CHAPTER 6

GRAPHS

All the programs in this file are selected from
Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
"Fundamentals of Data Structures in C",

Definition

☐ A graph G consists of two sets
  – a finite, <u>nonempty</u> set of vertices V(G)
  – a finite, possible empty set of edges E(G)
  – G(V, E) represents a graph

☐ An undirected graph is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$

☐ A directed graph is one in which each edge is a directed pair of vertices, $\langle v_0, v_1 \rangle \mathrel{!=} \langle v_1, v_0 \rangle$

tail   head
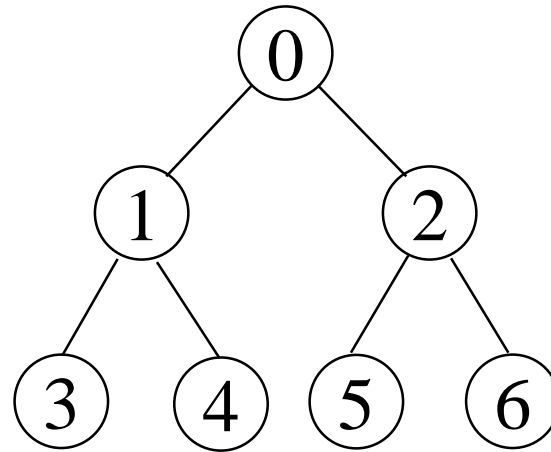
Examples for Graph
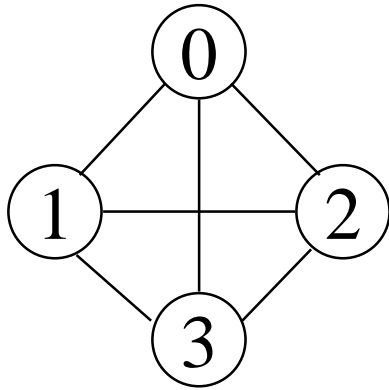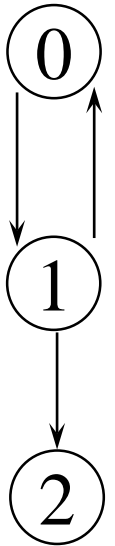

G_1

complete graph

G_2

incomplete graph

G_3

$V(G_1)=\{0,1,2,3\}$ $E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$
$V(G_2)=\{0,1,2,3,4,5,6\}$ $E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$
$V(G_3)=\{0,1,2\}$ $E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

complete undirected graph: n(n-1)/2 edges
complete directed graph: n(n-1) edges

Complete Graph

□ A complete graph is a graph that <u>has the maximum number of edges</u>

- for undirected graph with n vertices, the maximum number of edges is $n(n-1)/2$

- for directed graph with n vertices, the maximum number of edges is $n(n-1)$

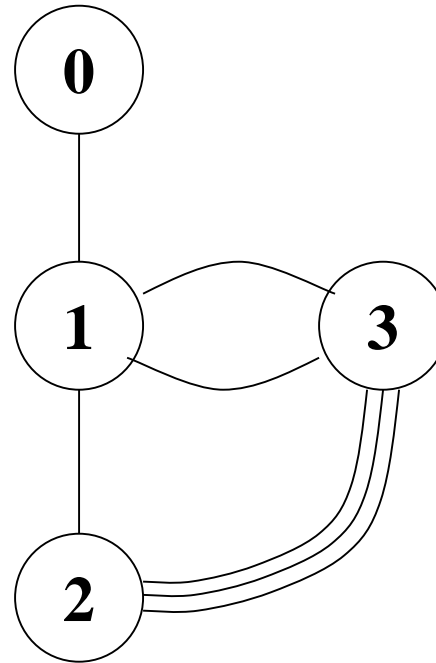- example: G1 is a complete graph

Adjacent and Incident

☐ If (v_0, v_1) is an edge in an undirected graph,
  – v_0 and v_1 are adjacent
  – The <u>edge (v_0, v_1) is incident</u> on vertices v_0 and v_1

☐ If $<v_0, v_1>$ is an edge in a <u>directed graph</u>
  – v_0 is adjacent **to** v_1, and v_1 is adjacent **from** v_0
  – The edge $<v_0, v_1>$ is incident on v_0 and v_1

***Figure 6.3:**Example of a graph with feedback loops and a multigraph

self edge

(a)
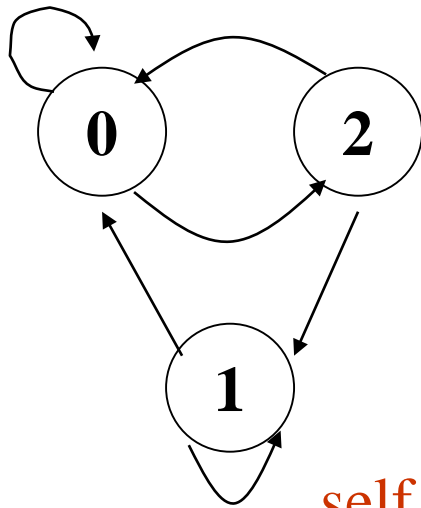
(b)   multigraph

multiple occurrences of the same edge
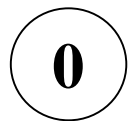
Subgraph and Path

☐ A subgraph of G is a graph G' such that V(G') is a subset of V(G) and E(G') is a subset of E(G)

☐ A path from vertex v_p to vertex v_q in a graph G, is a sequence of vertices, $v_p, v_{i1}, v_{i2}, …, v_{in}, v_q$, such that $(v_p, v_{i1}), (v_{i1}, v_{i2}), …, (v_{in}, v_q)$ are edges in an undirected graph

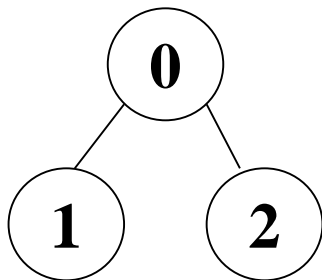☐ The length of a path is the number of edges on it

Figure 6.4: subgraphs of G$_1$ and G$_3$
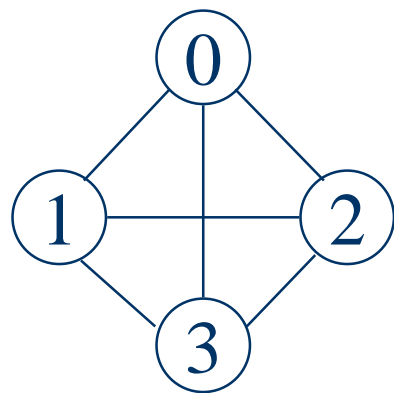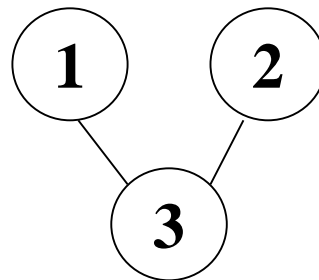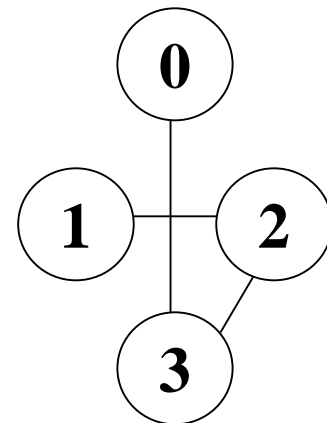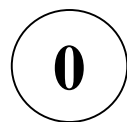

G$_1$

(i)   (ii)   (iii)   (iv)

(a) Some of the subgraph of G$_1$

單一

分開

G$_3$
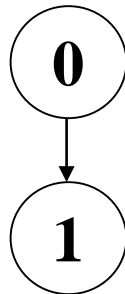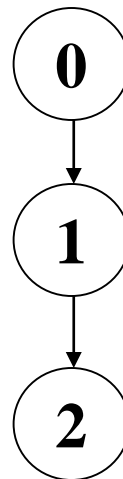
(i)   (ii)   (iii)   (iv)
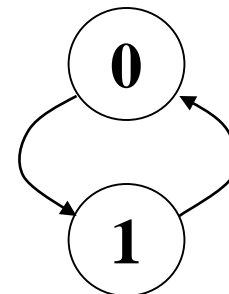
(b) Some of the subgraph of G$_3$

8

Simple Path and Style

☐ A simple path is a path in which all vertices, except possibly the first and the last, are distinct

☐ A cycle is a simple path in which the first and the last vertices are the same

☐ In an undirected graph G, two vertices, v_0 and v_1, are connected iff there is a path in G from v_0 to v_1

☐ An undirected graph is connected iff for every pair of distinct vertices v_i, v_j, there is a path from v_i to v_j

Connected
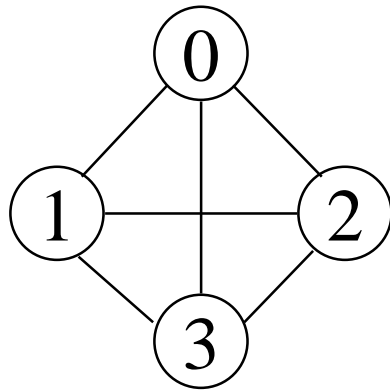
G_1
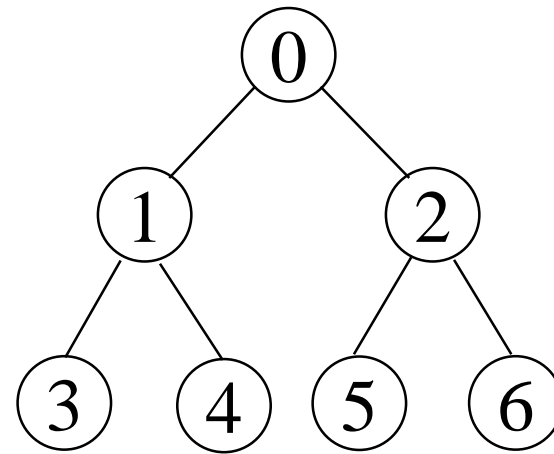
G_2

tree (acyclic graph)

connected component

Connected Component

☐ A connected component of an undirected graph is a maximal connected subgraph.

☐ A tree is a graph that is connected and acyclic (i.e., has no cycles).

☐ A directed graph is strongly connected if there is a directed path from v_i to v_j and also from v_j to v_i.

☐ A strongly connected component is a maximal subgraph that is strongly connected.

11

*Figure 6.5: A graph with two connected components (p.262)

connected component (maximal connected subgraph)

H_1

H_2

G_4 (not connected)

strongly connected component
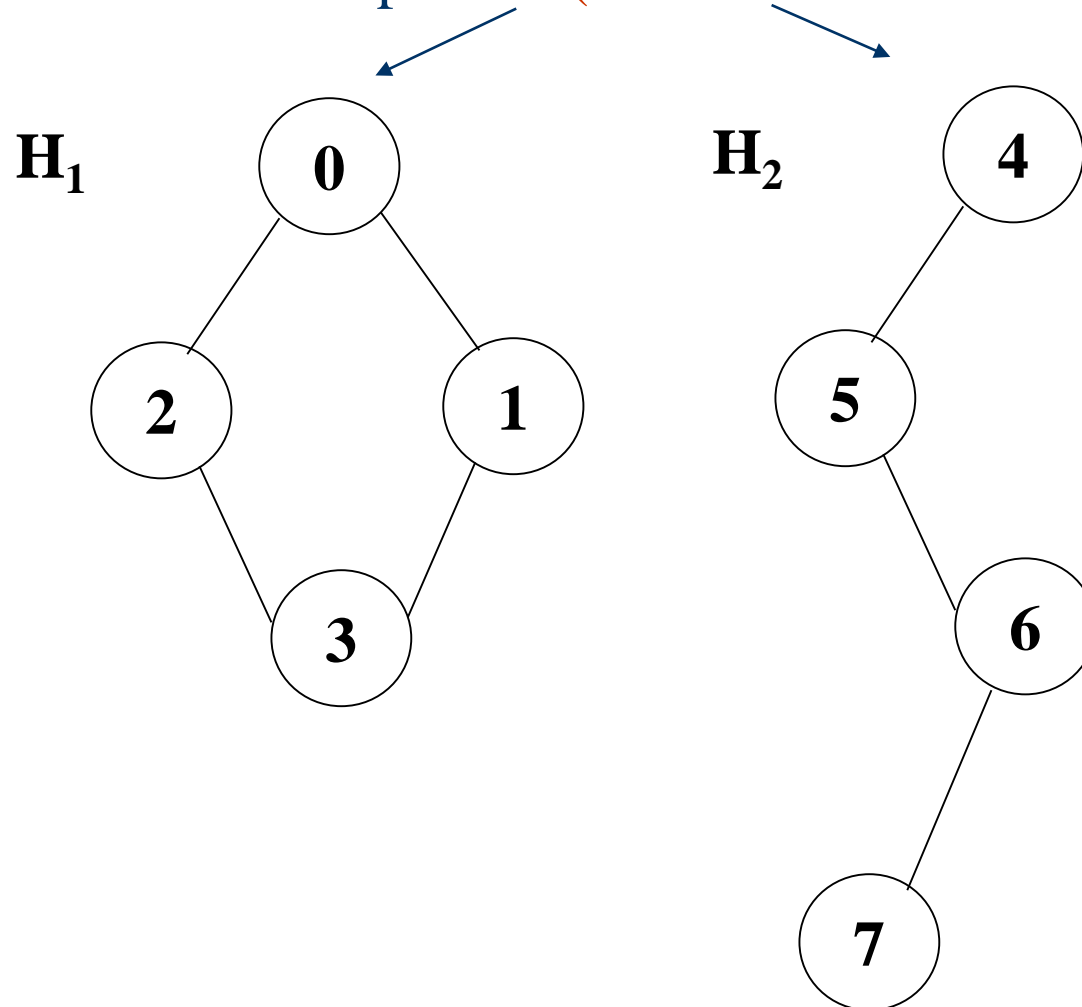
not strongly connected (maximal strongly connected subgraph)

G_3
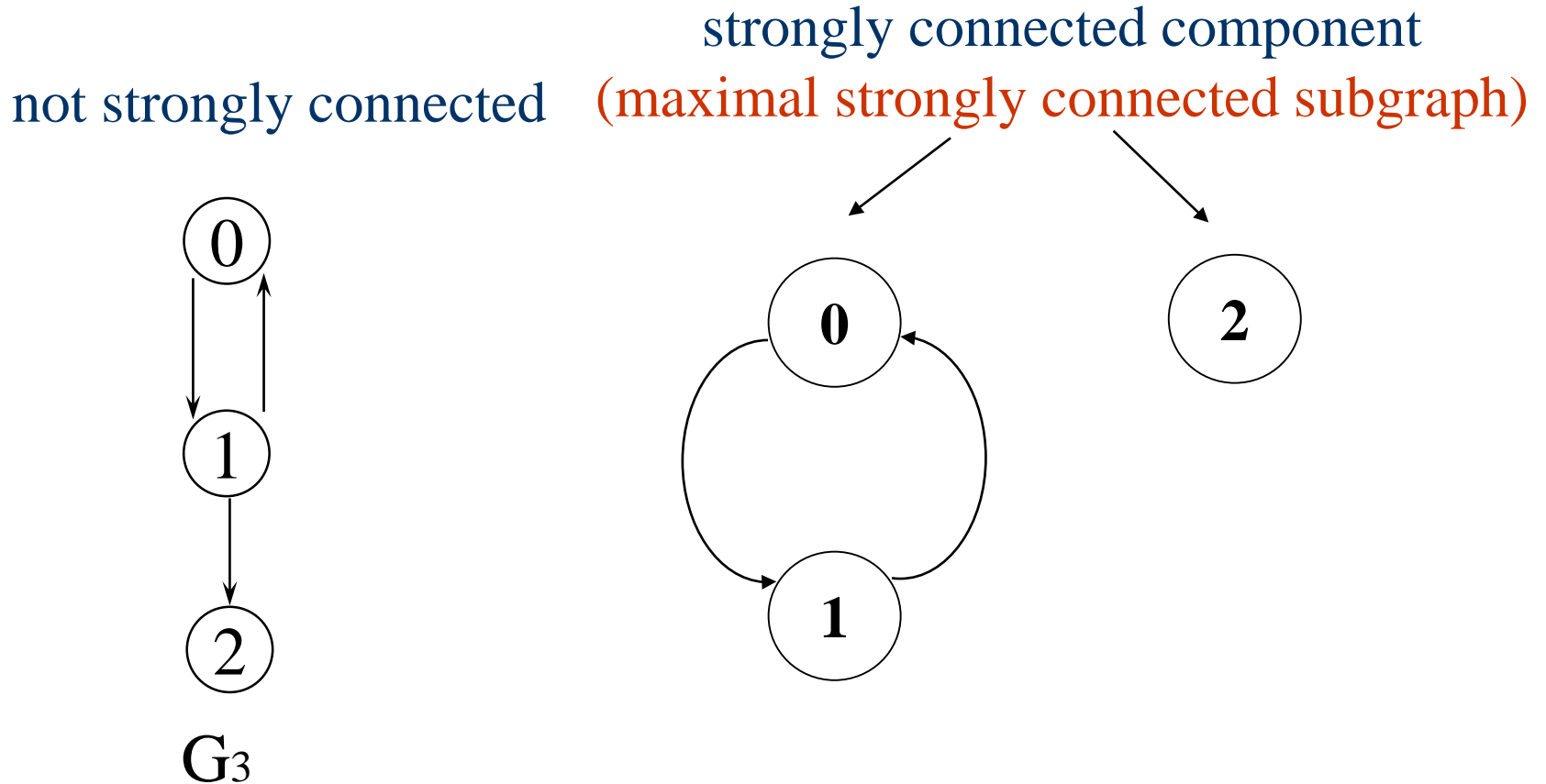
Degree

☐ The degree of a vertex is <u>the number of edges</u> incident to that vertex

☐ For directed graph,

- the in-degree of a vertex v is the number of edges that have v as the <u>head</u>
- the out-degree of a vertex v is the number of edges that have v as the <u>tail</u>
- if d_i is the degree of a vertex i in a graph G with n vertices and e edges, the number of edges is

$$e = (\sum_{0}^{n-1} d_i) / 2$$

undirected graph

degree
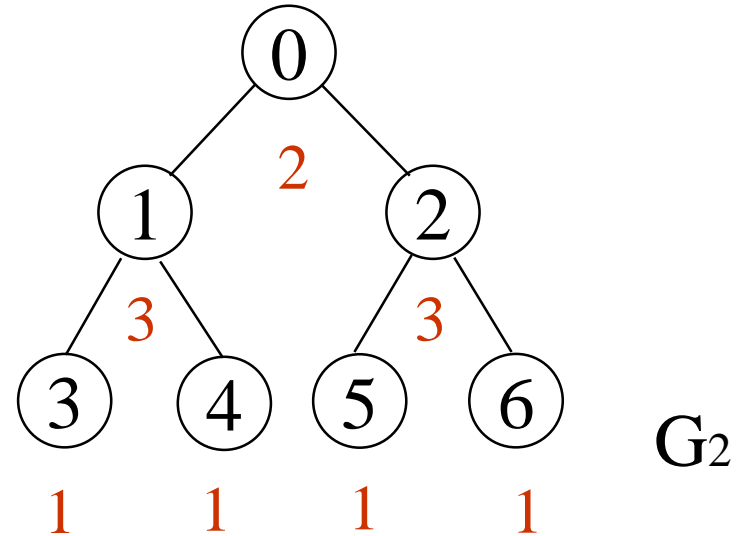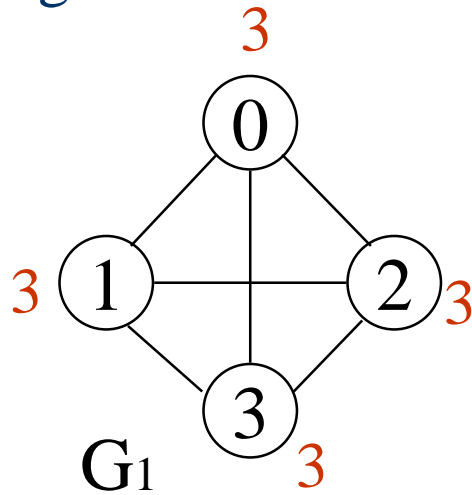
directed graph
in-degree
out-degree

0   in:1, out: 1

1   in: 1, out: 2

2   in: 1, out: 0

G₃
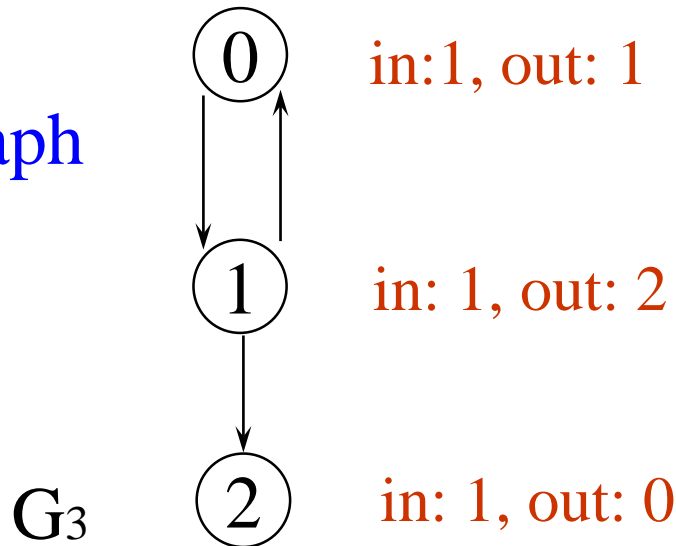
ADT for Graph

structure Graph is

 objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

 functions: for all $graph \in Graph$, v, v_1 and $v_2 \in Vertices$

 Graph Create()::=return an empty graph

 Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no incident edge.

