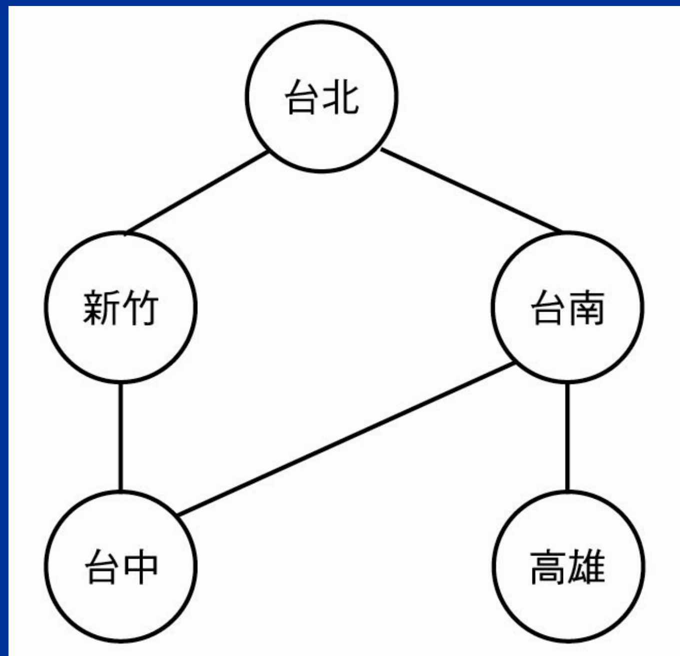
演算法 基本圖論

- 圖形專有名詞
- 圖形資料結構
- 圖形搜尋演算法

圖形 (Graph)

- 將複雜問題使用圖形來表達，以了解問題的本質



- 樹：節點間以邊線連結，節點間不循環的圖形
- 圖形：由節點和邊線所組成的集合

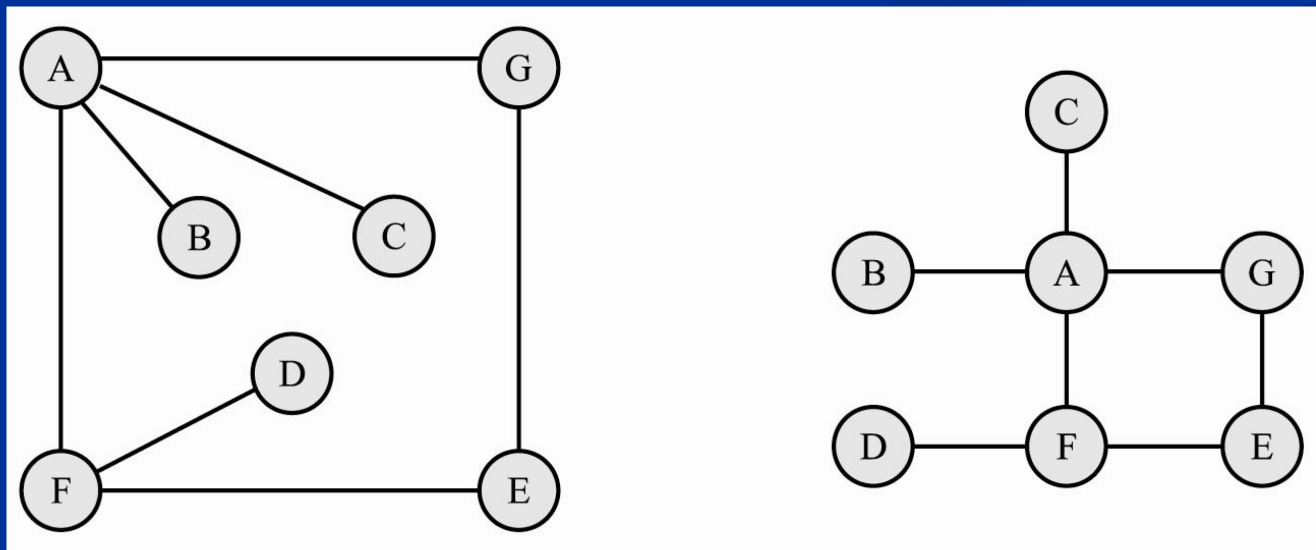
  (node, vertex)　(edge)

  以 G = (V, E) 來表示

  V 是所有節點的集合　V={ A, B, C, D, E, F, G }

  E 是所有邊線的集合

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



有向圖與無向圖

- 邊線 (x, y) 存在, x 能到 y, 則 y 與 x 相鄰 (adjacent)

  $x \rightarrow y$

- 圖形依其邊線是否具方向性區分爲有向圖和無向圖

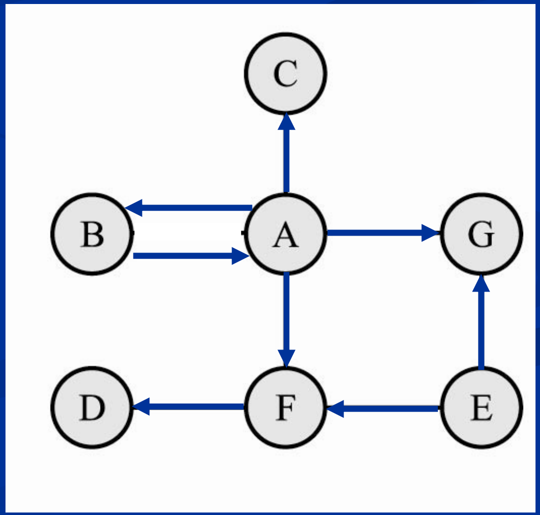
  無向(undirected): 邊線 (x, y) 等同於 (y, x)

