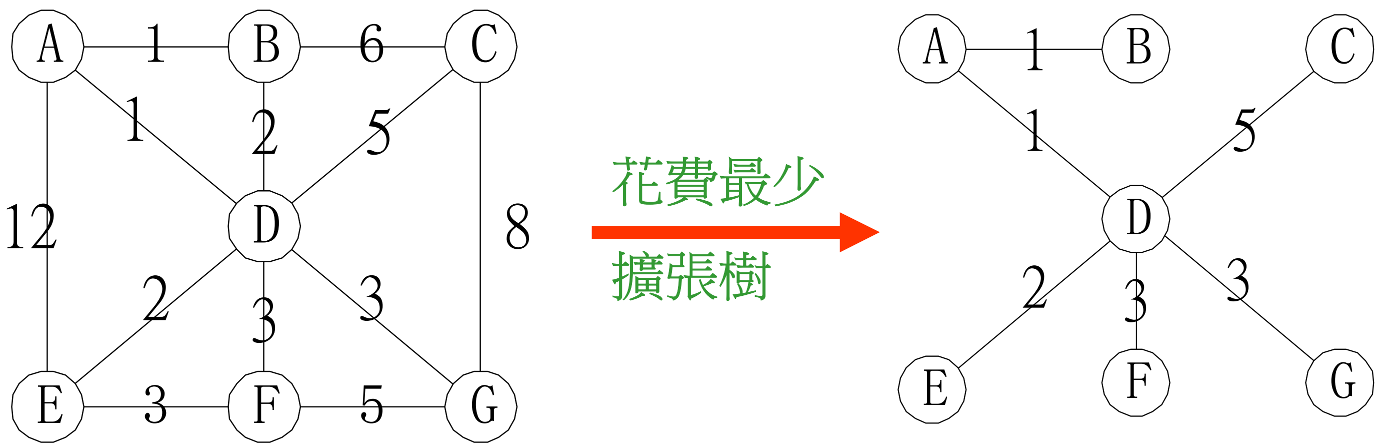
最小成本擴張樹

求最小成本擴張樹有兩種方法：

- **Kruskal’s algorithm**
- **Prim‘s algorithm** 。

Kruskal’s algorithm (1)

1. 令花費最少擴張樹 T= ψ 。
2. 從E中選取花費最少的邊(v , w) 。
3. 如果(v, w)不會使T產生迴路則將之加到T中；
   否則，自E中刪除之。
4. 重複步驟2、3，直到T的邊數等於 N-1 為止。

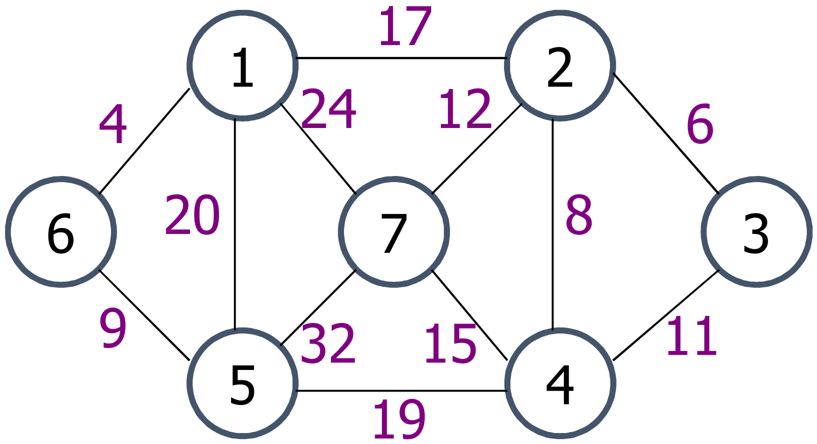


Kruskal's algorithm (2)

```
T= ∅ ;
while ((T contains less than n - 1 edges) && (E not empty)) {
        choose an edge (v, w) from E of lowest cost;
        delete (v, w) from E;
        if ((v, w) does not create a cycle in T) add (v, w) to T;
        else discard (v, w);
}
if (T contains fewer than n - 1 edges) cout « "no spanning tree" « endl;
```