 Graph InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*

 Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

 Graph DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is removed

 Boolean IsEmpty(*graph*)::= if (*graph==empty graph*) return TRUE else return FALSE

 List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

Graph Representations

☐ Adjacency Matrix

☐ Adjacency Lists

☐ Adjacency Multilists

Adjacency Matrix

☐ Let G=(V,E) be a graph with n vertices.

☐ The adjacency matrix of G is a two-dimensional $n*n$ array, say adj_mat

☐ If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1

☐ If there is no such edge in E(G), adj_mat[i][j]=0

☐ The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric

Examples for Adjacency Matrix
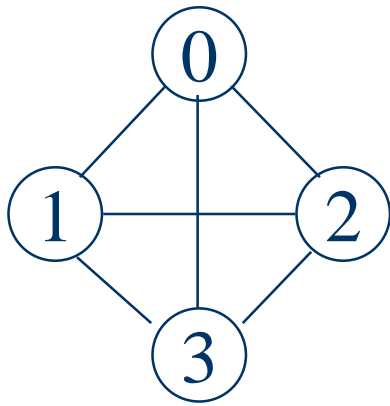
$$G_1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

$$G_2 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$
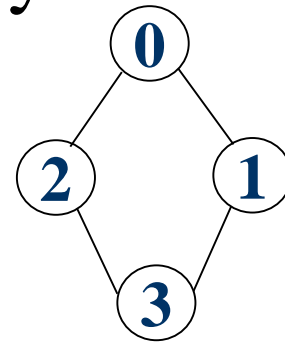
$$G_4 = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$
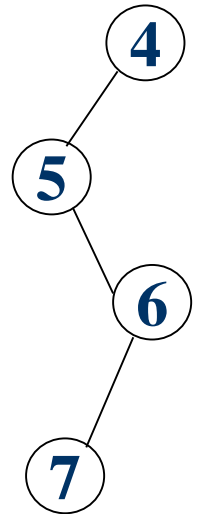
symmetric

undirected: n*(n+1)/2

directed: n^2

Memory request

Merits of Adjacency Matrix

☐ From the adjacency matrix, to determine the connection of vertices is easy

☐ The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$

☐ For a directed graph, <u>the row sum is the out_degree</u>, <u>while the column sum is the in_degree</u>

$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

Data Structures for Adjacency Lists

Each row in adjacency matrix is represented as an adjacency list.

```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```
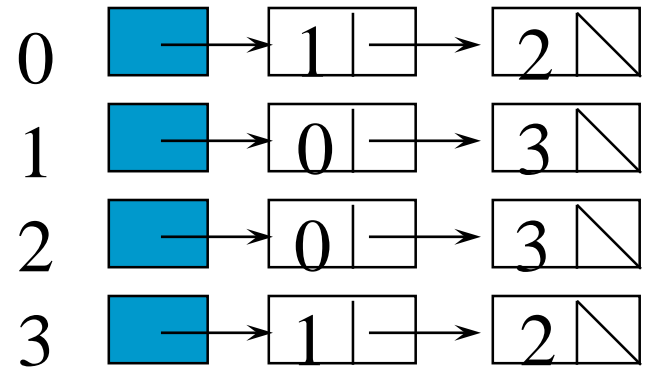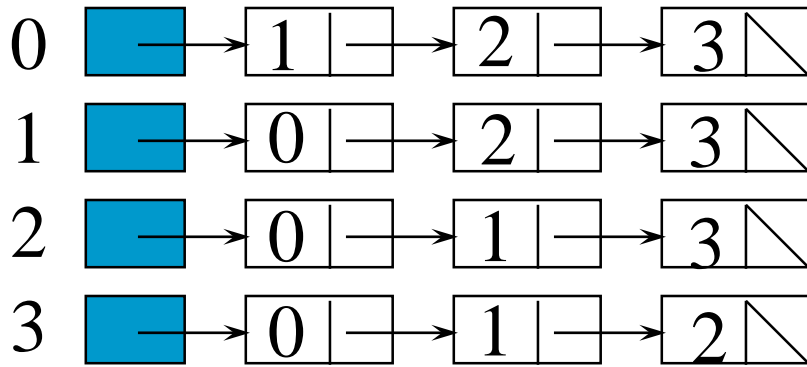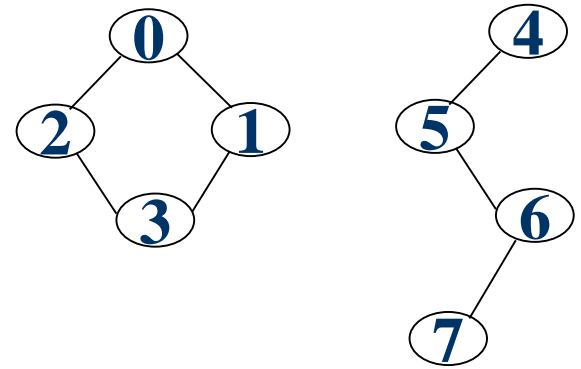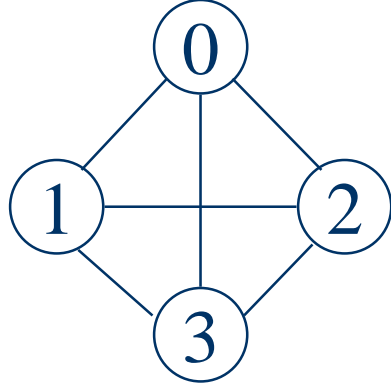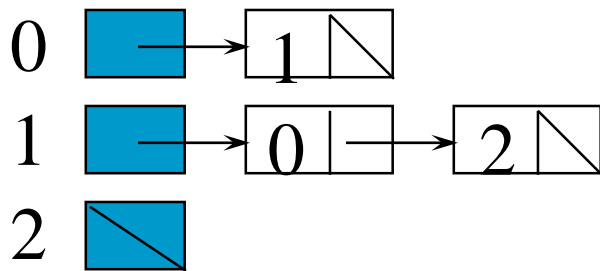
An undirected graph with *n* vertices and *e* edges ==> n head nodes and 2e list nodes

Alternate order adjacency list for G_1

Order is of no significance.

headnodes   vertax link
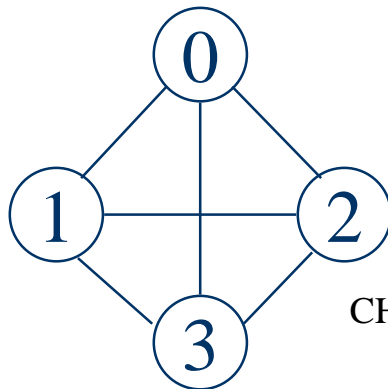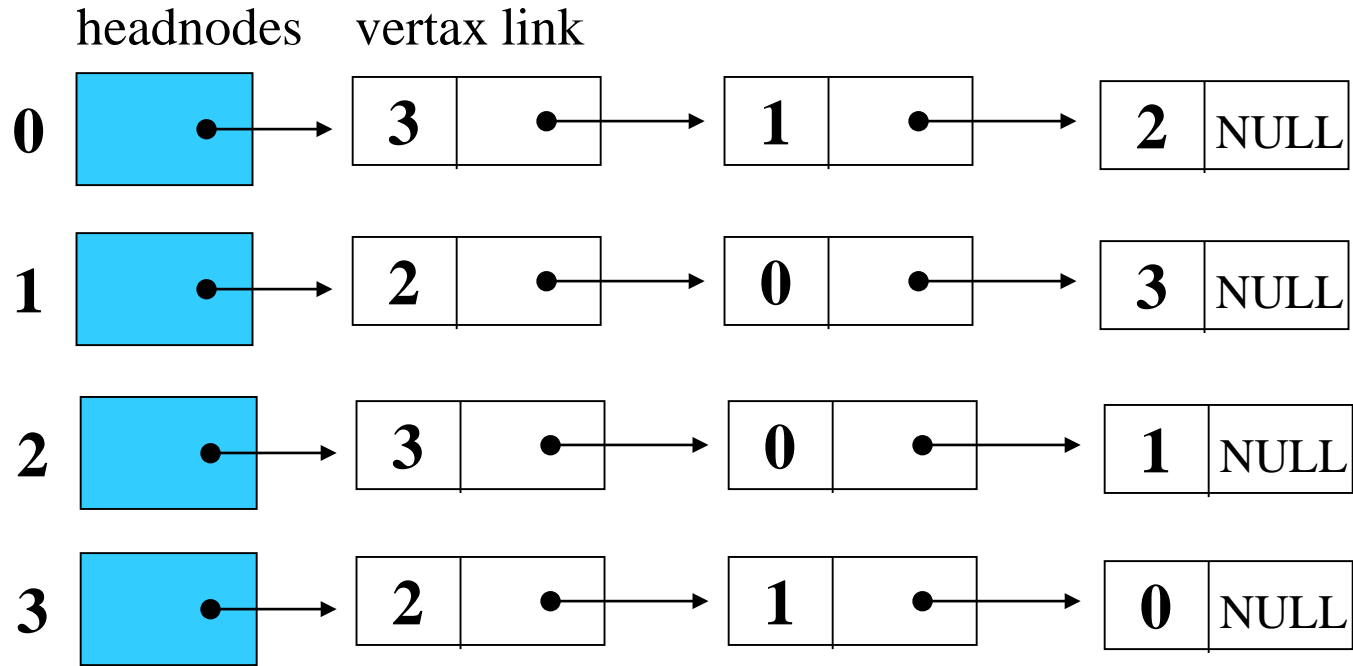
Interesting Operations

□degree of a vertex in an undirected graph

 – # of nodes in adjacency list

□# of edges in a graph

 – determined in O(n+e)

□out-degree of a vertex in a directed graph

 – # of nodes in its adjacency list

□in-degree of a vertex in a directed graph

 – traverse the whole data structure

CHAPTER 6

24

An undirected graph with *n* vertices and *e* edges

Figure 6.10: Inverse adjacency list for G_3
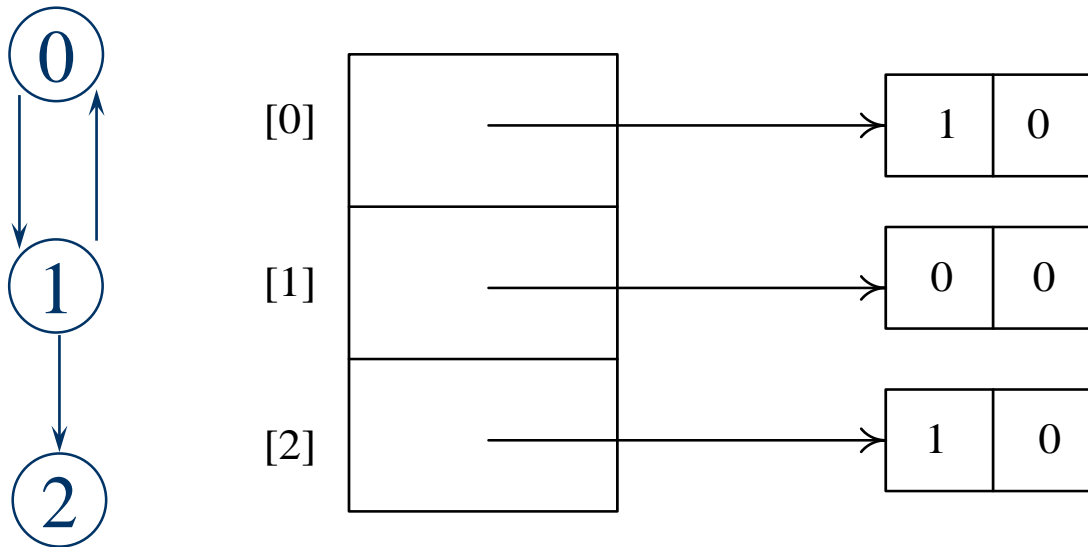
Determine in-degree of a vertex in a fast way.

Figure 6.11: Orthogonal representation for graph

標頭節點
（顯示兩次）
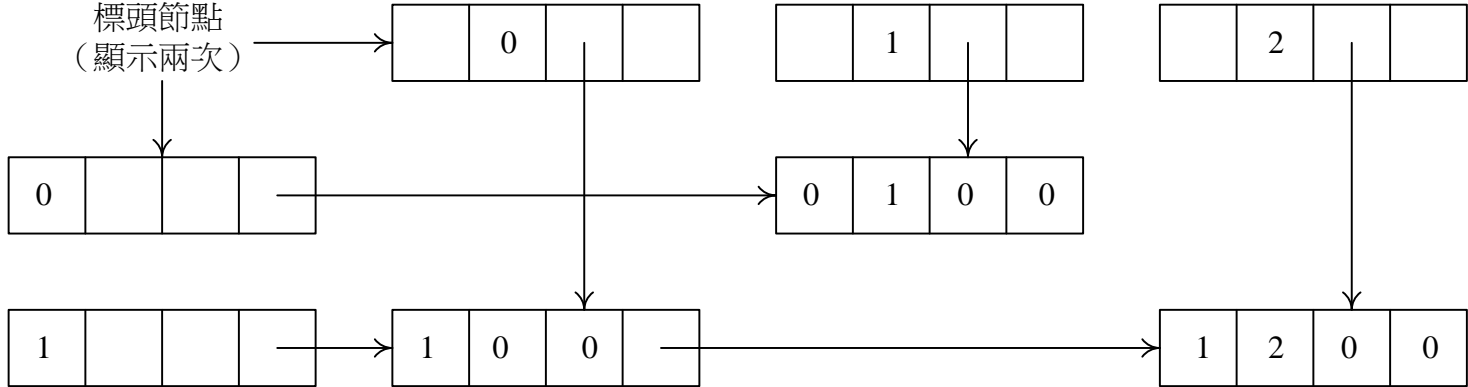
| 0 | | | |

| 1 | | | |

| 2 | | | |

| 0 | | | |

| 0 | 1 | 0 | 0 |

| 1 | | | |

| 1 | 0 | 0 | |

| 1 | 2 | 0 | 0 |

| 2 | | | 0 |

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

| row | col | column link for head | row link for tail |

Compact Representation
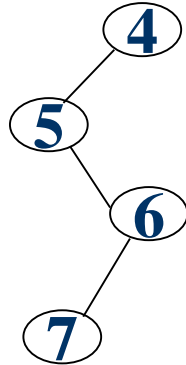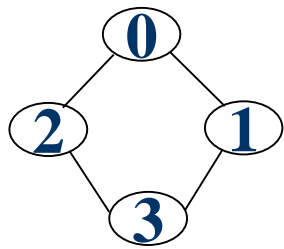
node[0] … node[n-1]: starting point for vertices
node[n]: n+2e+1
node[n+1] … node[n+2e]: head node of edge

[0]	9		[8]	23	
[1]	11	0	[9]	1	4
[2]	13		[10]	2	5
[3]	15	1	[11]	0	
[4]	17		[12]	3	6
[5]	18	2	[13]	0	
[6]	20		[14]	3	7
[7]	22	3	[15]	1	

[16]	2
[17]	5
[18]	4
[19]	6
[20]	5
[21]	7
[22]	6

Adjacency Multilists

☐ An edge in an undirected graph is represented by two nodes in adjacency list representation.

☐ Adjacency Multilists

–nodes may be shared among several lists.
(**a node is shared by two different paths**)

marked	vertex1	vertex2	path1	path2

Adjacency Multilists