  有向(directed): 邊線 (x, y) 不等同於 (y, x)

V={ A, B, C, D, E, F, G}

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

digraph



圖的表示 -- 相鄰矩陣

- adjacency matrix 使用二維陣列, 空間複雜度 $\Theta(V^2)$
  當邊線(x, y)存在時，adj[x][y]=1，否則adj[x][y]=0

無向 adj[x][y]=adj[y][x]

y →

x ↓	A	B	C	D	E	F	G
A	0	1	1	0	0	1	1
B	1	0	0	0	0	0	0
C	1	0	0	0	0	0	0
D	0	0	0	0	0	1	0
E	0	0	0	0	0	1	1
F	1	0	0	1	1	0	0
G	1	0	0	0	1	0	0

有向

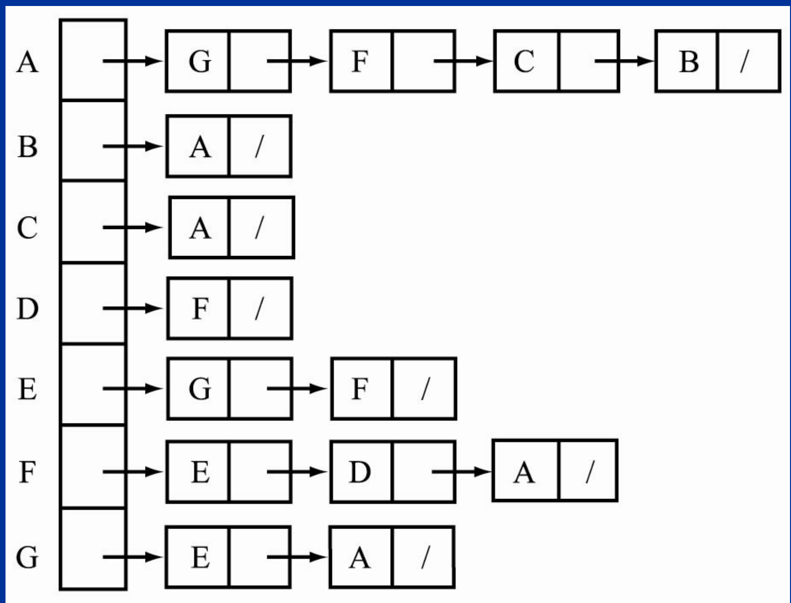
y →

x ↓	A	B	C	D	E	F	G
A	0	1	1	0	0	1	1
B	1	0	0	0	0	0	0
C	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0
E	0	0	0	0	0	1	1
F	0	0	0	1	0	0	0
G	0	0	0	0	0	0	0

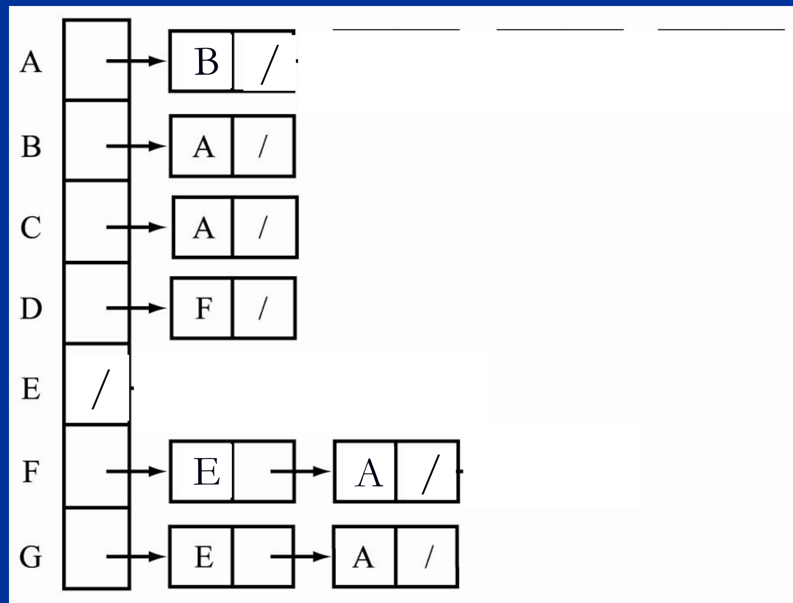
圖的表示 -- 相鄰串列

- adjacency list 一維陣列 adj[V]，空間複雜度 Θ(V+E)
  adj[x] 所指的節點串列, 即與 x 相鄰的節點(能進入 x)

無向



有向



struct node { int v; struct node* next; };

範例 10-1

```
#define index(v) (v-'A')
char *edges[]={"AB","AC","AF","AG","DF","EF","EG"};
#define  E_MAX  (sizeof(edges)/sizeof(char*))
#define  V_MAX  7

struct node {  int v;  struct node* next; };
struct node * adj[V_MAX];

void insert_edge(int x, int y){
      struct node *n;
      n=(struct node*)malloc(sizeof(struct node));
      n->v=x;
      n->next=adj[y];
      adj[y]=n;
}
```

邊線 (x, y) 存在

y 與 x 相鄰, x 能進入 y

範例 10-1

```
void adj_list(void){
        int i, x, y;
        for(i=0;i<V_MAX;i++) adj[i]=NULL;
        for(i=0;i<E_MAX;i++){
                x=index(edges[i][0]);
                y=index(edges[i][1]);
                insert_edge(x,y);
                insert_edge(y,x);
        }
}
```