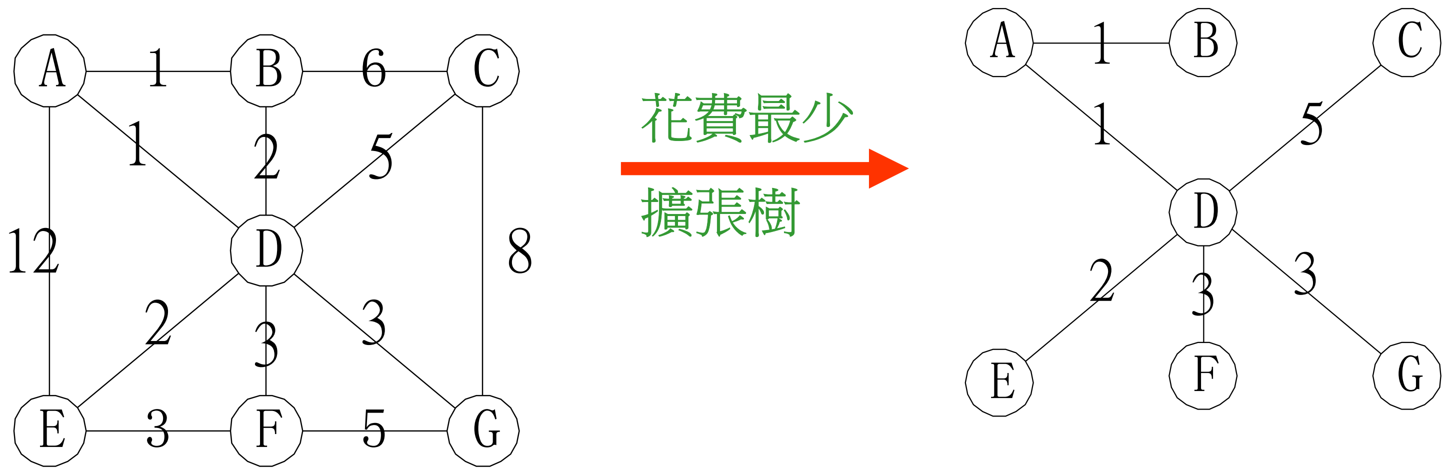
作業06-06

請以Kruskal's algorithm
來找出最小成本擴張樹



Prim's algorithm(1)

1. 令 TV_0=V，TV= ψ ，T= ψ 。
2. 從TV_0中任選一個頂點，將之從TV_0搬移到TV。
3. 找出一條連接TV_0和TV的最少花費邊(u, v)，其中u∈TV, v∈TV_0，且邊(u, v)加到T不會造成迴路。
4. 將頂點v自TV_0搬移到TV，並將邊(u, v)加入T。
5. 重複步驟3、4直到 TV_0= ψ 。

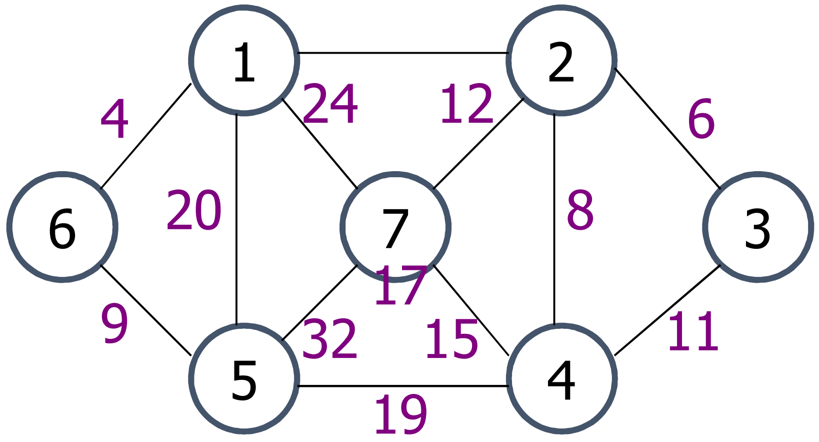


Prim's algorithm(2)

```
// Assume that G has at least one vertex.
TV = {A}; // start with vertex A and no edges
TV0 ={all vertices except A}
for (T = Ø; T contains fewer than n -1 edges; add (u, v) to T)
{
      Let (u, v) be a least-cost edge such that u ∈ TV and v ∈ TV0;
      if (there is no such edge) break;
      add v to TV;
}
if (T contains fewer than n -1 edges) cout « "no spanning tree" « endl;
```