```
typedef struct edge *edge_pointer;
typedef struct edge {
    short int marked;
    int vertex1, vertex2;
    edge_pointer path1, path2;
};
edge_pointer graph[MAX_VERTICES];
```

marked	vertex1	vertex2	path1	path2

Example for Adjacency Multlists

Lists: vertex 0: N0->N1->N2, vertex 1: N0->N3->N4
vertex 2: N1->N3->N5, vertex 3: N2->N4->N5

		(1,0)		
N0	⊠	0 1	N1	N3	edge (0,1)

(2,0)
N1 ⊠ 0 2 N2 N3 edge (0,2)

(3,0)
N2 ⊠ 0 3 N4 edge (0,3)

(2,1)
N3 ⊠ 1 2 N4 N5 edge (1,2)

(3,1)
N4 ⊠ 1 3 N5 edge (1,3)

(3,2)
N5 ⊠ 2 3 edge (2,3)

six edges
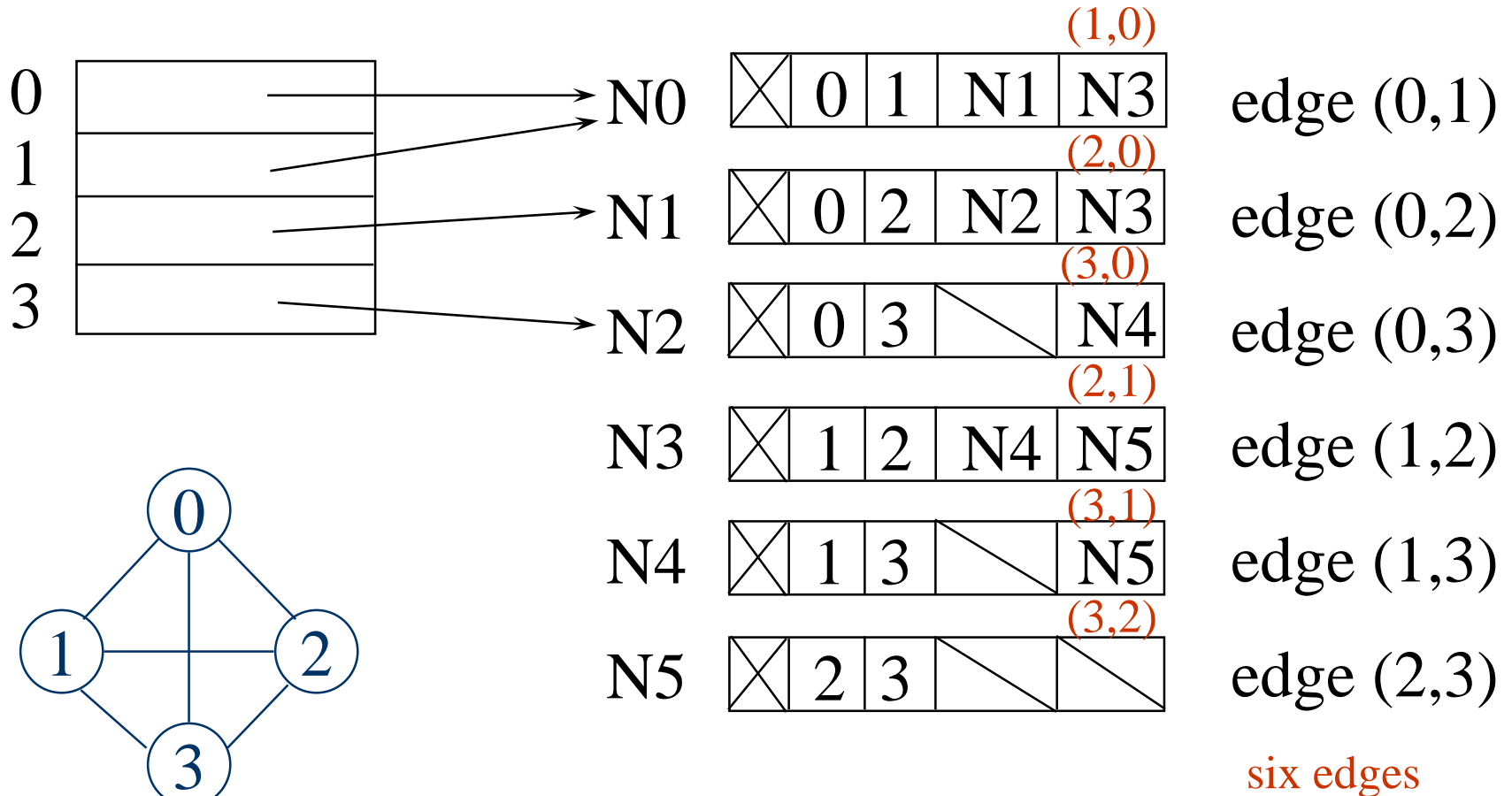
Some Graph Operations

☐ Traversal

Given G=(V,E) and vertex v, find all w∈V, such that w connects v.

– Depth First Search (DFS)

preorder tree traversal

– Breadth First Search (BFS)

level order tree traversal

☐ Connected Components

☐ Spanning Trees

*Figure 6.16: Graph *G* and its adjacency lists

depth first search: v0, v1, v3, v7, v4, v5, v2, v6
breadth first search: v0, v1, v2, v3, v4, v5, v6, v7

(a)

(b)
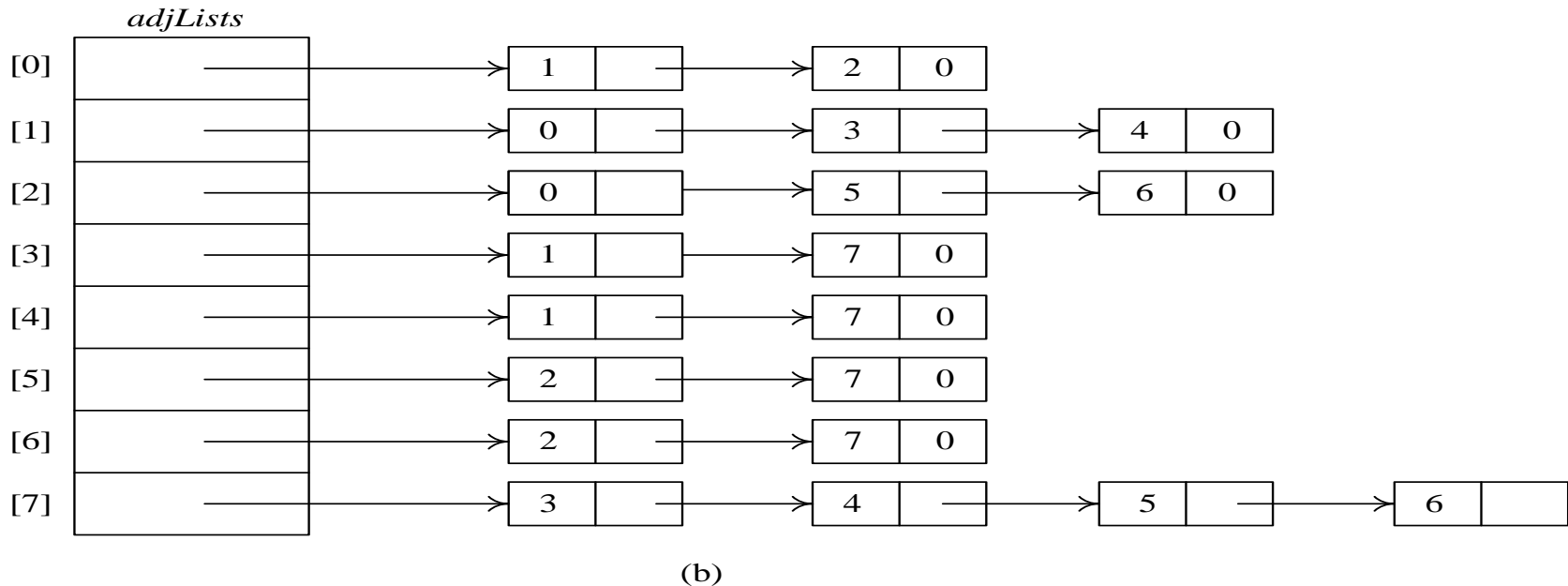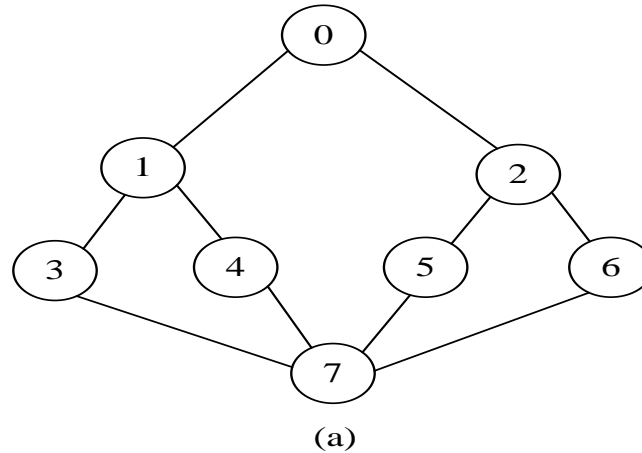
Depth First Search

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```
void dfs(int v)
{
  node_pointer w;
  visited[v]= TRUE;
  printf("%5d", v);
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}
```

Data structure
adjacency list: O(e)
adjacency matrix: O(n^2)

33

DFS and BFS

☐ DFS
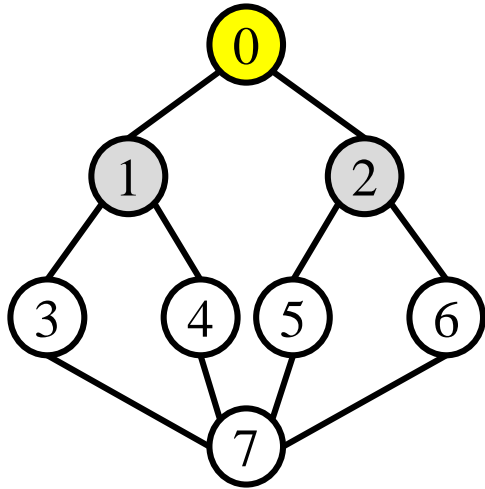
從頂點0開始
有頂點1, 2可以選
選數字小的頂點1

DFS and BFS

☐ DFS
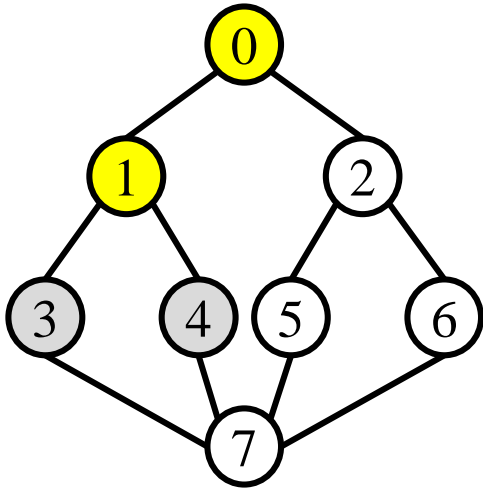
頂點1有頂點3, 4可以選
選數字小的頂點3

DFS and BFS

☐ DFS
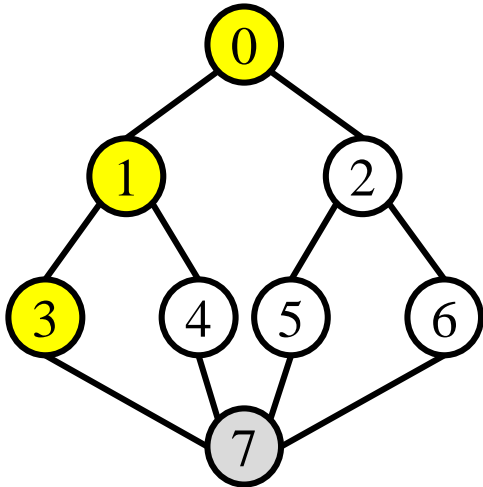
頂點3有頂點1, 7可以選
1選過了，所以選頂點7

DFS and BFS

☐ DFS
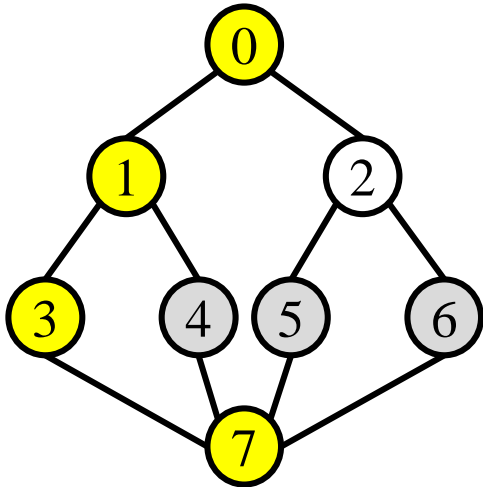
頂點7有頂點3, 4, 5, 6可以選
3選過了
有頂點4, 5, 6可以選
選數字小的頂點4

DFS and BFS

☐ DFS
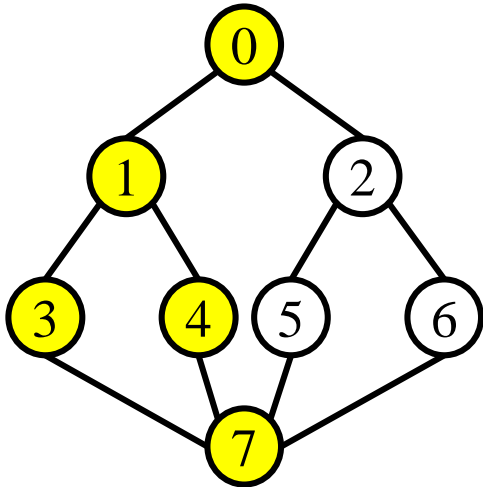
頂點4有頂點1,7可以選
1,7都選過了
回上一個頂點，也就是頂
點7

DFS and BFS

☐ DFS
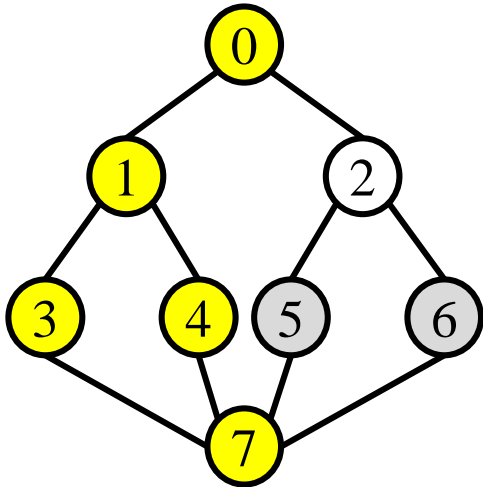
頂點7有頂點3, 4, 5, 6可以
選
3, 4選過了
有頂點 5, 6可以選
選數字小的頂點5

DFS and BFS

□DFS
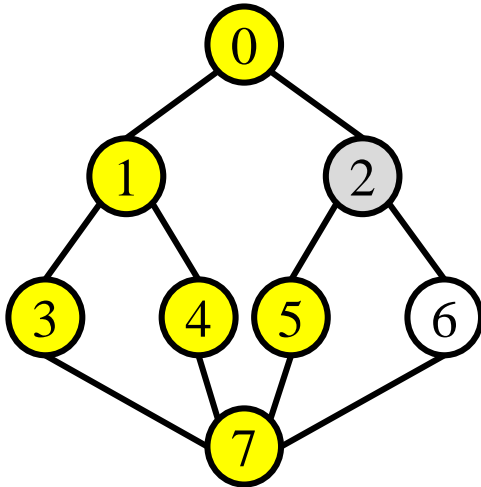
頂點5有頂點2, 7可以選
7選過了，所以選頂點2

DFS and BFS

☐ DFS
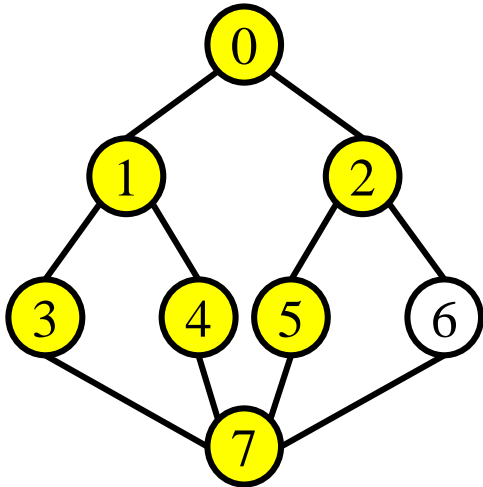
頂點2有頂點0, 5, 6可以選
0, 5選過了，所以選頂點6

DFS and BFS

□ DFS
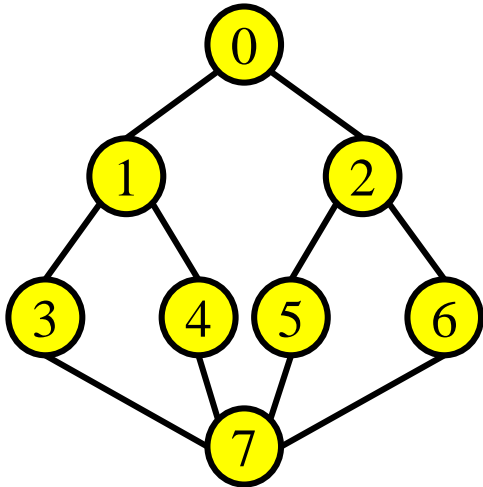
Breadth First Search

```
typedef struct queue *queue_pointer;
typedef struct queue {
    int vertex;
    queue_pointer link;
};
void addq(int);
int deleteq();
```

Breadth First Search *(Continued)*

```
void bfs(int v)
{
  node_pointer w;
  queue_pointer front, rear;
  front = rear = NULL;
  printf("%5d", v);
  visited[v] = TRUE;
  addq(v);
```

adjacency list: O(e)
adjacency matrix: $O(n^2)$

```
while (front) {
  v= deleteq();
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex]) {
      printf("%5d", w->vertex);
      addq(w->vertex);
      visited[w->vertex] = TRUE;
    }/* unvisited vertices*/
}
}
```