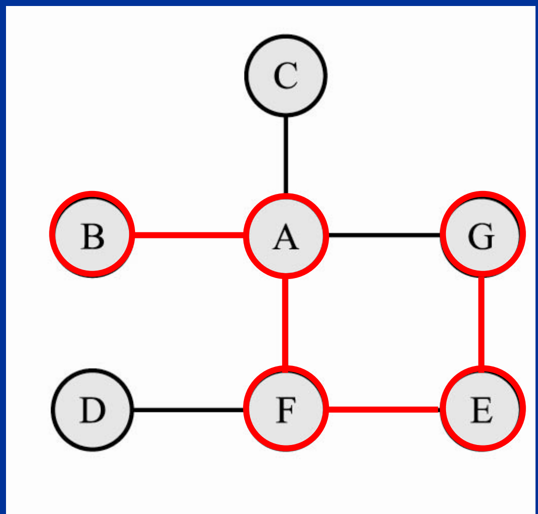
```
A -> G F C B
B -> A
C -> A
D -> F
E -> G F
F -> E D A
G -> E A
```

- **路徑(Path)** ：從節點 X 到 Y 所經過的節點串列
  路徑長度：路徑的邊線數量

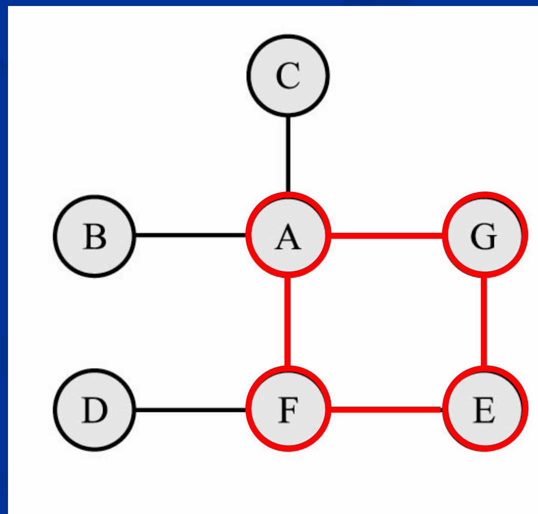
- **簡單路徑(Simple Path)**
  一條路徑沒有經過重複的節點

- **迴路(cycle)** 一條簡單路徑的起訖都是同一節點

{B, A, F, E, G}



{A, G, E, F, A}



- 連接圖(connected graph)

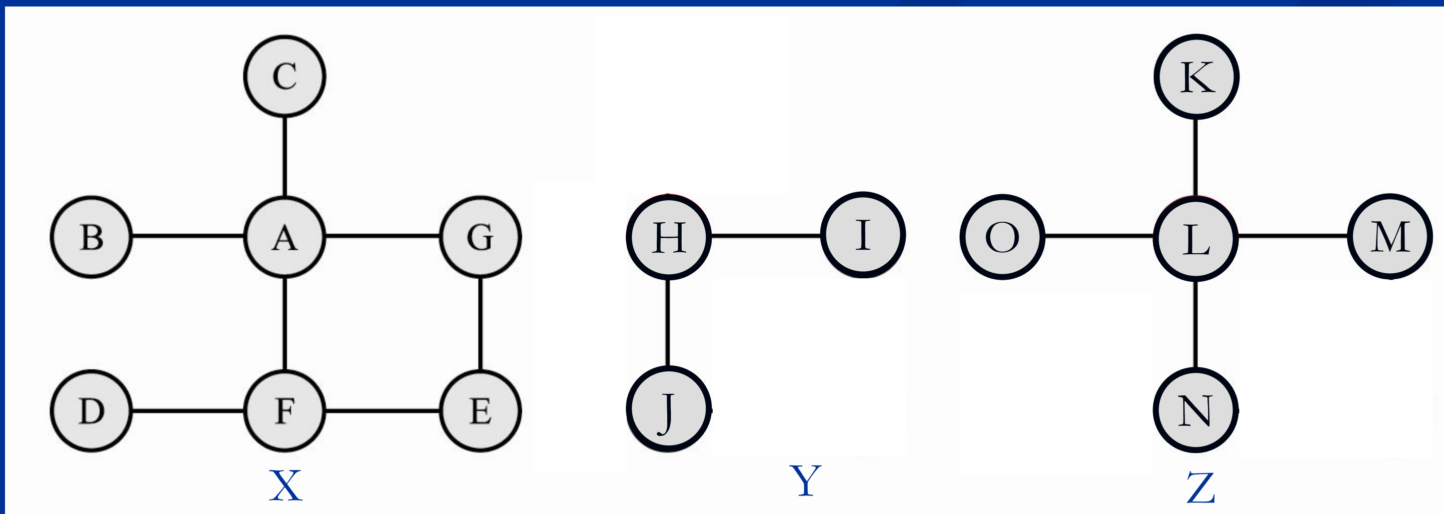
  每個節點到其他節點都存在一條路徑 {X} {Y} {Z}

  連接元件(connected component)