作業06-07

- 請以Prim's algorithm來找最小成本擴張樹



進階作業06-01

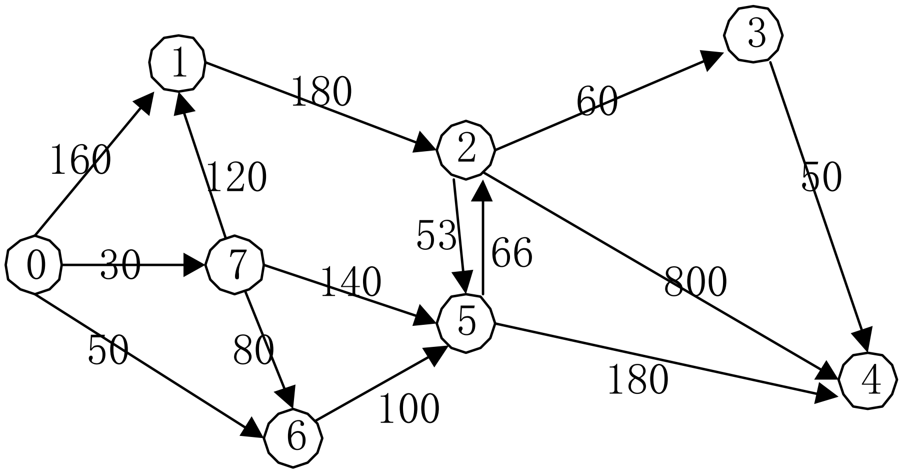
請實作Kruskal's algorithm and Prim's algorithm

最短路徑(shortest path)

- 單點到所有其他所有點
  - 找出網路中某一頂點到其他頂點的最短路徑。
  - Dijkstra ‘s algorithm
- 單點到單點(All Pairs Shortest Paths)
  - 地圖中找出A到B最短距離
  - Floyd algorithm

Dijkstra 's algorithm (1)

- 從頂點0到其餘每一個頂點之最短徑和花費



$$
\mathrm{COST} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{bmatrix}
\infty & 160 & \infty & \infty & \infty & \infty & 50 & 30 \\
\infty & \infty & 180 & \infty & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & 60 & 800 & 53 & \infty & \infty \\
\infty & \infty & \infty & \infty & 50 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\
\infty & \infty & 66 & \infty & 180 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & 100 & \infty & \infty \\
\infty & 120 & \infty & \infty & \infty & 140 & 80 & \infty
\end{bmatrix}
\end{array}
$$

Dijkstra 's algorithm (2)

- 頂點0到其餘頂點之最短距離選取過程

步驟	被選上之頂點 U	距離矩陣 DIST								未被找過之頂點 (MATRK(i)=0)
		0	1	2	3	4	5	6	7	
0		0	160	∞	∞	∞	∞	50	30	1, 2, 3, 4, 5, 6, 7
1	7	0	150	∞	∞	∞	170	50	30	1, 2, 3, 4, 5, 6
2	6	0	150	∞	∞	∞	150	50	30	1, 2, 3, 4, 5
3	1	0	150	330	∞	∞	150	50	30	2, 3, 4, 5
4	5	0	150	216	∞	330	150	50	30	2, 3, 4
5	2	0	150	216	276	330	150	50	30	3, 4
6	3	0	150	216	276	326	150	50	30	4
7	4	0	150	216	276	326	150	50	30	

- 頂點0到其餘頂點之最短路徑和距離

起點	終點	最 短 路 徑	最短距離
0	1	0 → 7 → 1	150
0	2	0 → 6 → 5 → 2	216
0	3	0 → 6 → 5 → 2 → 3	276
0	4	0 → 6 → 5 → 2 → 3 → 4	326
0	5	0 → 6 → 5	150
0	6	0 → 6	50
0	7	0 → 7	30

Dijkstra 's algorithm (3)