DFS and BFS

☐ BFS
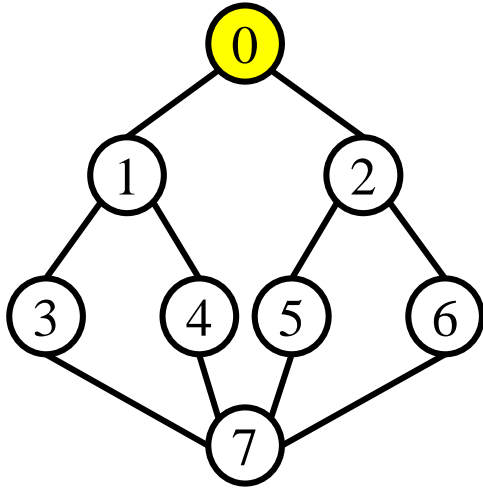
Queue

0

DFS and BFS

□ 步驟1
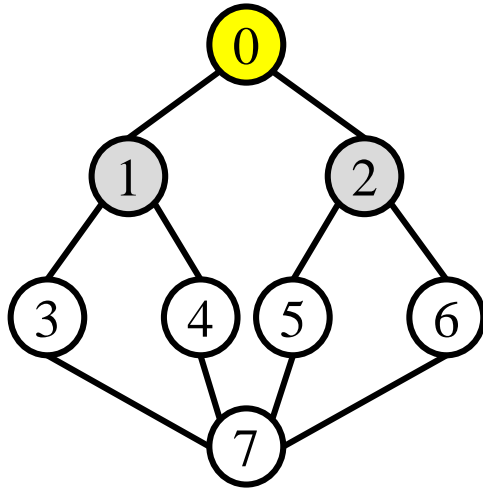
Queue

0   1   2

輸出：

DFS and BFS

□ 步驟2
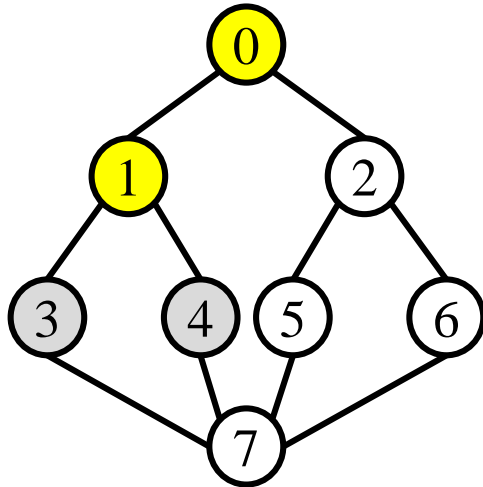
Queue

1   2   3   4

輸出 : 0

DFS and BFS

☐ 步驟3
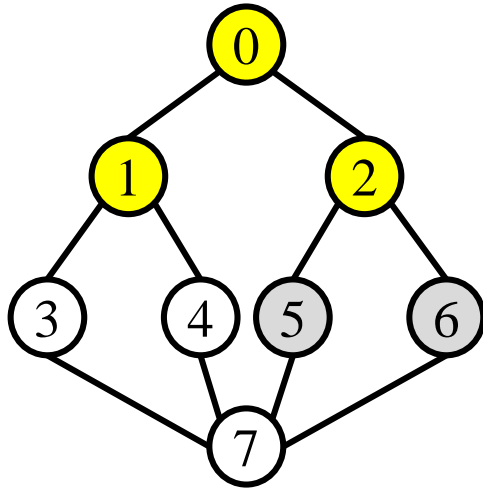
Queue

2　3　4　5　6

輸出：0　1

DFS and BFS

□ 步驟4
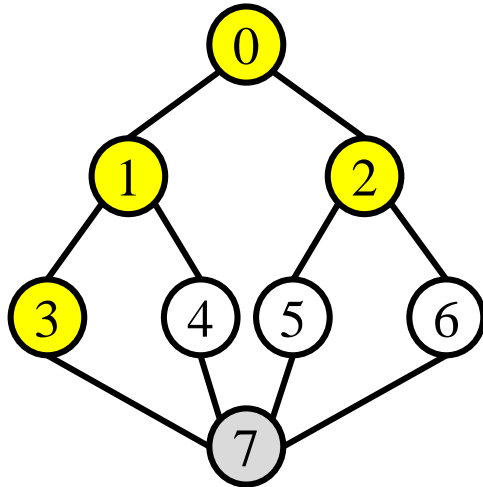
Queue

3　4　5　6　7

輸出: 0　1　2

DFS and BFS

□ 步驟5
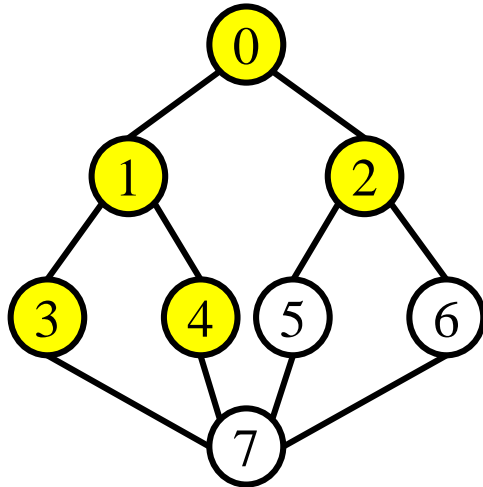
Queue

4 5 6 7

輸出: 0 1 2 3

DFS and BFS

□ 步驟6
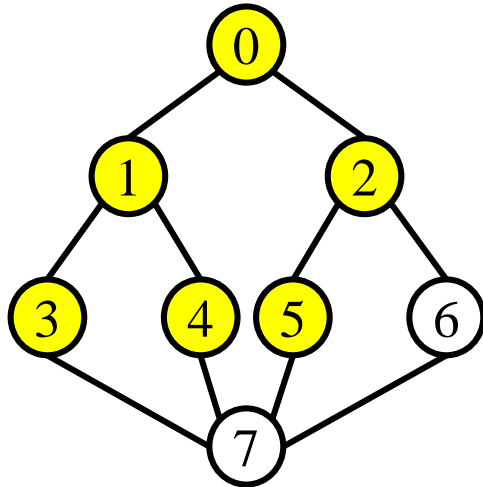
Queue

5 6 7

輸出 : 0 1 2 3 4

DFS and BFS

☐ 步驟7

Queue

6 7
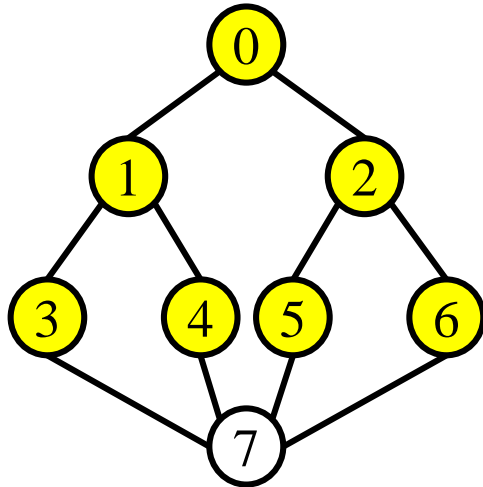
輸出：0 1 2 3 4 5

DFS and BFS

□ 步驟8

Queue

7
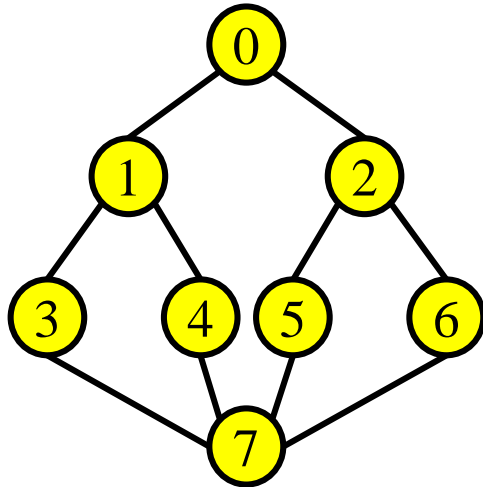
輸出: 0　1　2　3　4　5　6

Connected Components

```
void connected(void)
{  /*determine the connected components of
    a graph */
      for (i=0; i<n; i++) {
          if (!visited[i]) {
              dfs(i); // dfs→O(n)
              printf("\n");
          }
      }
}
```

adjacency list: O(n+e)
adjacency matrix: O(n^2)

Spanning Trees

☐ When graph G is connected, a depth first or breadth first search starting at any vertex will visit all vertices in G

☐ A spanning tree is any tree that consists solely of edges in G and that includes all the vertices

☐ E(G): T (tree edges) + N (nontree edges) where   T: set of edges used during search
                    N: set of remaining edges

Examples of Spanning Tree
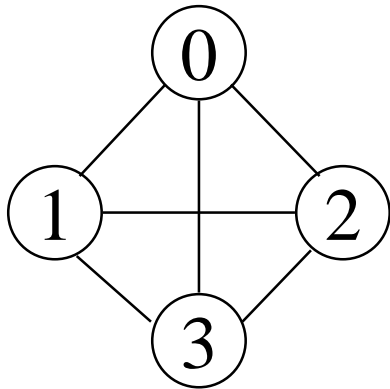
G1
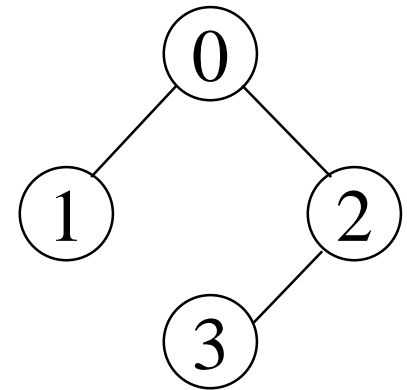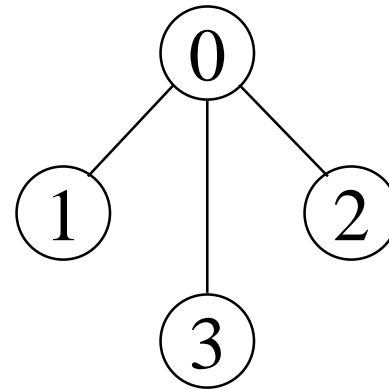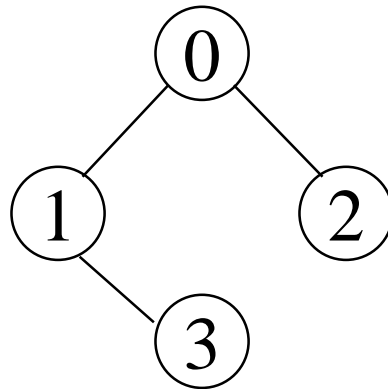
Possible spanning trees

Spanning Trees

□ Either dfs or bfs can be used to create a spanning tree

- When *dfs* is used, the resulting spanning tree is known as a **depth first spanning tree**
- When *bfs* is used, the resulting spanning tree is known as a **breadth first spanning tree**

□ While adding a nontree edge into any spanning tree, this will create a cycle

DFS vs BFS Spanning Tree

DFS Spanning

nontree edge cycle
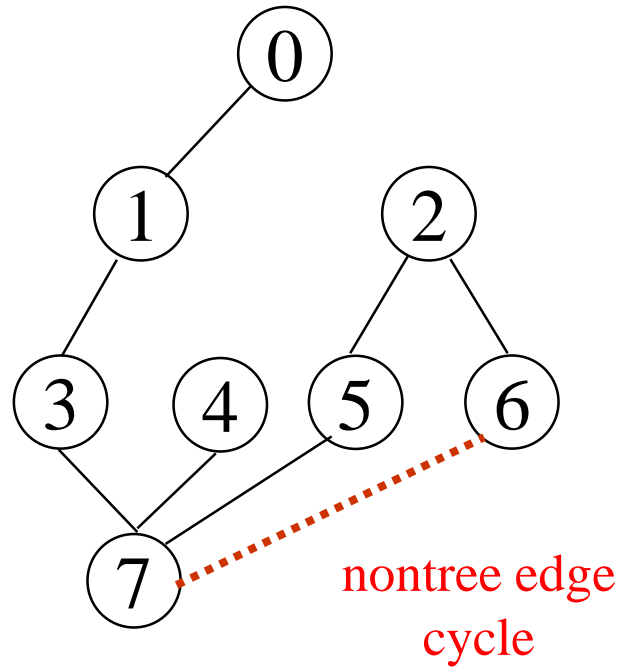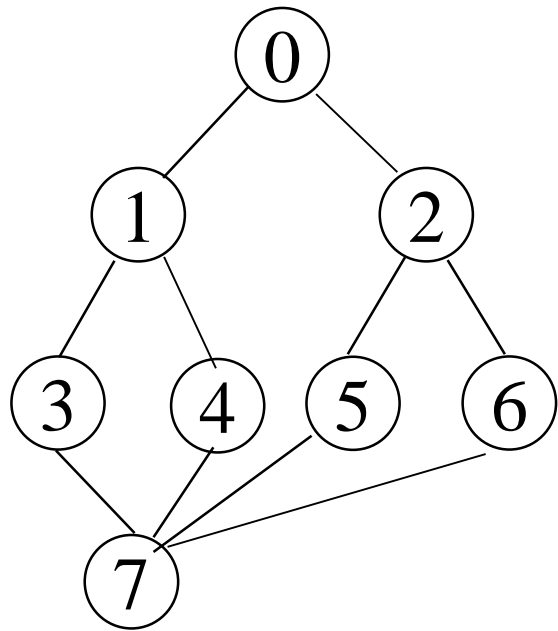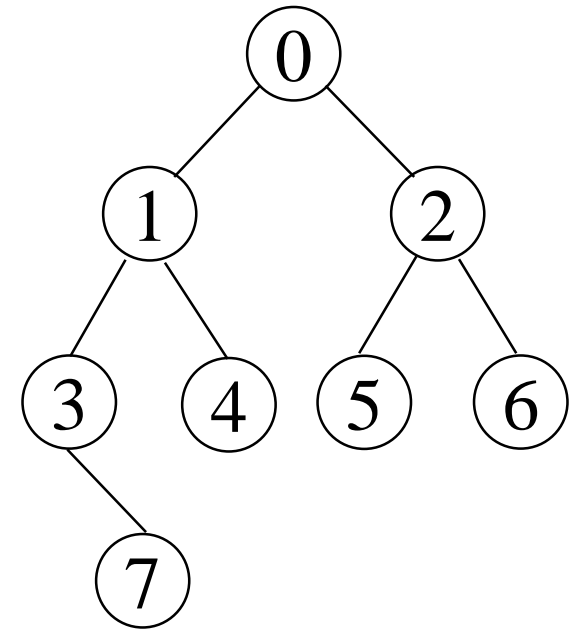
BFS Spanning

A spanning tree is a minimal subgraph, G', of G such that V(G')=V(G) and G' is connected.

Any connected graph with *n* vertices must have at least *n-1* edges.

A biconnected graph is a connected graph that has no articulation points.
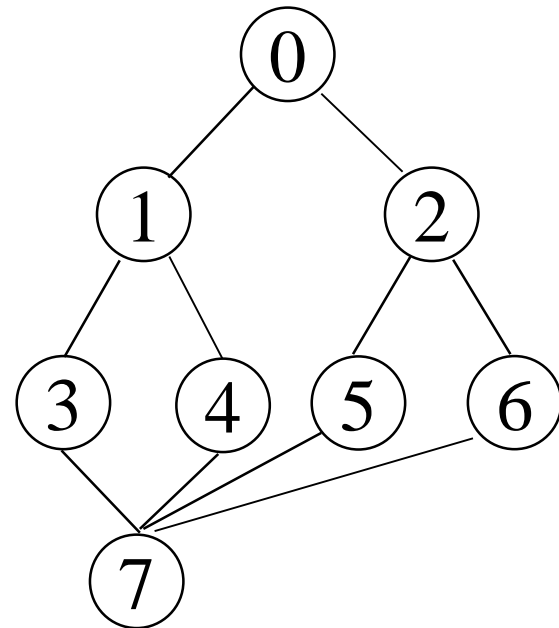
Articulation points

關節點是讓一張無向圖維持連通，不可或缺的點。只要從一張無向圖上移除了關節點（以及與之相連的邊），就會讓這張圖分離成更多部分，呈現不連通的狀態。
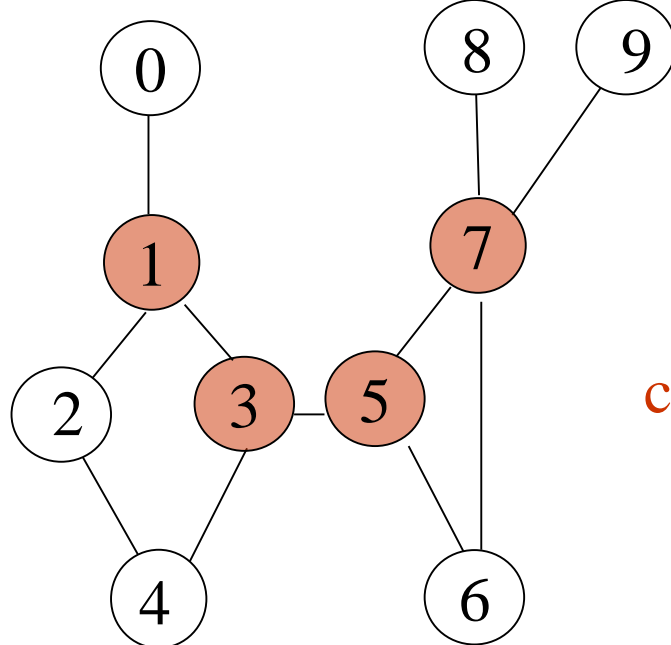
connected graph

two connected components
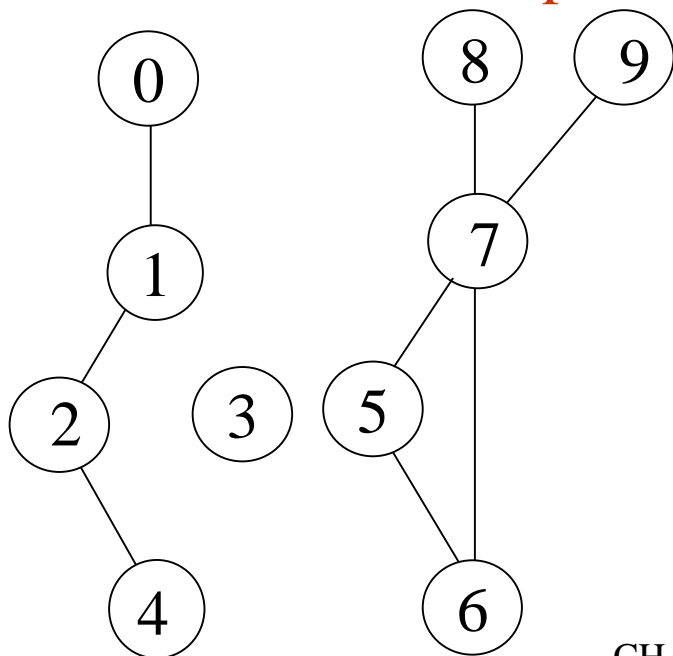
one connected graph
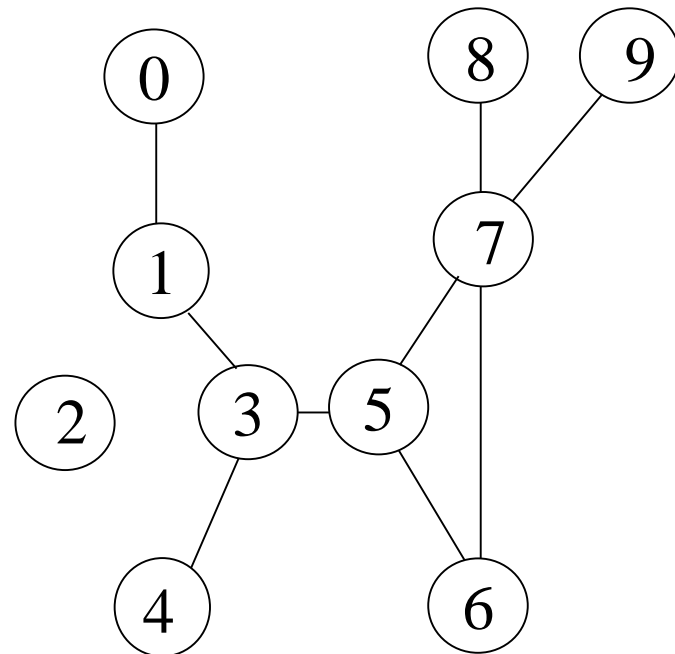
biconnected component: a maximal connected subgraph H
(no subgraph that is both biconnected and properly contains H)
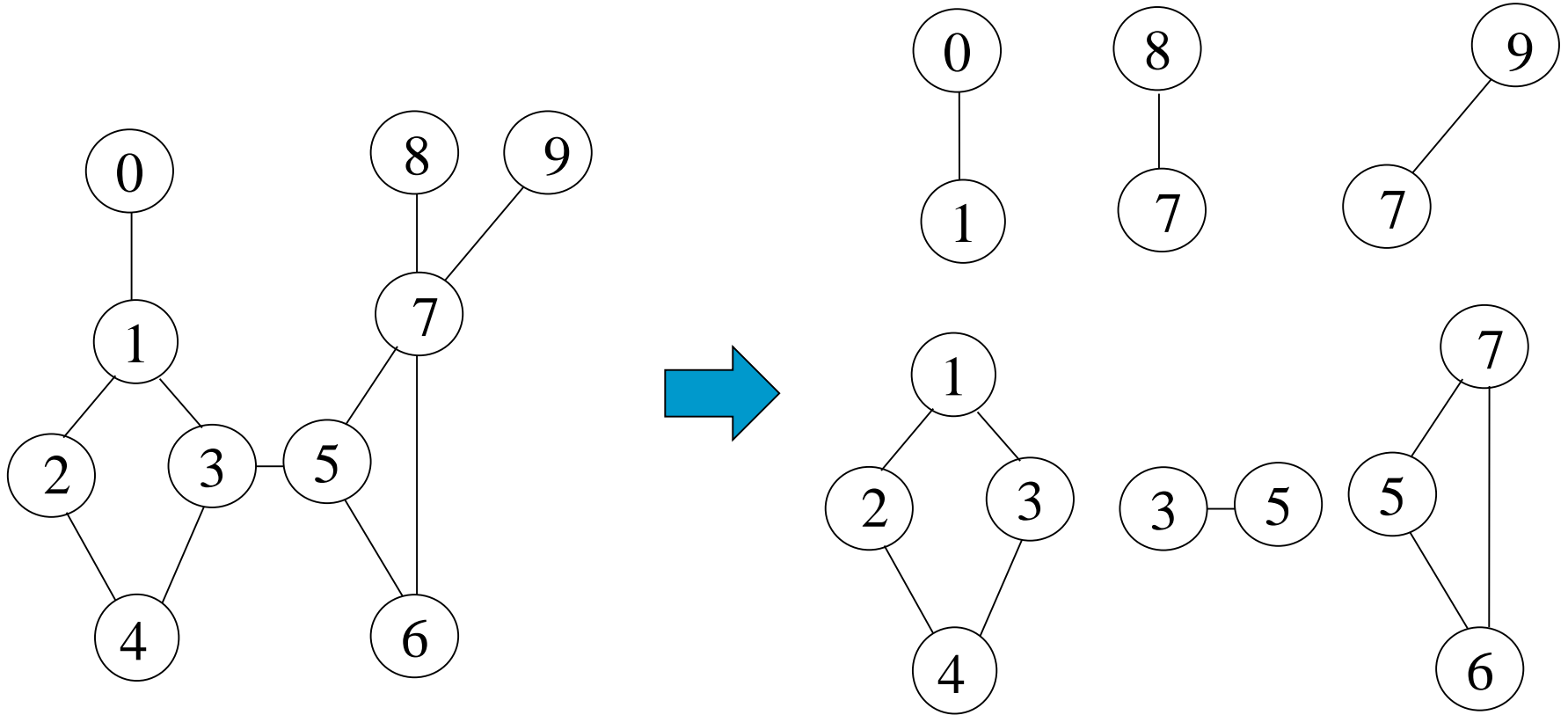
biconnected components