  一個圖的各個連接的子圖形,{X, Y, Z} 中的 X, Y, Z

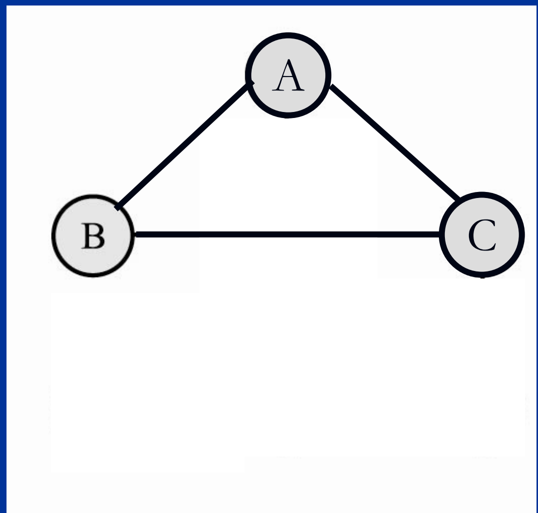
- 樹(tree) 一個沒有迴路(不循環 acyclic)的連接圖

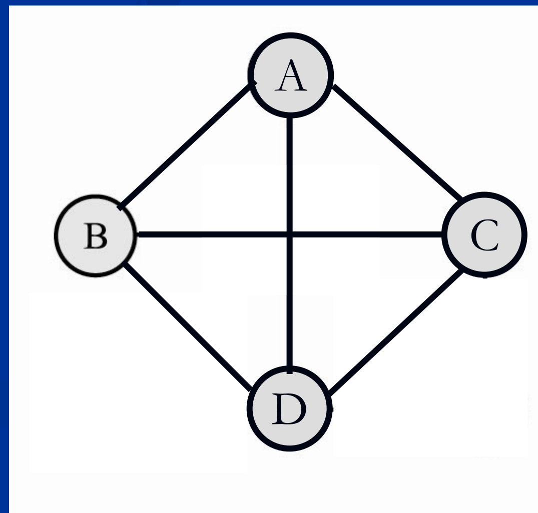
  森林(forest) 由一群不相連的樹所組成



完全圖 (Complete graph)

- 一個圖形包含 V 個節點，則
  1) 可能有 0 到 V(V-1)/2 條邊線
  2) 完全圖：圖形具有 V(V-1)/2 條邊線
  3) 邊線數較少，如少於 V (logV) ，稱爲稀疏圖 (sparse)，反之稱爲茂密圖(dense)



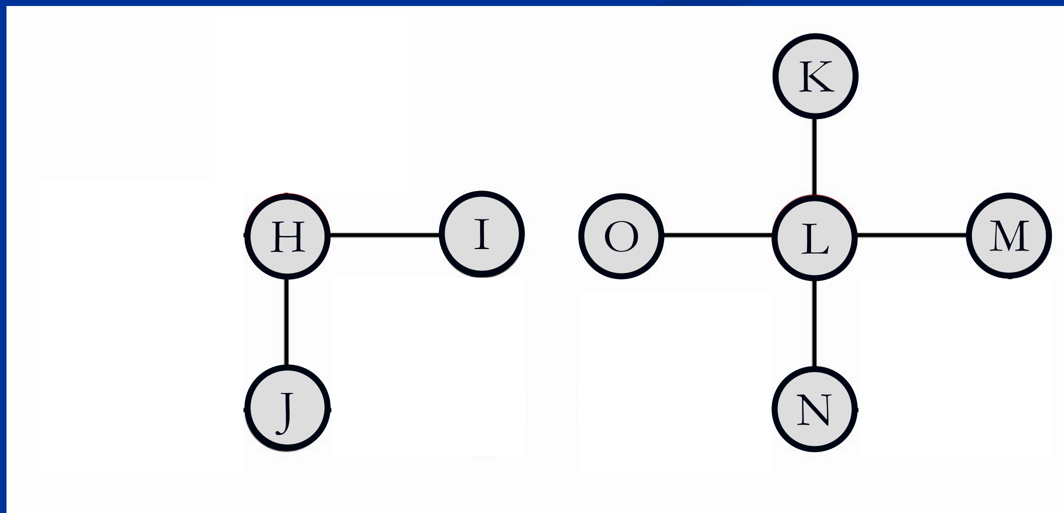


樹

- 包含 V 個節點的樹，剛好只會有 V-1 條邊線

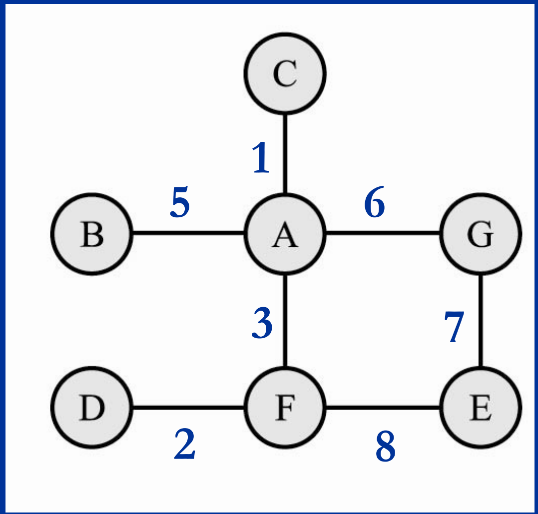
$$E = V - 1$$

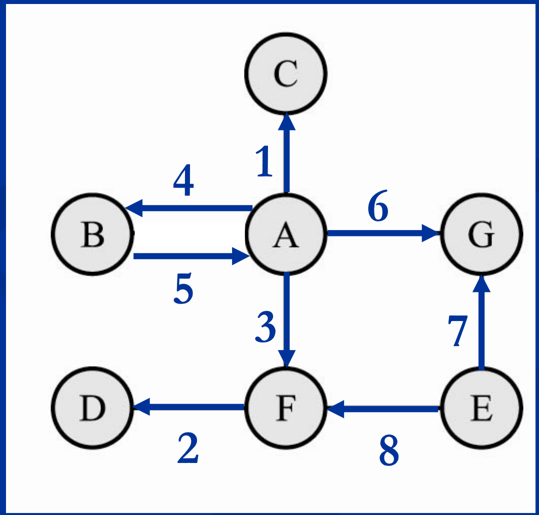
- 任意加上一條邊線，此圖形一定會形成迴路



加權圖(Weighted graph)