1. 設定花費矩陣COST之初值，即對於每一個邊，令

   COST[i][j]=邊<i,j>之距離，若<i,j> $\in$ E(G)。

   COST[i][j]=∞ ，若<i,j> $\notin$ E(G)。

2. 設定MARK和DIST兩矩陣之初值，即對每一頂點令

   MARK[i]=0;

   DIST[i]=COST[V][i];

3. 處理起始頂點 V，即令

   MARK[V]=1;

   DIST[V]= 0;

4. 當還有未被選取之頂點時重複步驟 5、6、7。

5. 選取一個頂點 U，使得 U 是所有未被選取之頂點中DIST[U]是最少 者，即

   DIST[U] = min {DIST[W]}, W為未被選取之頂點的編號

6. 將頂點 U 做上記號，即令

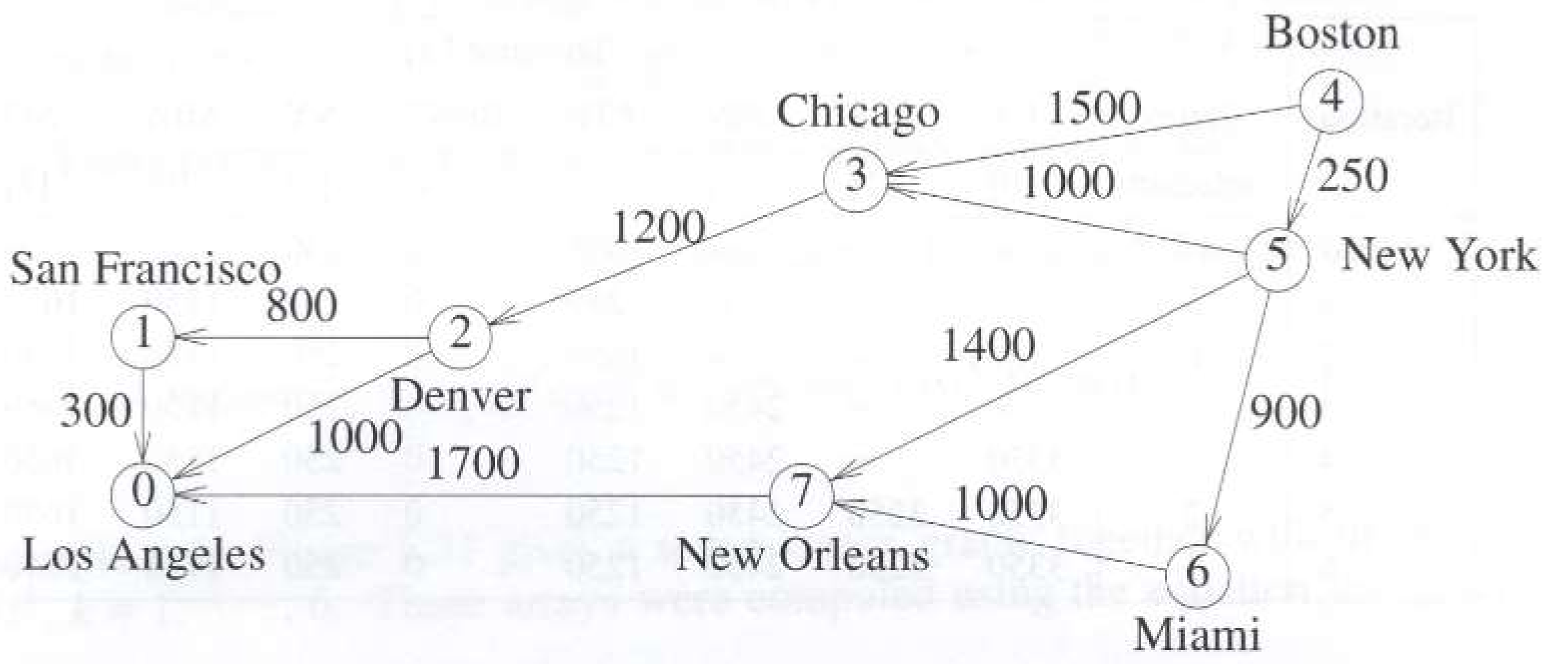
   MARK[U]=1。

7. 更新剩餘未被選取的頂點(MARK[W]=0)之距離矩陣值，即令

   DIST[W]=min｛DIST[W]，DIST[U]+COST[U][W]｝。

作業06-08

- 請使用Dijkstra ‘s algorithm，找出Boston(4)到所有點的最短路徑



進階作業06-02

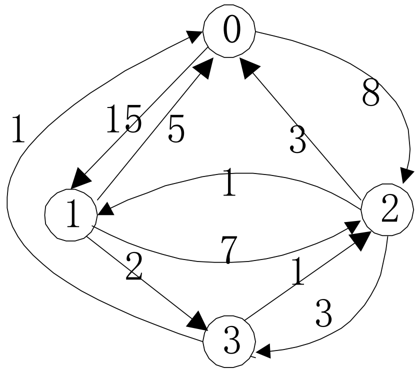
請實作Dijkstra ‘s algorithm

Floyd algorithm(1)

- 任兩頂點對之最短距離 (All Pairs Shortest Paths)
  - 設頂點編號為0，1，2，…，N-1
  - 令$A^{-1}[i][j]$=COST[i][j]，$A^{-1}[i][j]$是頂點 i 至頂點 j 之直通距離。
  - 求出$A^{K}[i][j]$
    - $A^{K}[i][j]= \min\{A^{K-1}[i][j], A^{K-1}[i][k] + A^{K-1}[k][j]\}$，$0 \leqq k \leqq N-1$。
    - k表示經過的頂點，$A^{k}[i][j]$為從頂點i到j的經由k頂點的最短路徑。
  - $A^{N-1}[i][j]$代表i到j的最短距離，即A^{N-1}便是我們所要求的最短路徑成本矩陣。

Floyd algorithm(2)

- 任兩頂點對之最短距離 (All Pairs Shortest Paths)