Find biconnected component of a connected undirected graph by depth first spanning tree

depth first number (dfn)
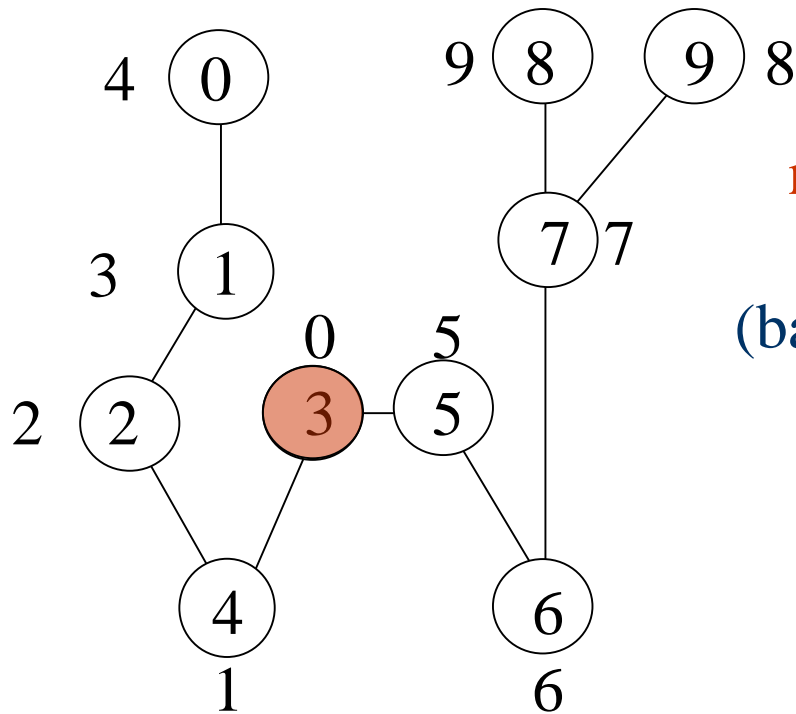
(a) depth first spanning tree
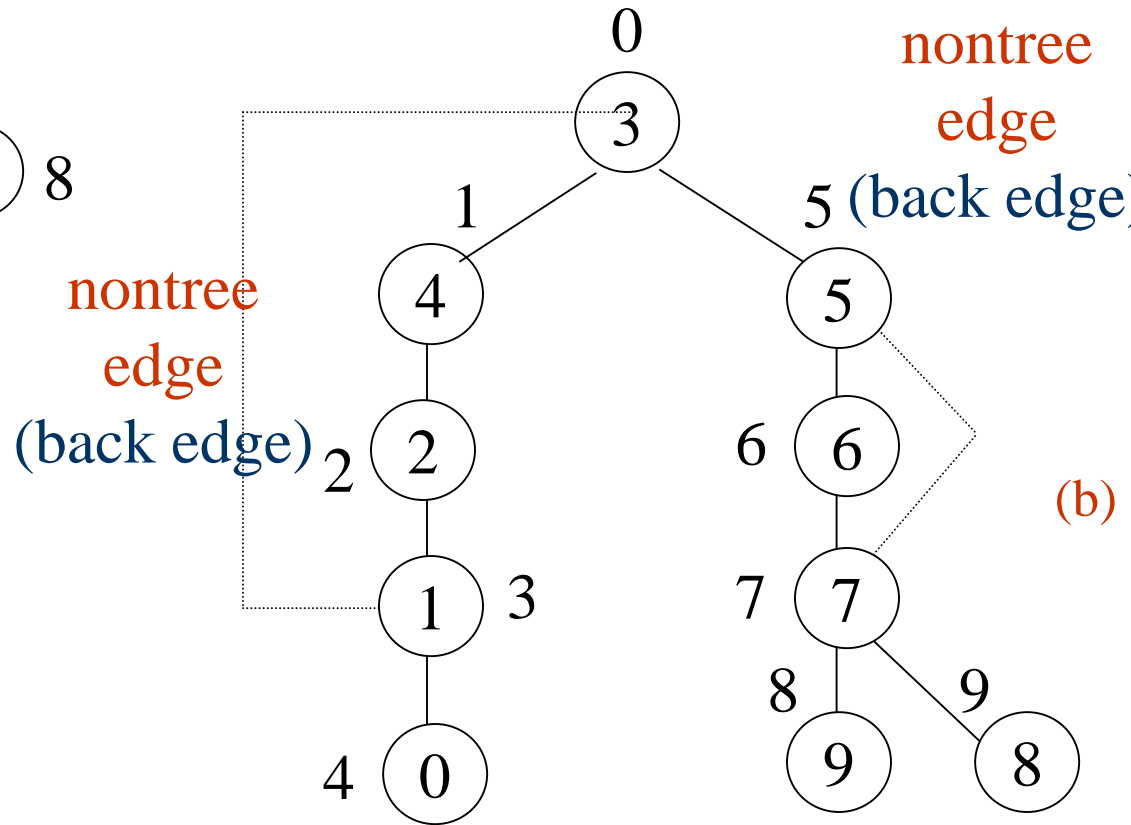
nontree edge (back edge)

nontree edge (back edge)

(b)

Any other vertex u is an articulation point iff it has at least one child w such that we cannot reach an ancestor of u using a path

If u is an ancestor of v then dfn(u) < dfn(v).

Vertax	0	1	2	3	4	5	6	7	8	9
dfn	4	3	2	0	1	5	6	7	9	8
low	4	0	0	0	0	5	5	5	9	8

low(u)=min{dfn(u), min{low(w)|w is a child of u},

min{dfn(w)|(u,w) is a back edge}

u: articulation point
low(child) ≥ dfn(u)
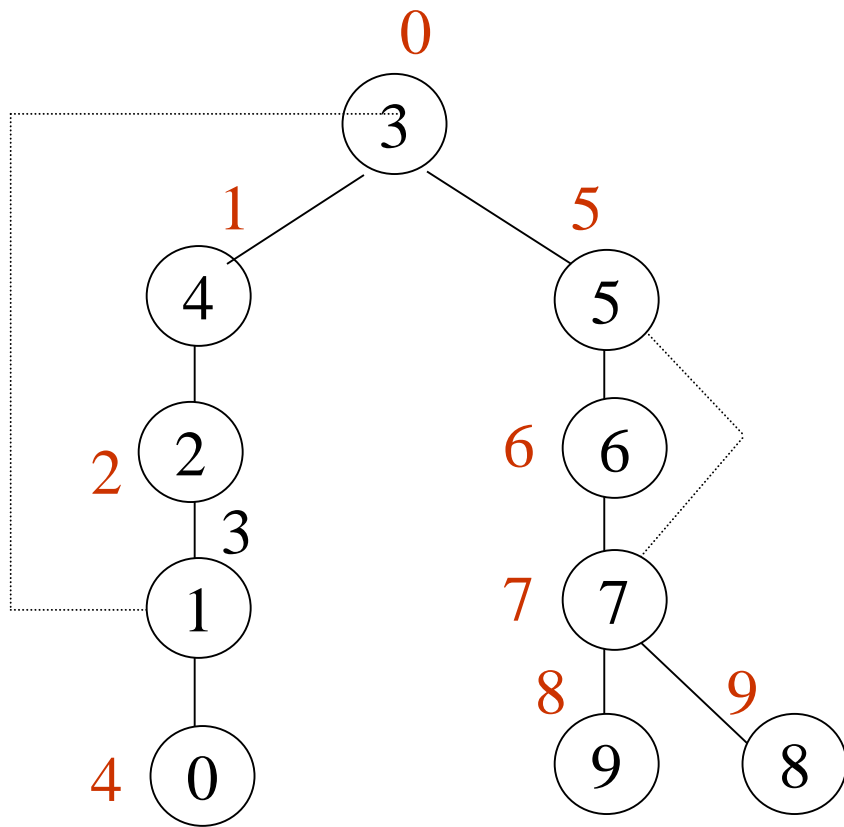
*The root of a depth first spanning tree is an <u>articulation point</u> iff it has **at least two children**.

*Any other vertex u is an articulation point iff it has at least one child w such that <u>we cannot reach an ancestor of u</u> using a path that consists of (1) only w; (2) descendants of w; (3) single back edge.

$$low(u) = \min\{dfn(u), \min\{low(w)|w \text{ is a child of } u\},$$
$$\min\{dfn(w)|(u,w) \text{ is a back edge}\}$$

u: articulation point ➔ low(child) ≥ dfn(u)

vertex	dfn	low	child	low (child)	low:dfn
0	4	4 (4,n,n)	null	null	null:4
1	3	0 (3,4,0)	0	4	4 ≥ 3 •
2	2	0 (2,0,n)	1	0	0 < 2
3	0	0 (0,0,n)	4,5	0,5	0,5 ≥ 0 •
4	1	0 (1,0,n)	2	0	0 < 1
5	5	5 (5,5,n)	6	5	5 ≥ 5 •
6	6	5 (6,5,n)	7	5	5 < 6
7	7	5 (7,8,5)	8,9	9,8	9,8 ≥ 7 •
8	9	9 (9,n,n)	null	null	null, 9
9	8	8 (8,n,n)	null	null	null, 8

$low(u) = \min\{dfn(u),$
$\quad \min\{low(w) | w \text{ is a child of } u\},$
$\quad \min\{dfn(w) | (u,w) \text{ is a back edge}\}$

u: articulation point ➔ low(child) ≥ dfn(u)

66

```
#define MIN2(x,y) ( (x)<(y) ? (x):(y) )

void init(void)
 {
  int i;
  for (i = 0; i < n; i++) {
       visited[i] = FALSE;
       dfn[i] = low[i] = -1;
       }
       num = 0;
 }
```

*Program 6.5: Initializaiton of *dfn* and *low*

***Program 6.4:** Determining *dfn* and *low*   num has been initialized to 0

```
void dfnlow(int u, int v)                 Initial call: dfn(x,-1)
{
/* compute dfn and low while performing a dfs search
    beginning at vertex u, v is the parent of u (if any) */
        node_pointer ptr;
        int w;
        dfn[u] = low[u] = num++;          low[u]=min{dfn(u), …}
        for (ptr = graph[u]; ptr; ptr = ptr ->link) {
            w = ptr ->vertex;
            if (dfn[w] < 0) { /*w is an unvisited vertex */
                dfnlow(w, u);
                low[u] = MIN2(low[u], low[w]);
            }       low[u]=min{…, min{low(w)|w is a child of u}, …}
            else if (w != v)   dfn[w]≠0 非第一次，表示藉back edge
                low[u] =MIN2(low[u], dfn[w] );
        }
}          low[u]=min{…,…,min{dfn(w)|(u,w) is a back edge}
```

v
↓
u
↓
w

v
↓↑
u

O X
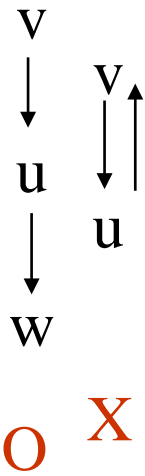
*Program 6.6: Biconnected components of a graph

```
void bicon(int u, int v)
{
/* compute dfn and low, and output the edges of G by their
    biconnected  components , v is the parent ( if any) of the u
   (if any)  in the resulting spanning tree. It is assumed that all
  entries of dfn[ ] have been initialized to -1, num has been
  initialized to 0,  and the stack has been set to empty */
    node_pointer ptr;
    int w, x, y;
    dfn[u] = low[u] = num ++;
    for (ptr = graph[u]; ptr; ptr = ptr->link) {
      w = ptr ->vertex;
      if ( v != w && dfn[w] < dfn[u] )
        push(u, w);        /* add edge to stack */
```

low[u]=min{dfn(u), …}

(1) dfn[w]=-1 第一次

(2) dfn[w]!=-1非第一次，藉back edge

```c
if(dfn[w] < 0) {/* w has not been visited */
    bicon(w, u); low[u]=min{…, min{low(w)|w is a child of u}, .
    low[u] = MIN2(low[u], low[w]);
    if (low[w] >= dfn[u] ){      articulation point
        printf("New biconnected component: ");
        do { /* delete edge from stack */
            pop(&x, &y);
            printf(" <%d, %d>" , x, y);
        } while (!(( x = = u) && (y = = w)));
        printf("\n");
    }
}
else if (w != v)  low[u] = MIN2(low[u], dfn[w]);
}   low[u]=min{…, …, min{dfn(w)|(u,w) is a back edge}}
}
```

Minimum Cost Spanning Tree

☐ The cost of a spanning tree of <u>a weighted undirected graph</u> is <u>the sum of the costs of the edges</u> in the spanning tree

☐ A minimum cost spanning tree is a spanning tree of least cost

☐ Three different algorithms can be used
  – Kruskal
  – Prim
  – Sollin

Select *n-1* edges from a weighted graph of *n* vertices with minimum cost.

Greedy Strategy

☐ An optimal solution is constructed in stages

☐ At each stage, the best decision is made at this time

☐ Since this decision cannot be changed later, we make sure that the decision will result in a feasible solution

☐ Typically, the selection of an item at each stage is based on a least cost or a highest profit criterion

Kruskal's Idea

- Build a minimum cost spanning tree T by adding edges to T one at a time

- Select the edges for inclusion in T in nondecreasing order of the cost

- An edge is added to T if it does not form a cycle

- Since G is connected and has n > 0 vertices, exactly n-1 edges will be selected

1    $0 \xrightarrow{10} 5$

2    $2 \xrightarrow{12} 3$

3    $1 \xrightarrow{14} 6$

4    $1 \xrightarrow{16} 2$

5    $3 \xrightarrow{18} 6$

6    $3 \xrightarrow{22} 4$

7    $4 \xrightarrow{24} 6$

8    $4 \xrightarrow{25} 5$

9    $0 \xrightarrow{28} 1$

(a)

(b)

(c)

74

(d)   (e)   (f)

75
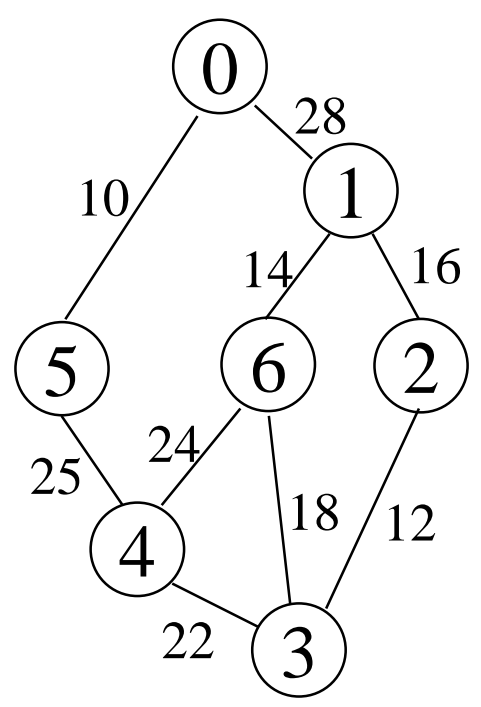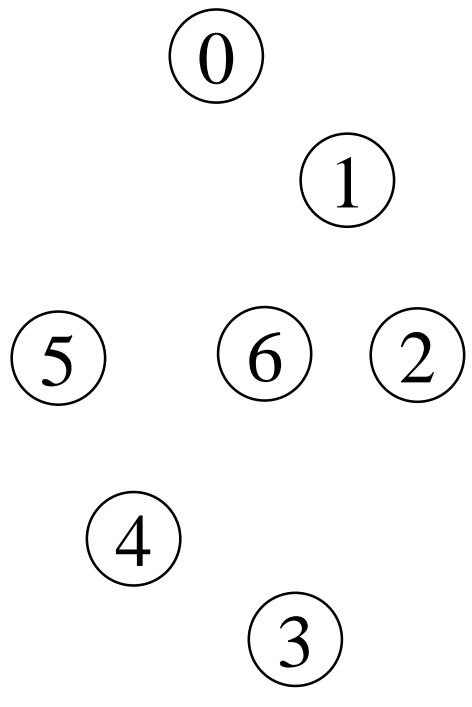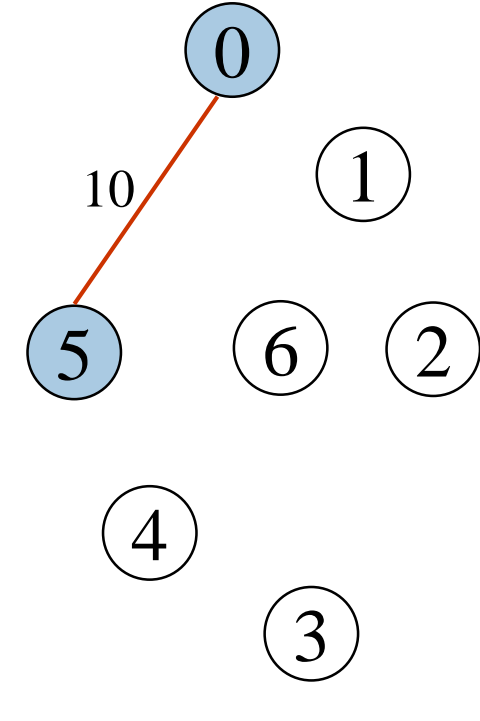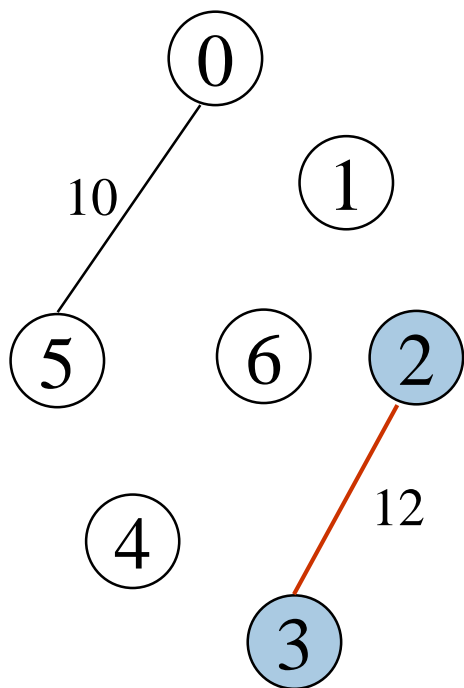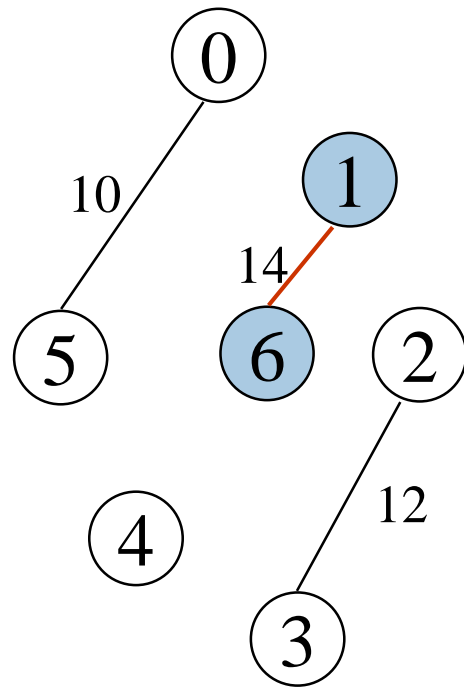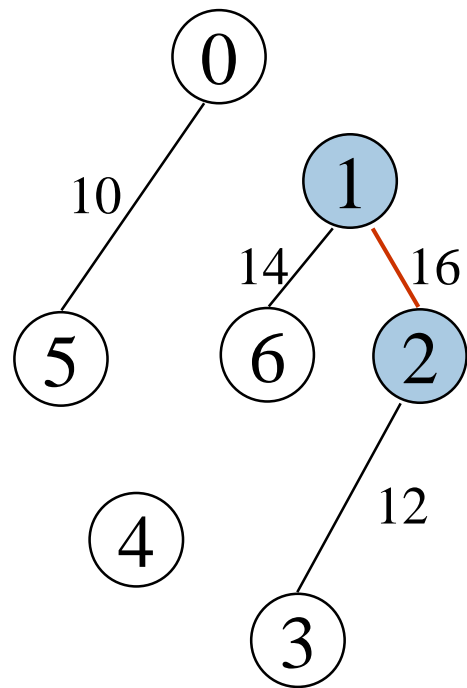
1    $0 \xrightarrow{10} 5$