- 邊線除了代表兩個節點間的連接性外，另包含加權值(weight, cost)，稱爲加權圖

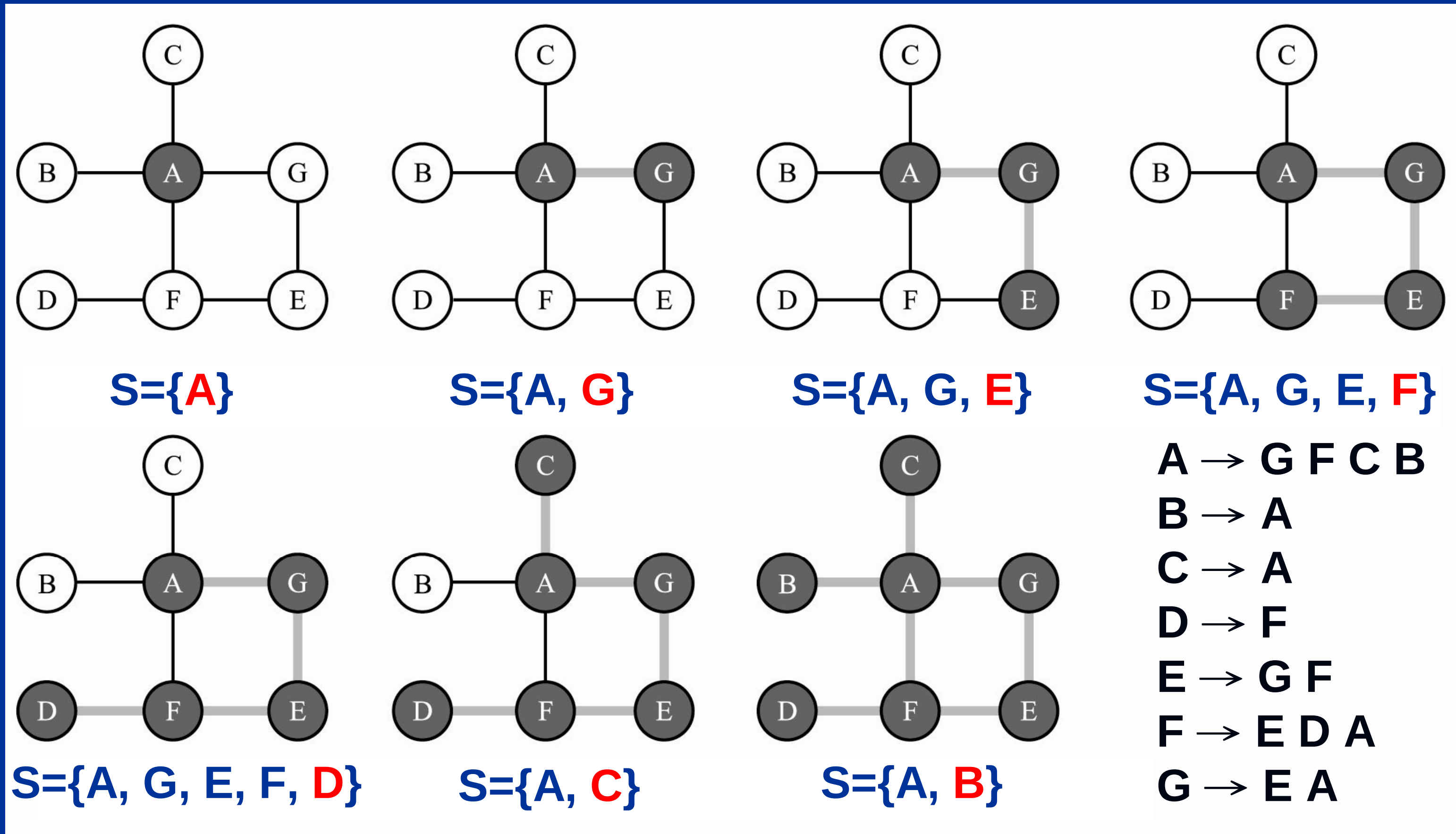




加權 + 有向圖 = 網路

圖形搜尋 -- 深度優先搜尋

- 圖形搜尋或圖形的訪問 (graph traversal)
- 拜訪的節點還有連結其他節點，就繼續訪問下一個節點，直到沒有連接節點或者遇到已訪問節點



深度優先搜尋

範例 10-2

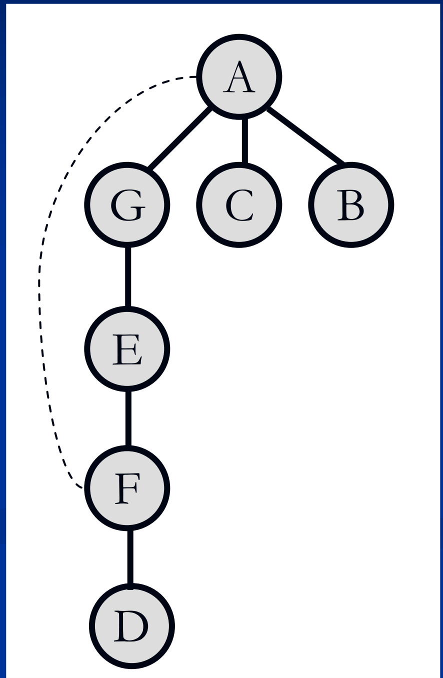
```
int visit[V_MAX];
static int seq=1;
void dfs(int v){
        struct node *t;
        printf("->%c", v+'A');       visit[v]=seq;   seq++;

        for( t=adj[v]; t != NULL; t=t->next )
                if( visit[t->v] == 0) dfs(t->v);
}
main(){
        int i;    adj_list();
        for( i=0; i<V_MAX; i++) visit[i]=0;
        for( i=0; i<V_MAX; i++) if( visit[i]==0 ) dfs(i);
}
```