  - 由 A^3 得知頂點 0 到頂點 3 之最短距離為 11，
  - 頂點 3 到頂點 1 之最短距離為 2



(a)

$$\begin{array}{c|c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 15 & 8 & \infty \\ 5 & 0 & 7 & 2 \\ 3 & 1 & 0 & 3 \\ 1 & \infty & 1 & 0 \end{bmatrix} \end{array}$$

(b) A^{-1}

$$\begin{array}{c|c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 15 & 8 & \infty \\ 5 & 0 & 7 & 2 \\ 3 & 1 & 0 & 3 \\ 1 & 16 & 1 & 0 \end{bmatrix} \end{array}$$

(c) A^0

$$\begin{array}{c|c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 15 & 8 & 17 \\ 5 & 0 & 7 & 2 \\ 3 & 1 & 0 & 3 \\ 1 & 16 & 1 & 0 \end{bmatrix} \end{array}$$

(d) A^1

$$\begin{array}{c|c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 9 & 8 & 11 \\ 5 & 0 & 7 & 2 \\ 3 & 1 & 0 & 3 \\ 1 & 2 & 1 & 0 \end{bmatrix} \end{array}$$

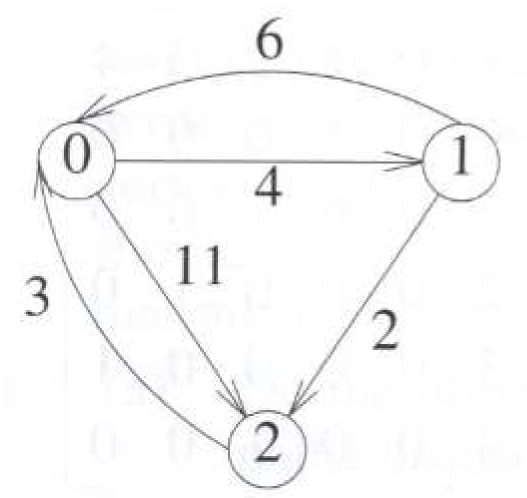
(e) A^2

$$\begin{array}{c|c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 9 & 8 & 11 \\ 3 & 0 & 3 & 2 \\ 3 & 1 & 0 & 3 \\ 1 & 2 & 1 & 0 \end{bmatrix} \end{array}$$

(f) A^3

作業06-09

- 請使用Floyd algorithm，找出所有點對點的最短路徑



進階作業06-03

- 請實作Floyd algorithm