2    $2 \xrightarrow{12} 3$

3    $1 \xrightarrow{14} 6$

4    $1 \xrightarrow{16} 2$

5    $3 \xrightarrow{18} 6$

6    $3 \xrightarrow{22} 4$

7    $4 \xrightarrow{24} 6$

8    $4 \xrightarrow{25} 5$

9    $0 \xrightarrow{28} 1$

cost = 10 +25+22+12+16+14

(g)

+   **4** —— **6**

cycle

(h)

76

Kruskal's Algorithm

目標：取出n-1條edges

```
T= {};
while (T contains less than n-1 edges
      && E is not empty) {
choose a least cost edge (v,w) from E;
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)
  printf("No spanning tree\n");
```

min heap construction time O(e)

choose and delete O(log e)

find find & union O(log e)

O(e log e)

Kruskal's Algorithm

☐ 取得graph的Adjacency Matrix來實作 Kruskal's Algorithm

☐ 實作步驟

1. 取得圖的關係
2. 每次找出最低cost的邊且不會使圖產生 cycle
3. 直到產生(頂點數-1)個邊的最小花費生 成樹

Kruskal's Algorithm

☐ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4

Kruskal's Algorithm

☐ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4

Kruskal's Algorithm

☐ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4

Kruskal's Algorithm

☐ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6
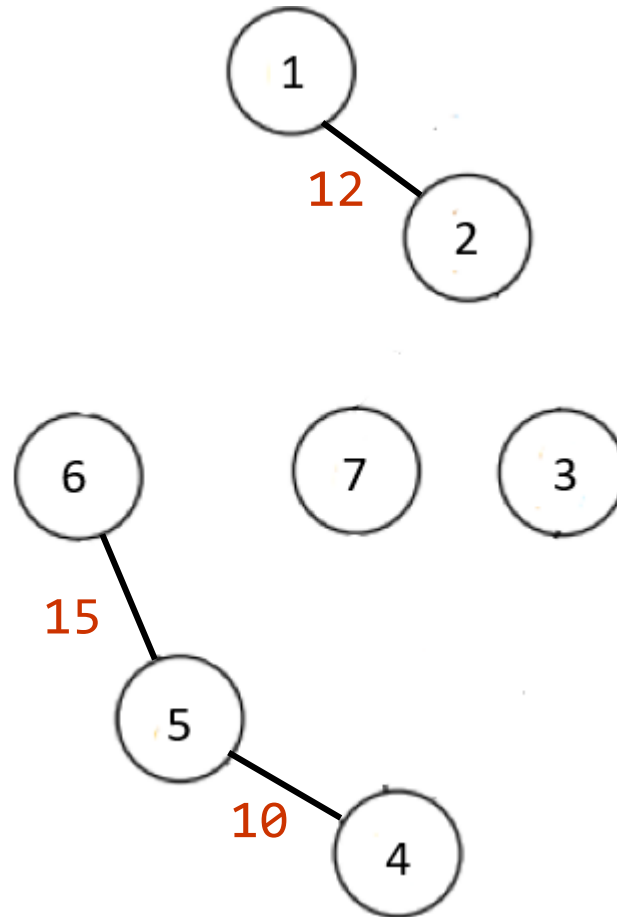
4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4

Kruskal's Algorithm

☐ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4

Kruskal's Algorithm

☐ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4

Kruskal's Algorithm

□ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4

Kruskal's Algorithm

☐ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4

產生(頂點數-1)個邊

Prim's Algorithm

(tree all the time vs. forest)

```
T={};
TV={0};
while (T contains fewer than n-1 edges)
{
  let (u,v) be a least cost edge such
    that u ∈ TV and v ∉ TV
  if (there is no such edge ) break;
  add v to TV;
  add (u,v) to T;
}
if (T contains fewer than n-1 edges)
  printf("No spanning tree\n");
```

Examples for Prim's Algorithm

(a)   (b)   (c)
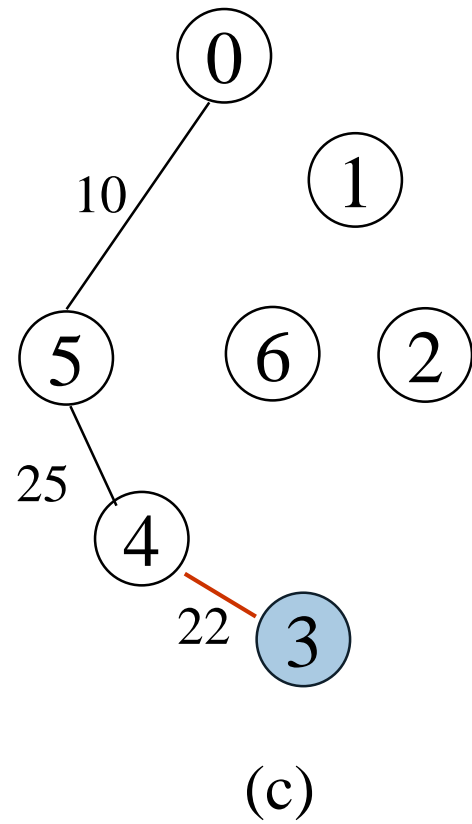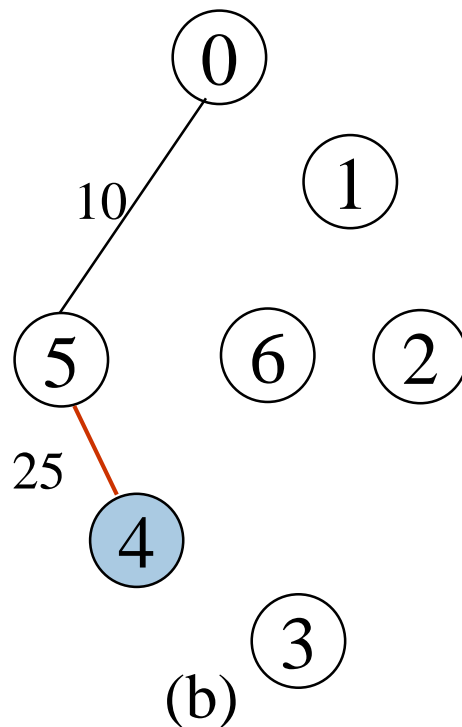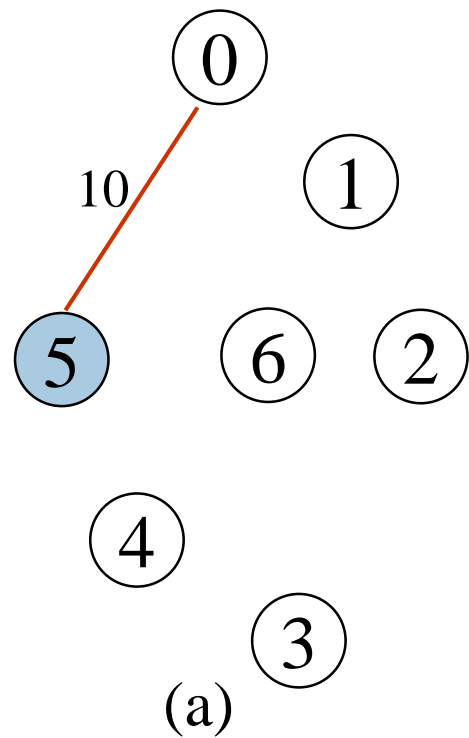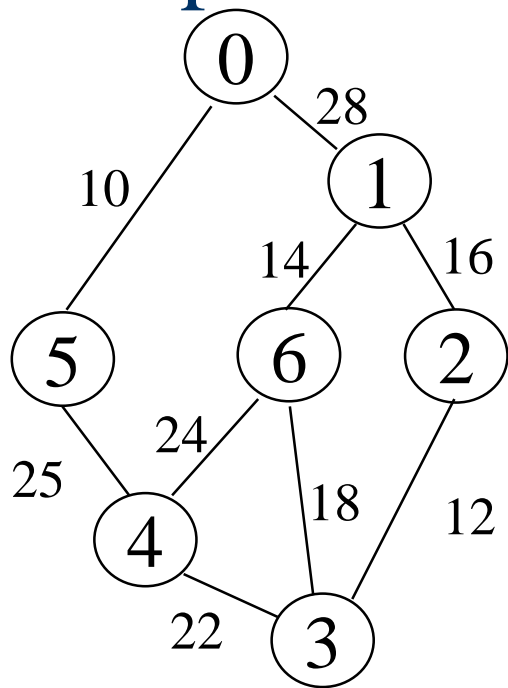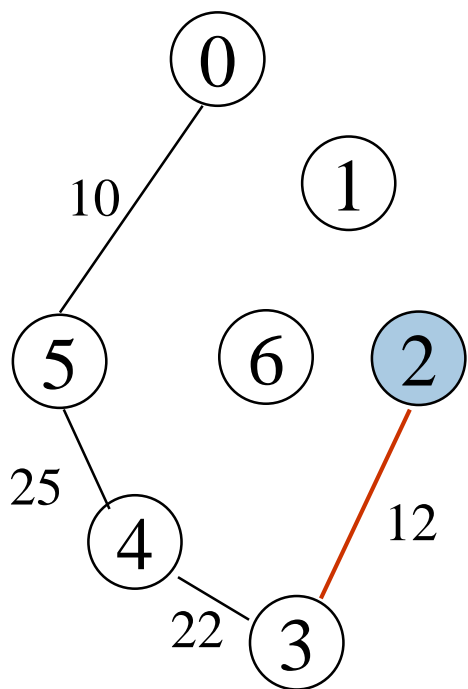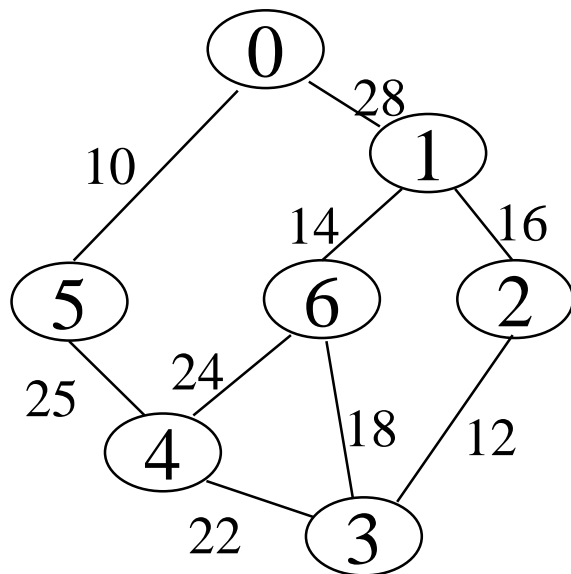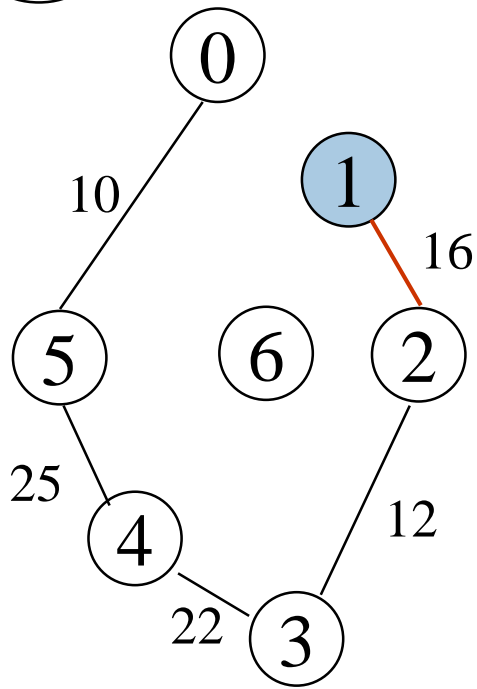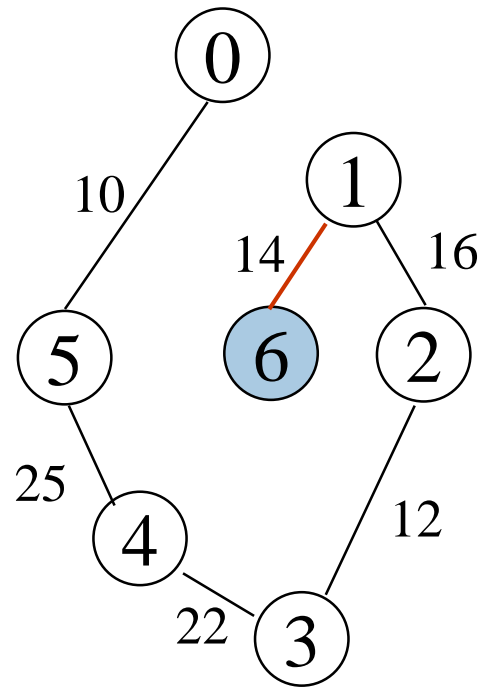
(d)   (e)   (f)

Prim's Algorithm

☐ 取得graph的Adjacency Matrix來實作 Prim's Algorithm

☐ 實作步驟

1. 取得圖的關係

2. 從頂點1開始選cost最小的邊來建立最小花費生成樹且不會使圖產生cycle

Prim's Algorithm

□ 圖解步驟
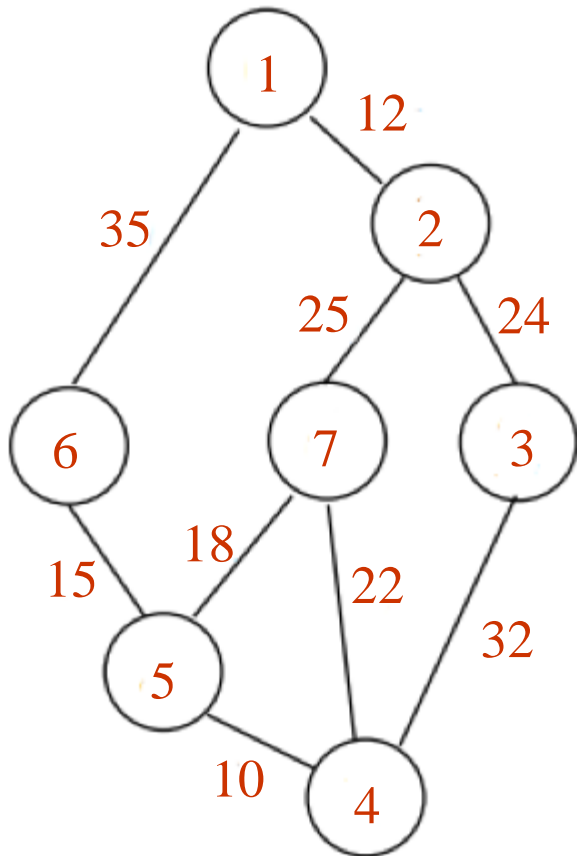
1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4

Prim's Algorithm

□ 圖解步驟

1 --12--> 2 選cost小的

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6
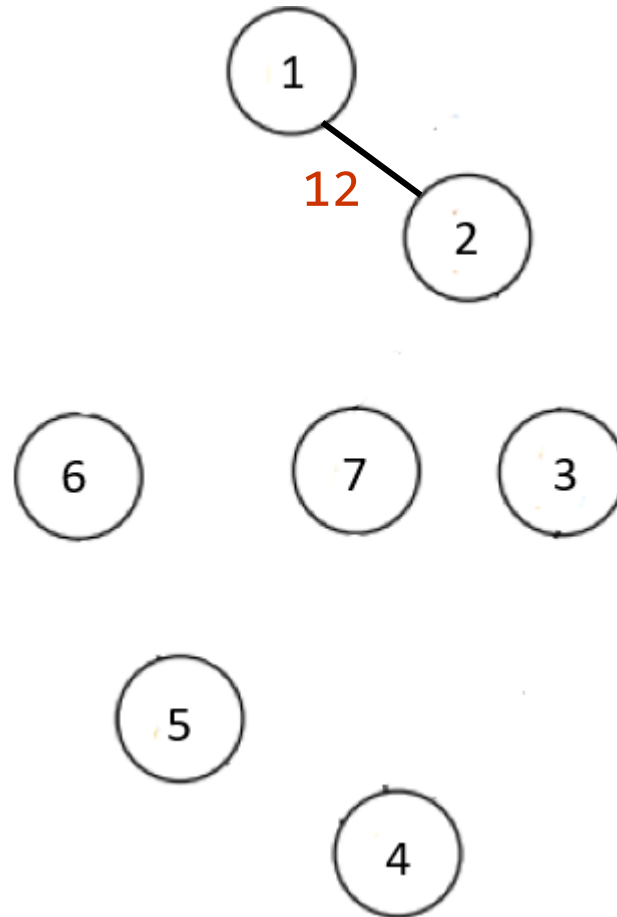
4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4

Prim's Algorithm

圖解步驟

1 --12--> 2

2 --24--> 3 選cost小的

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5
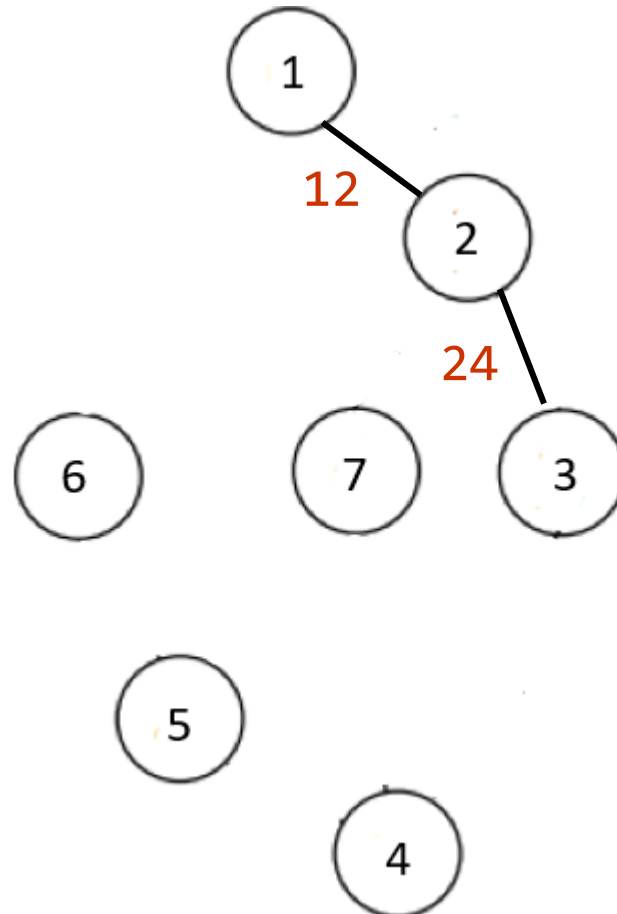
5 --18--> 7