深度優先搜尋分析

輸出結果

->A->G->E->F->D->C->B



深度優先搜尋樹
DFS tree

- 效能分析

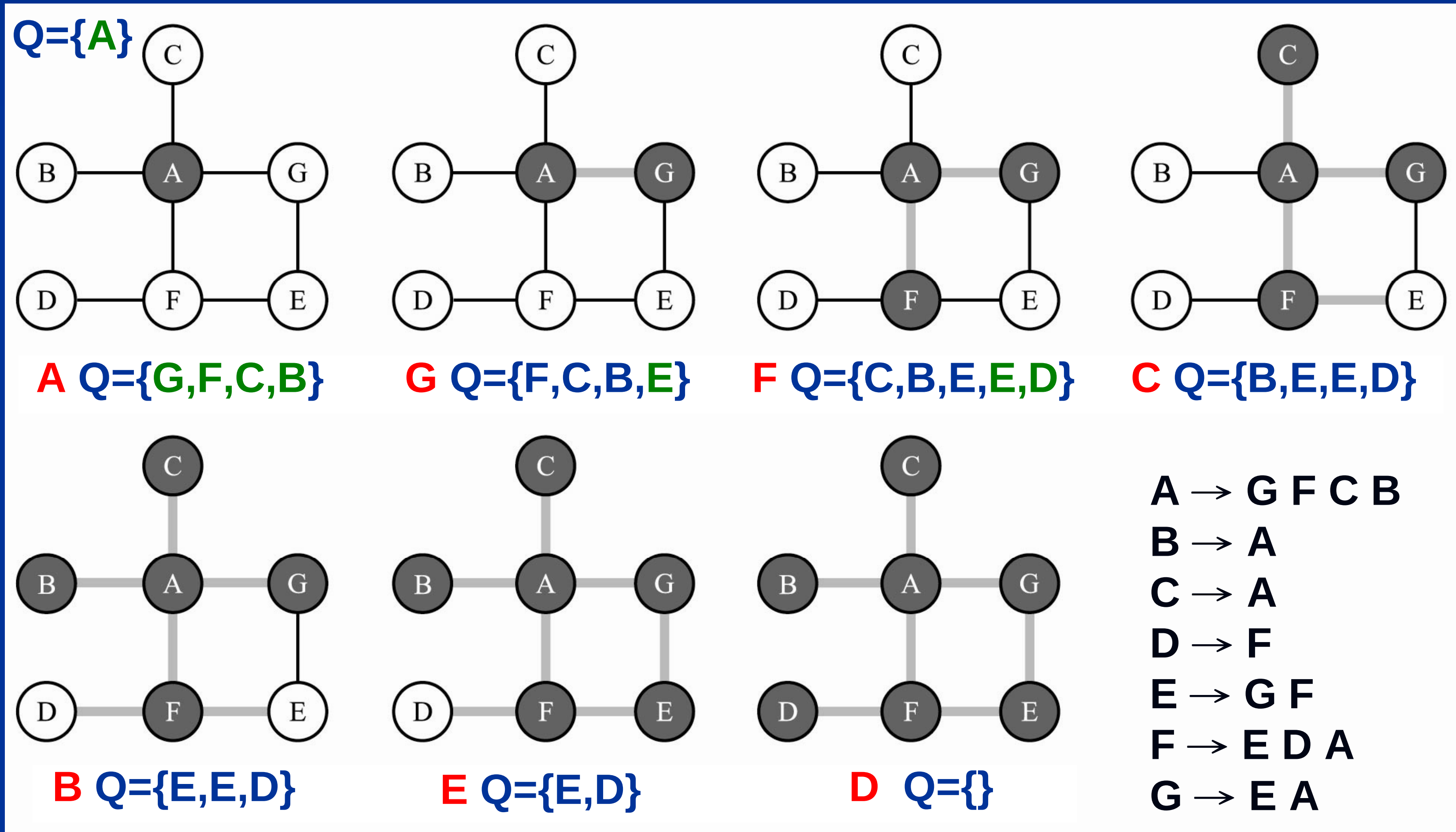
  時間複雜度: 與使用的資料結構空間複雜度相同

  1) 相鄰串列 O(V+E)　2)相鄰矩陣 $O(V^2)$

  空間複雜度: O(V)

圖形搜尋 -- 廣度優先搜尋

- 先拜訪完某節點其所有相鄰節點後，再深入下一層繼續探索



範例 10-3

廣度優先搜尋

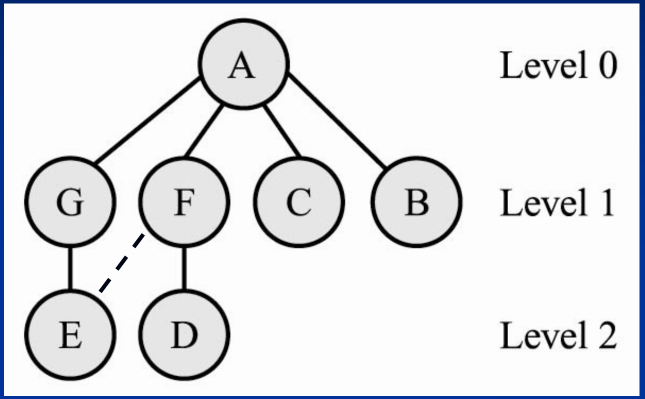
```
int visit[V_MAX];     static int seq=1;
void bfs(int v){         struct node *t;
        enqueue(v);
        while( ! is_queue_empty() ){
                v=dequeue();
                  if( visit[v] != 0) continue;
                  printf("->%c", v+'A');    visit[v]=seq; seq++;
                for( t=adj[v]; t!=NULL; t=t->next )
                  if( visit[t->v] == 0) enqueue(t->v);
        }
}
main(){
        int i;        adj_list();        queue_init(E_MAX);
        for( i=0; i<V_MAX; i++) visit[i]=0;
        for( i=0; i<V_MAX; i++) if( visit[i]==0 ) bfs(i);
}
```

範例 10-3

廣度優先搜尋分析

輸出結果

->A->G->F->C->B->E->D



廣度優先搜尋樹

BFS tree

- 效能分析

  時間複雜度: 與使用的資料結構空間複雜度相同

  1) 相鄰串列 O(V+E)　2)相鄰矩陣 $O(V^2)$

  空間複雜度:　O(V+E)

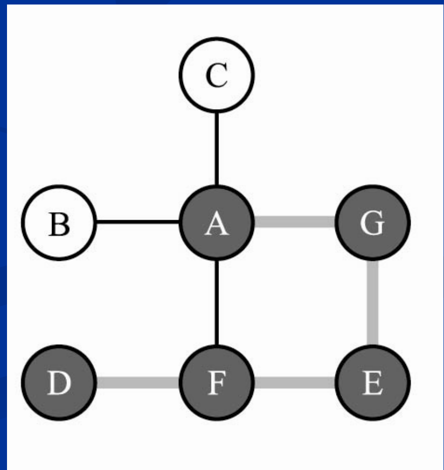
深度優先搜尋 -- 偵測迴路

範例 10-4

```
static int seq=1, cycle=0;
void dfs_cycle(int v, int p){
   struct node *t;
   if (cycle != 0) return;
   printf("->%c", v+'A');  visit[v]=seq;   seq++;
   for( t=adj[v]; t != NULL && cycle ==0 ; t=t->next )
        if( visit[t->v] == 0 ) dfs_cycle(t->v, v);
        else if ( t->v != p ) {
              printf("->[%c]\n", t->v+'A');
              cycle++;
              return;
        }
}
```

S={A, G, E, F, D}

->A->G->E->F->D->[A]

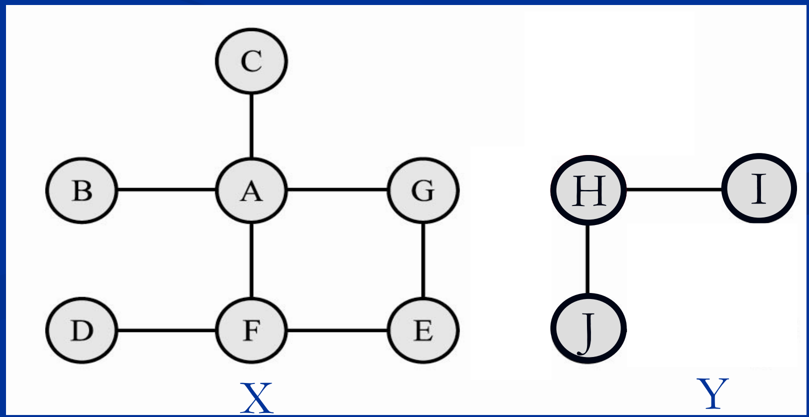


深度優先搜尋 -- 連接元件

範例 10-5

```
main(){
   int i, count=0, start[V_MAX];
   adj_list();
   for( i=0; i<V_MAX; i++) visit[i]=0;
   for( i=0; i<V_MAX; i++)
        if( visit[i]==0 ) {
             dfs(i);
             start[count]=i; count++;
             printf("\n");
        }
}
```