4 --22--> 7

3 --32--> 4

Prim's Algorithm

□ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7 選cost小的

5 --15--> 6

4 --10--> 5

5 --18--> 7

4 --22--> 7
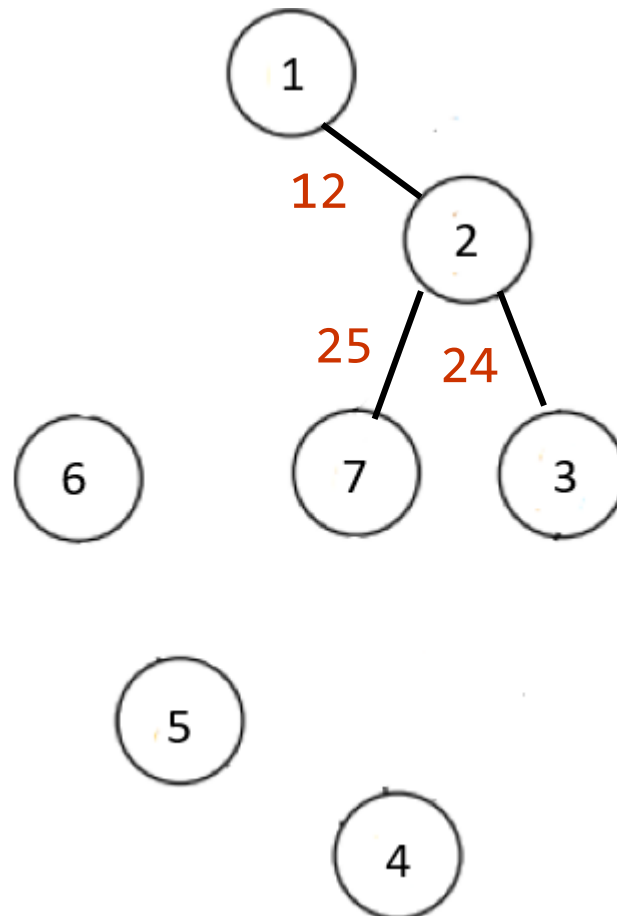
3 --32--> 4

Prim's Algorithm

□ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5

5 --18--> 7 選cost小的

4 --22--> 7

3 --32--> 4
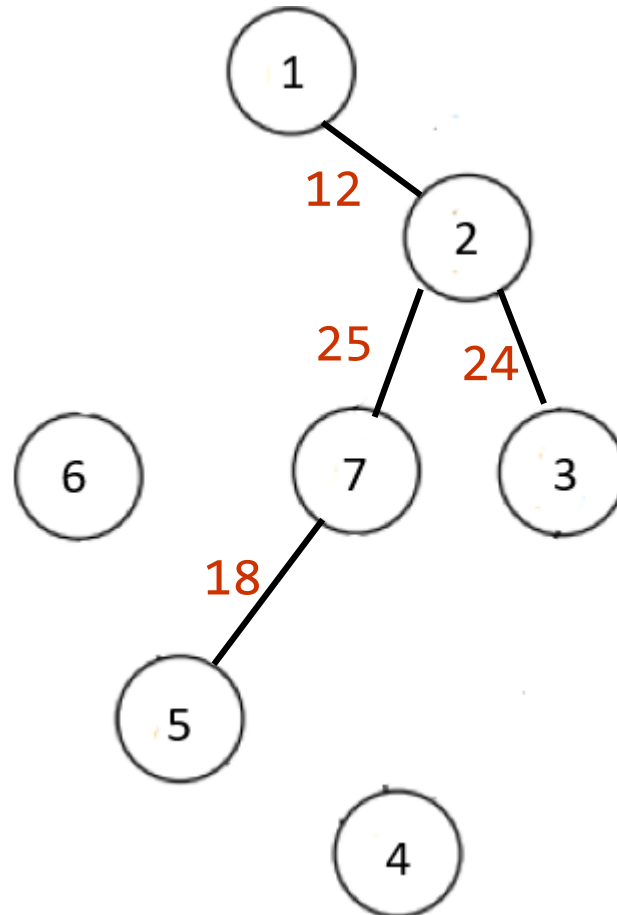
Prim's Algorithm

□ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5   選cost小的

5 --18--> 7

4 --22--> 7

3 --32--> 4
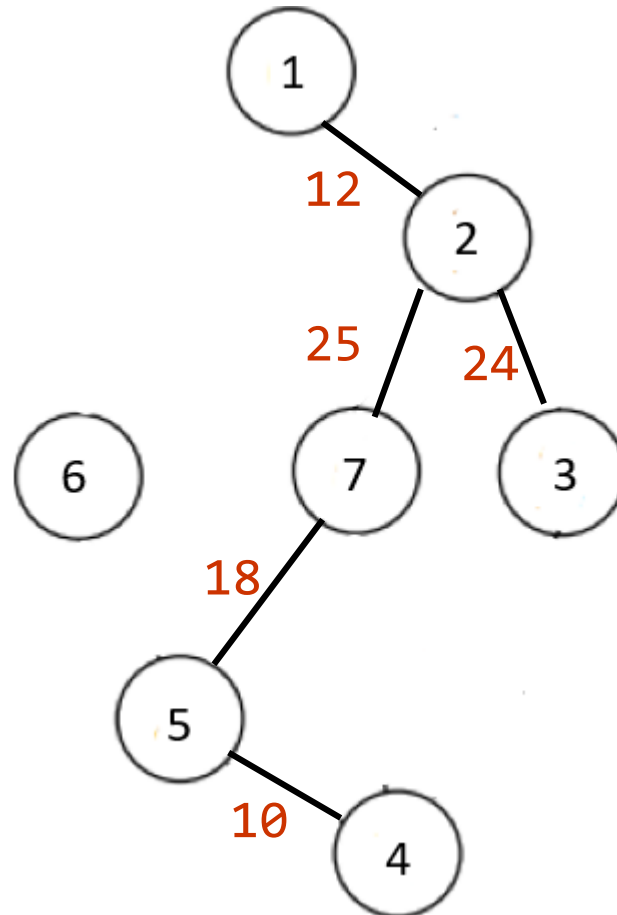
Prim's Algorithm

☐ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

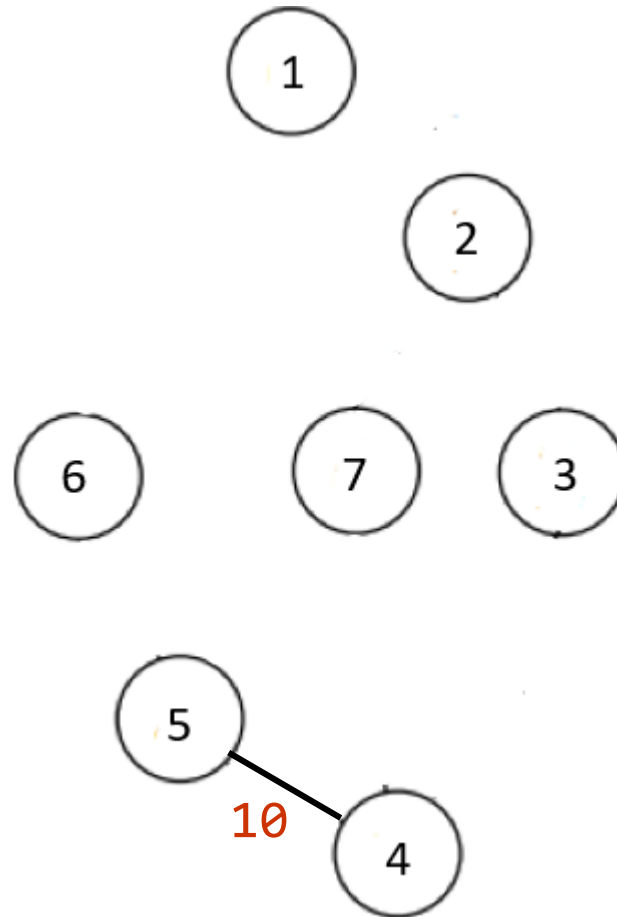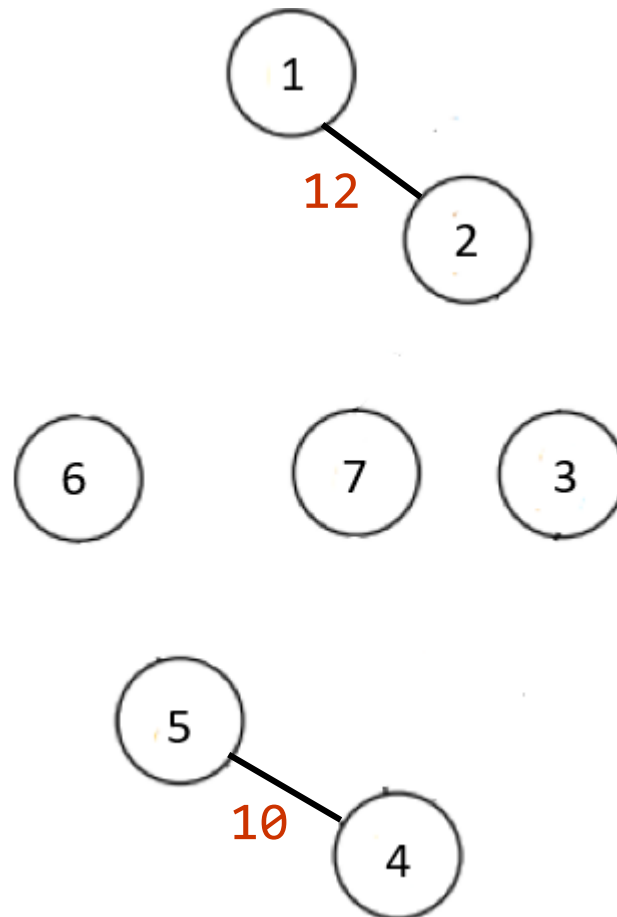2 --25--> 7

5 --15--> 6　選cost小的

4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4
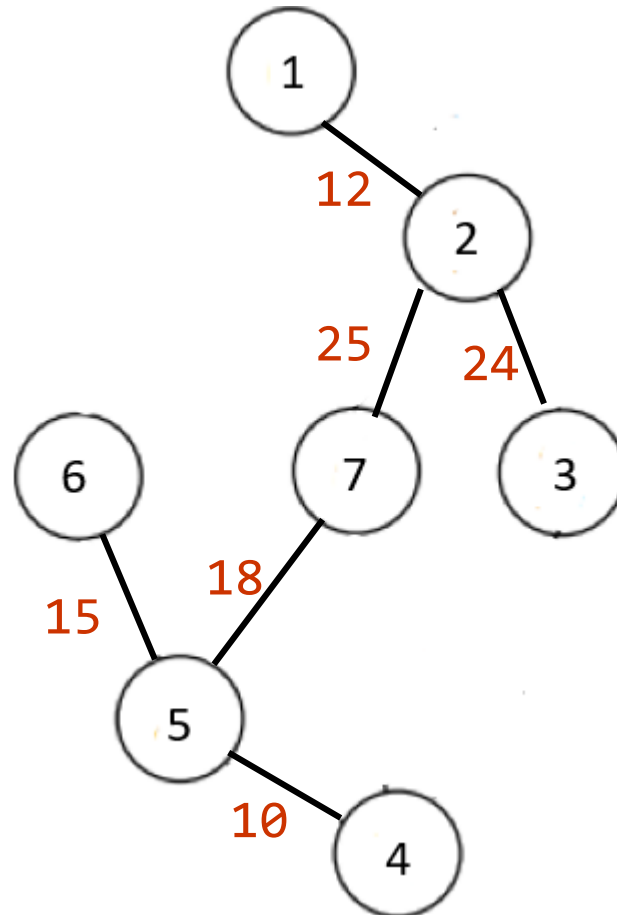
Prim's Algorithm

☐ 圖解步驟

1 --12--> 2

2 --24--> 3

1 --35--> 6

2 --25--> 7

5 --15--> 6

4 --10--> 5

5 --18--> 7

4 --22--> 7

3 --32--> 4
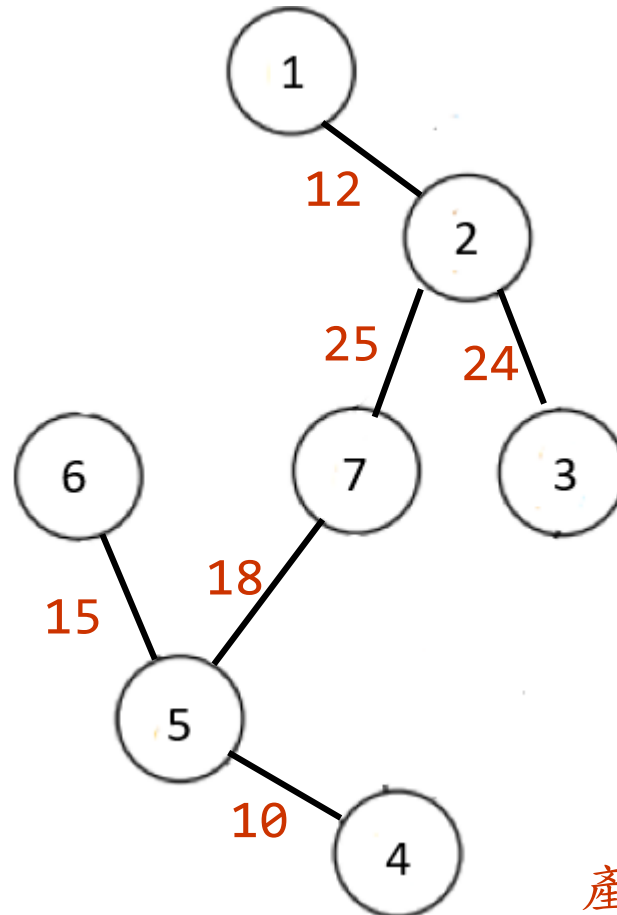
產生(頂點數-1)個邊

Sollin's Algorithm
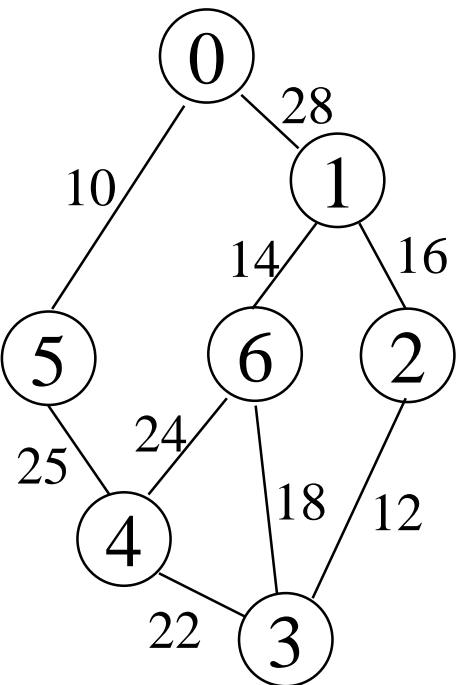
vertex	edge
0	0 -- 10 --> 5, 0 -- 28 --> 1
1	1 -- 14 --> 6, 1-- 16 --> 2, 1 -- 28 --> 0
2	2 -- 12 --> 3, 2 -- 16 --> 1
3	3 -- 12 --> 2, 3 -- 18 --> 6, 3 -- 22 --> 4
4	4 -- 22 --> 3, 4 -- 24 --> 6, 4 -- 25 --> 5
5	5 -- 10 --> 0, 5 -- 25 --> 4
6	6 -- 14 --> 1, 6 -- 18 --> 3, 6 -- 24 --> 4

(a)

{0,5}

$5 \overset{25}{—} 4$ $0 \overset{28}{—} 1$

{1,6}

$1 \overset{16}{—} 2$ $1 \overset{28}{—} 0$

$6 \overset{18}{—} 3$ $6 \overset{24}{—} 4$

(b)

(c)
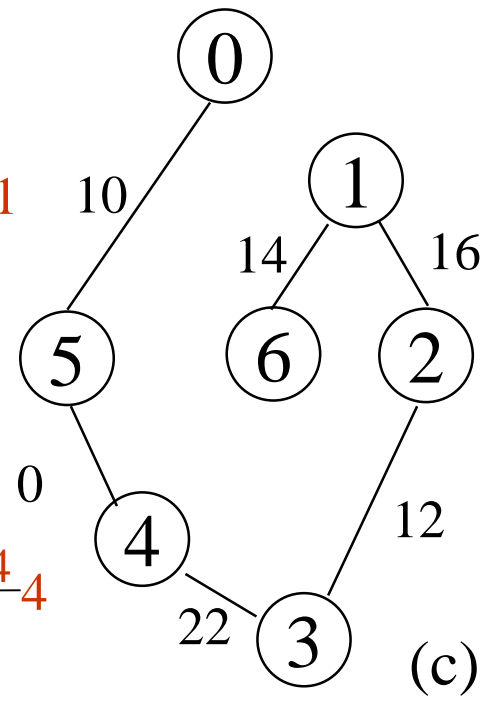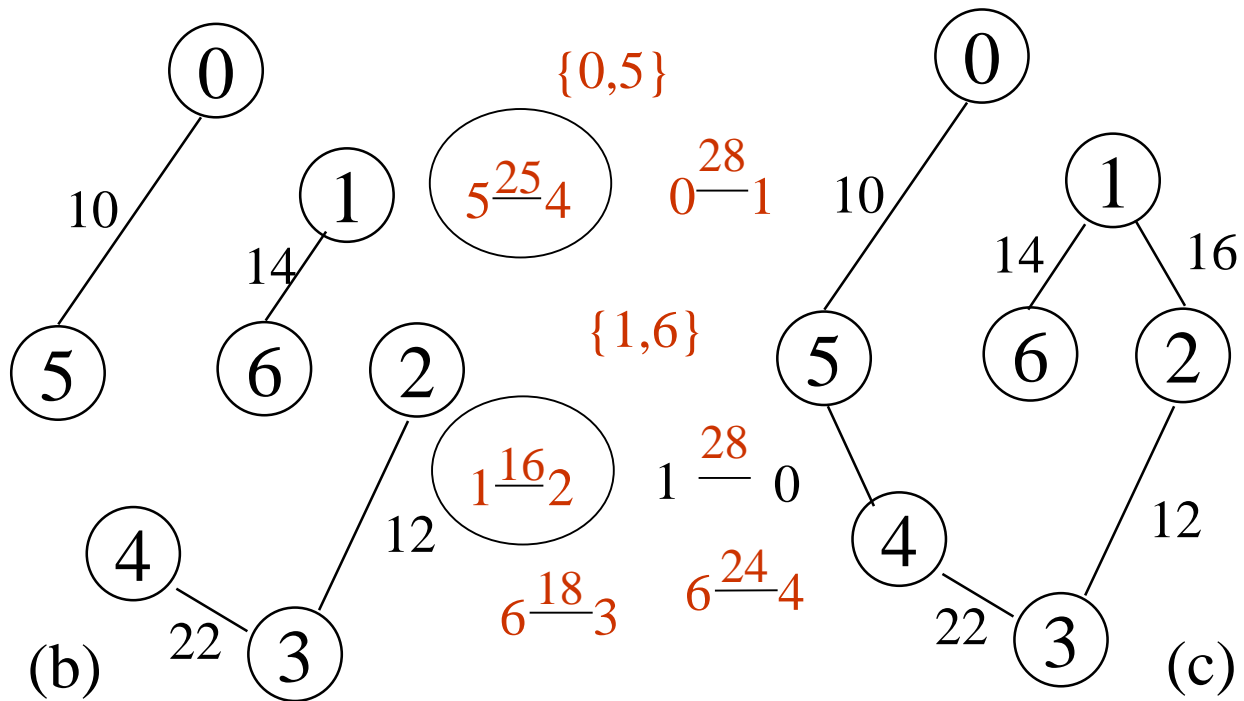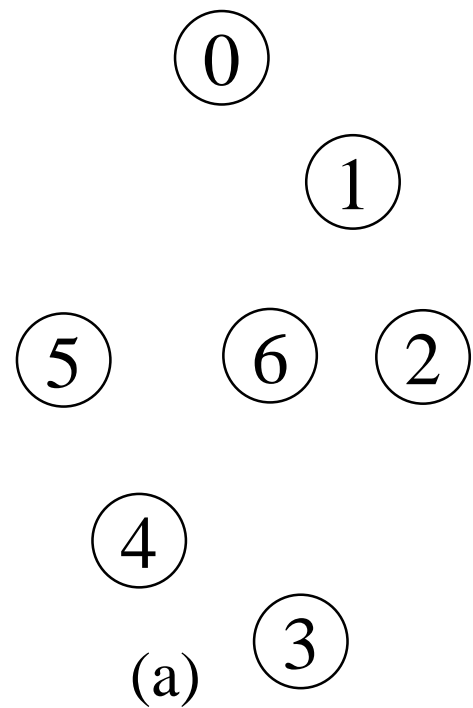
Single Source to All Destinations

Determine the shortest paths from v_0 to all the remaining vertices.