->A->G->E->F->D->C->B
->H->J->I

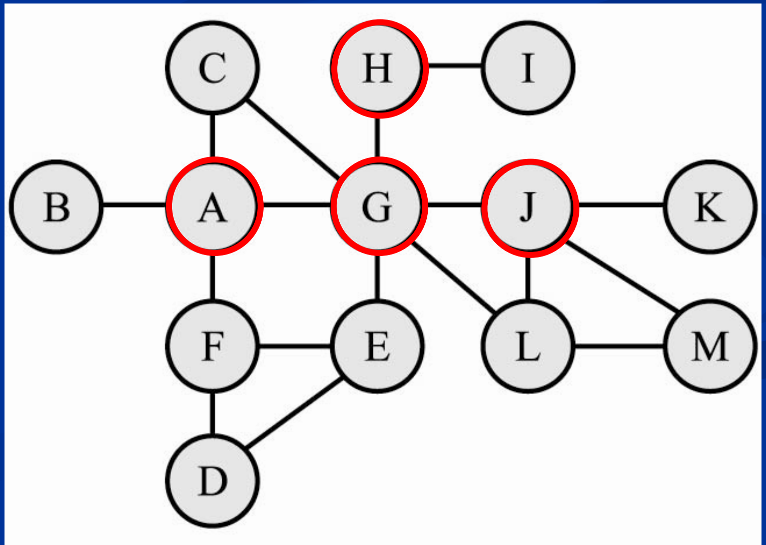


關節點

- 關節點(articulation point)
  當一個節點從連接的圖形中移除後，會導致整個圖形分裂成數個連接元件。

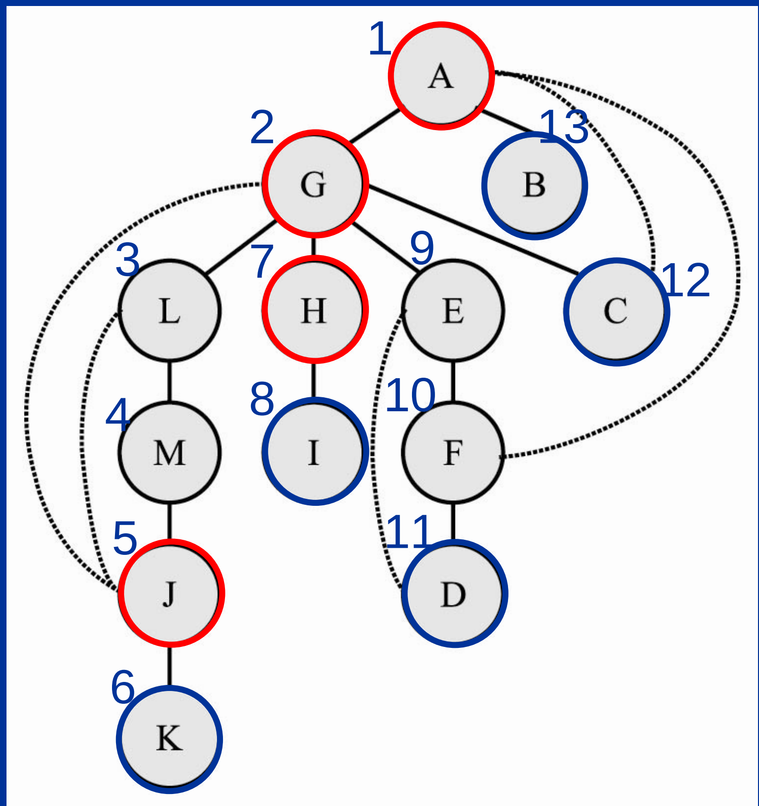
- 雙連通性(biconnectivity) 一個圖形不存在關節點

關節點 {A, H, G, J}



深度優先搜尋 -- 關節點

- 只要節點 X 的子節點都具有一替代路徑通往 X 的 proper ancestor ，就不是關節點
- 根節點有兩個以上子節點即爲關節點，因爲只能通過根節點相連



關節點 {A, H, G, J}

深度優先搜尋樹

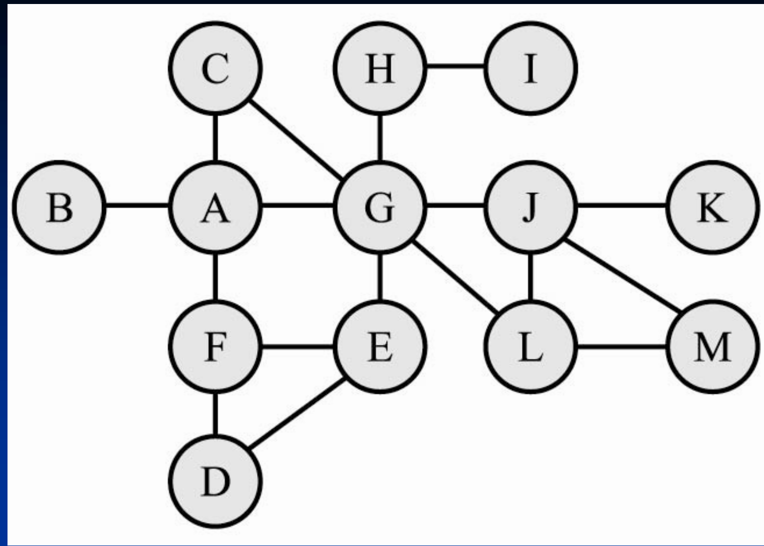
範例 10-6

```
static int seq=1;
int dfs_arti(int v){  struct node* t;      int m, min;
   printf("->%c", v+'A'); visit[v]=min=seq; seq++;
   for( t=adj[v]; t!=NULL; t=t->next ){
        if (visit[t->v] == 0 ){
                m=dfs_arti(t->v);
                if (m<min) min = m;
                if (m>= visit[v]) printf("@%c", v+'A');
        } else if (visit[t->v] < min) min=visit[t->v];
   }
   return min;
}
```

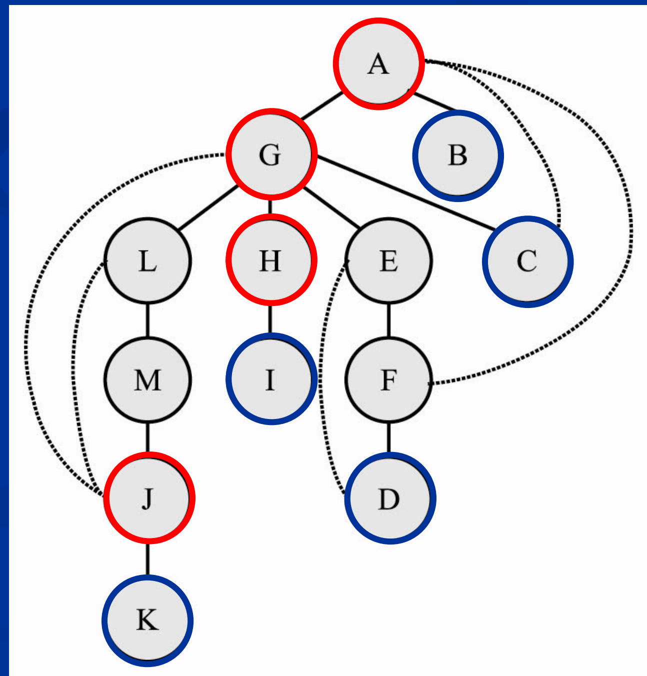
子節點無法連通到上層節點

範例 10-6

A -> G F C B
B -> A
C -> G A
D -> F E
E -> G F D
F -> E D A
G -> L J H E C A
H -> I G
I -> H
J -> M L K G
K -> J
L -> M J G
M -> L J



->A->G->L->M->J->K->H
->I->E->F->D->C->B



->A->G->L->M->J->K@J@G->H
->I@H@G->E->F->D->C->B@A