Dijkstra's algorithm

(a) 圖
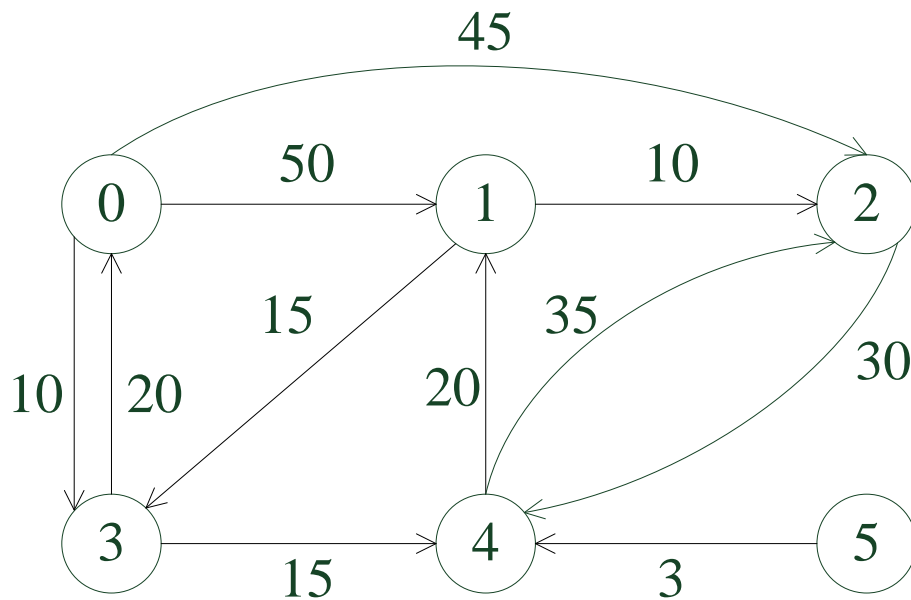
路徑	長度
1) 0, 3	10
2) 0, 3, 4	25
3) 0, 3, 4, 1	45
4) 0, 2	45

(b) 從 0 出發的最短路徑

Figure 6.26: Graph and shortest paths from v_0

Example

Cost adjacency matrix

(a)

1500

4

3

250

5

選5

(b)

1500

4

3

250

1000

5

4到3由1500改成1250

(c)

4

250

5

900

6

4到6由∞改成1150

(d)

4

250

5

7

1400

4到7由∞改成1650

(e)

3

4

1000

250

7

5

1400

900

6

選6

(f)

4

250

7

5

1400

1000

900
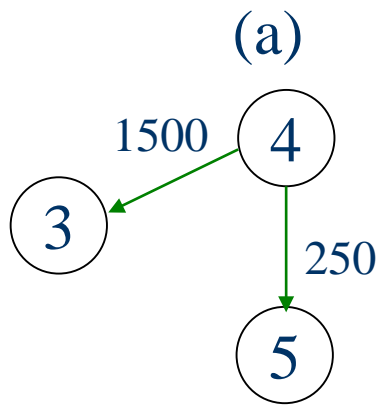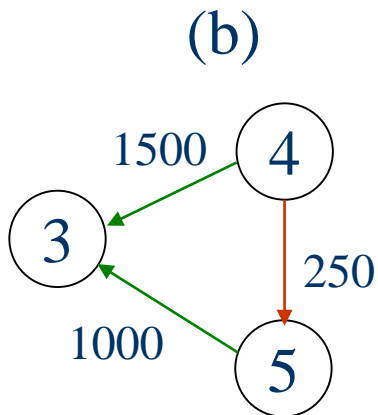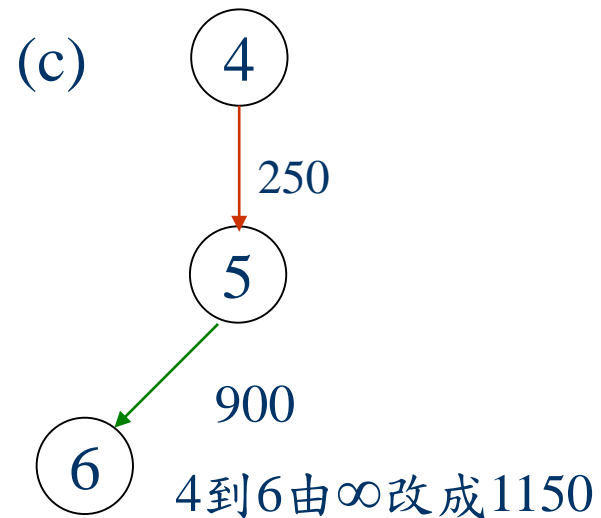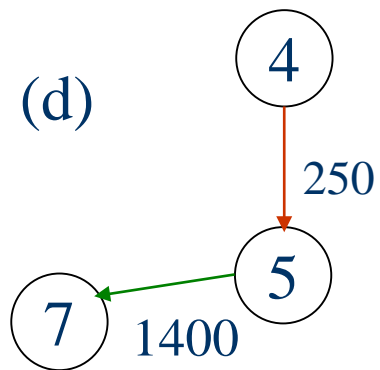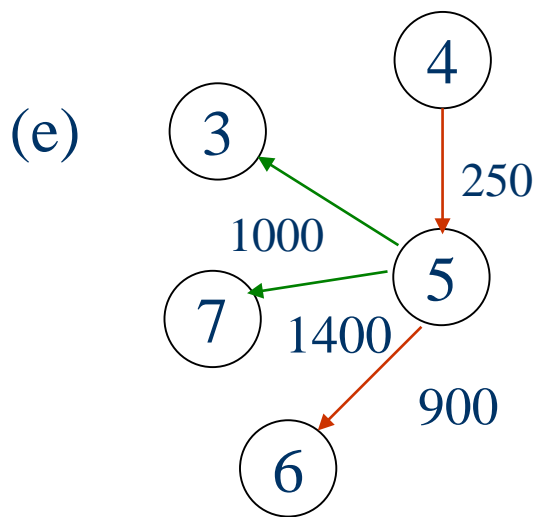
6
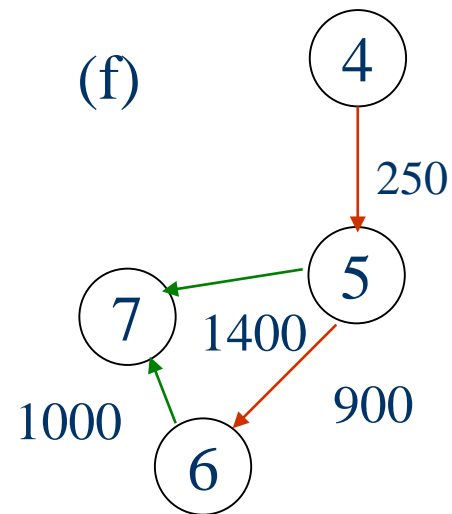
4-5-6-7比4-5-7長

(g) 選3

(h) 4到2由∞改成2450

(i) 選7

(j) 4到0由∞改成3350

103
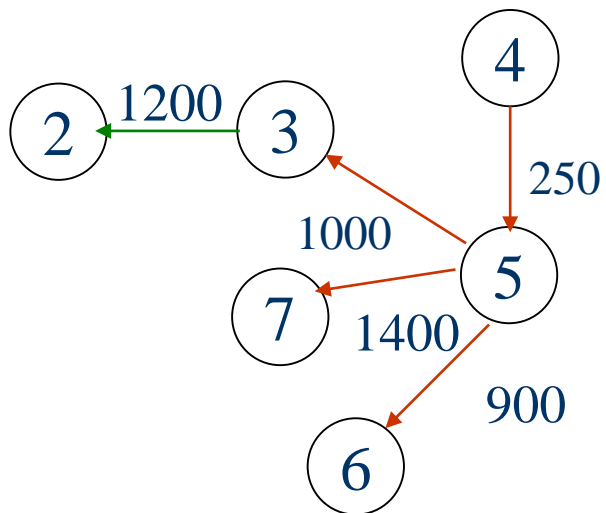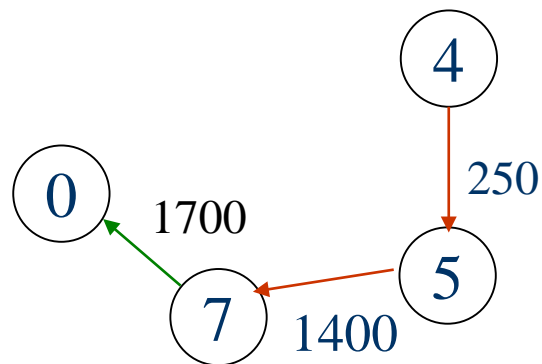
Example for the Shortest Path
(*Continued*)

Iteration	S	Vertex Selected	LA [0]	SF [1]	DEN [2]	CHI [3]	BO [4]	NY [5]	MIA [6]	NO
Initial	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{4}	(a) 5	+∞	+∞	+∞	1250	0	250	1150	1650
2	{4,5}	(e) 6	+∞	+∞	+∞	1250	0	250	1150	1650
3	{4,5,6}	(g) 3	+∞	+∞	2450	1250	0	250	1150	1650
4	{4,5,6,3}	(i) 7	3350	+∞	2450	1250	0	250	1150	1650
5	{4,5,6,3,7}	2	3350	3250	2450	1250	0	250	1150	1650
6	{4,5,6,3,7,2}	1	3350	3250	2450	1250	0	250	1150	1650
7	{4,5,6,3,7,2,1}									

(b) (c) (d) (f) (h) (j)

Dijkstra's algorithm

☐ 取得graph的Adjacency Matrix來實作 Dijkstra's algorithm

☐ 實作步驟

1. 取得圖的關係
2. 算出從起點到其他頂點的最短距離

Dijkstra's algorithm

□ 圖解步驟
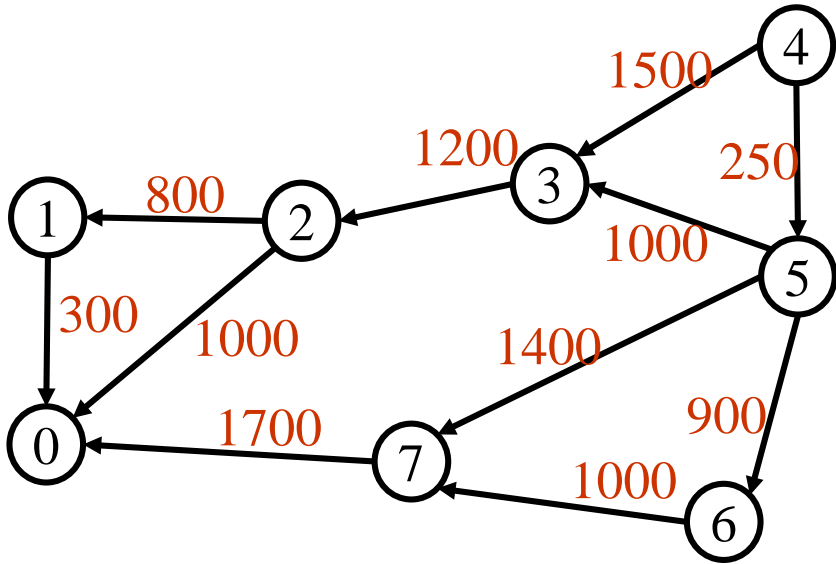
步驟	0	1	2	3	4	5	6	7
init	∞	∞	∞	∞	0	∞	∞	∞

Dijkstra's algorithm

□ 步驟1
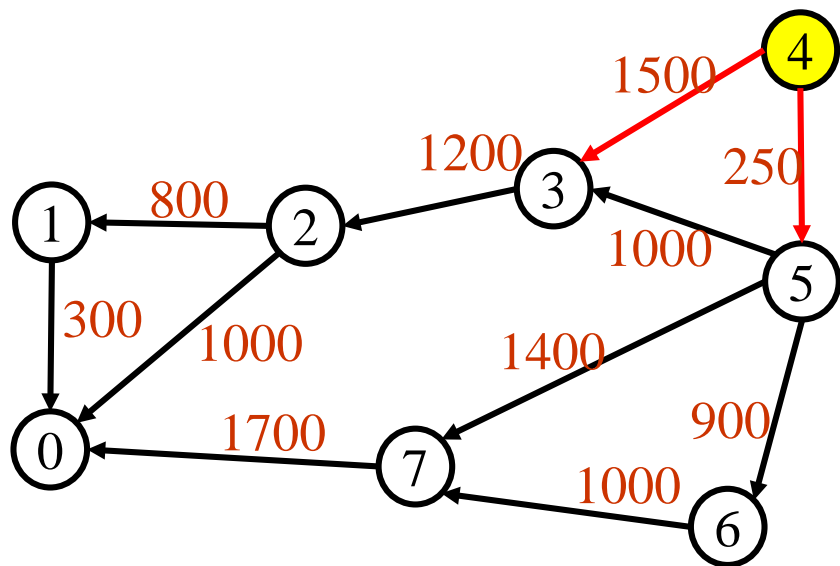
步驟	0	1	2	3	4	5	6	7
init	∞	∞	∞	∞	0	∞	∞	∞
1	∞	∞	∞	1500	0	250	∞	∞

從頂點4開始，與其相連的有頂點3和頂點5
頂點4->頂點5，距離250（新增）
頂點4->頂點3，距離1500（新增）

Dijkstra's algorithm

□ 步驟2
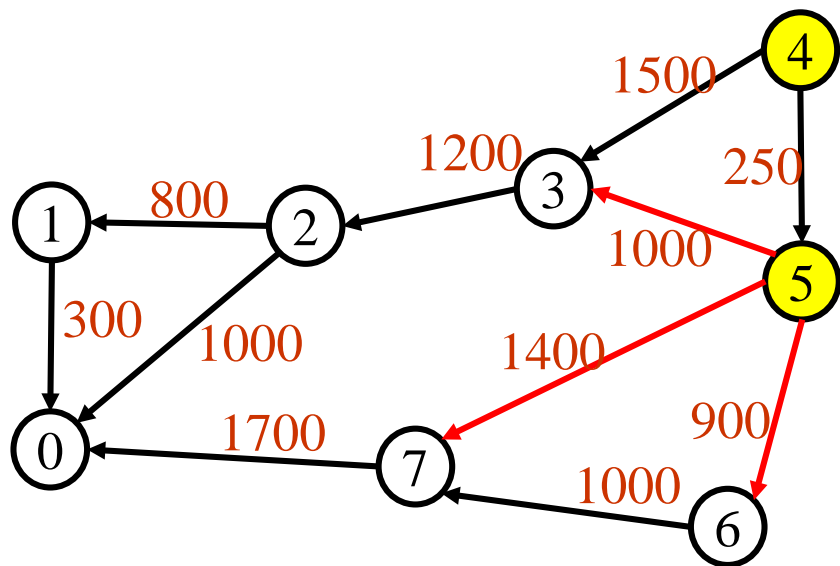
步驟	0	1	2	3	4	5	6	7
init	∞	∞	∞	∞	0	∞	∞	∞
1	∞	∞	∞	1500	0	250	∞	∞
2	∞	∞	∞	1250	0	250	1150	1650

加入頂點5，與其相連的有頂點3, 6, 7
頂點5->頂點3，距離250+1000=1250（更新）
頂點5->頂點6，距離250+900=1150（新增）
頂點5->頂點7，距離250+1400=1650（新增）

Dijkstra's algorithm

□ 步驟3
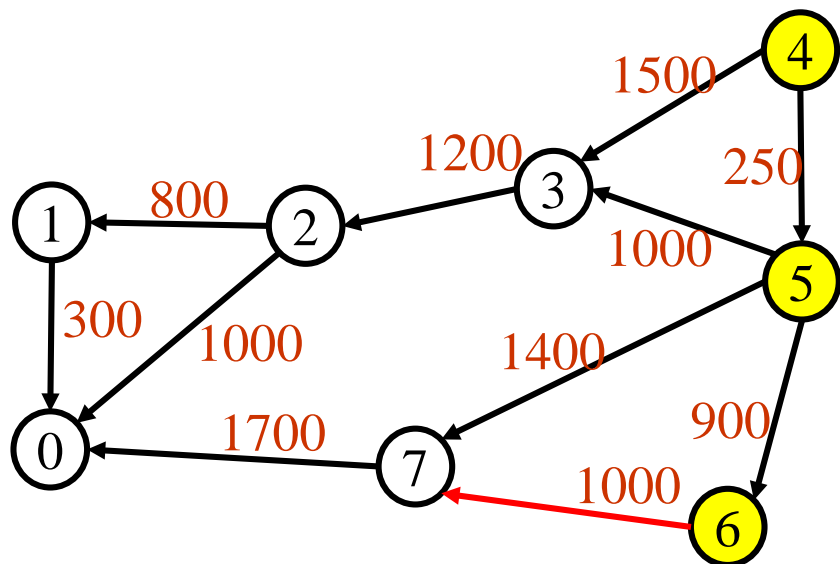
步驟	0	1	2	3	4	5	6	7
init	∞	∞	∞	∞	0	∞	∞	∞
1	∞	∞	∞	1500	0	250	∞	∞
2	∞	∞	∞	1250	0	250	1150	1650
3	∞	∞	∞	1250	0	250	1150	1650

加入頂點6，與其相連的有頂點7
頂點6->頂點7，距離1150+1000=2150（不更新）

Dijkstra's algorithm

□ 步驟4
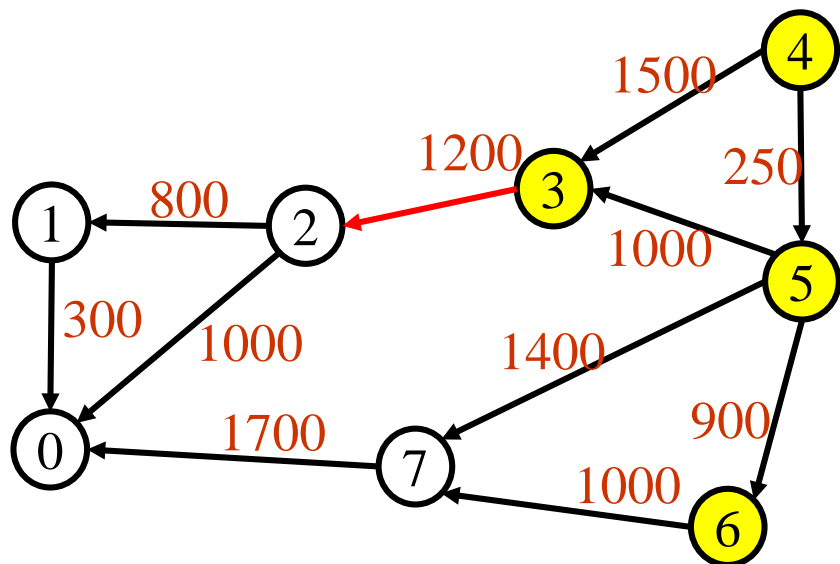
步驟	0	1	2	3	4	5	6	7
init	∞	∞	∞	∞	0	∞	∞	∞
1	∞	∞	∞	1500	0	250	∞	∞
2	∞	∞	∞	1250	0	250	1150	1650
3	∞	∞	∞	1250	0	250	1150	1650
4	∞	∞	2450	1250	0	250	1150	1650

加入頂點3，與其相連的有頂點2
頂點3->頂點2，距離1250+1200=2450（新增）

Dijkstra's algorithm

□ 步驟5

步驟	0	1	2	3	4	5	6	7
init	∞	∞	∞	∞	0	∞	∞	∞
1	∞	∞	∞	1500	0	250	∞	∞
2	∞	∞	∞	1250	0	250	1150	1650
3	∞	∞	∞	1250	0	250	1150	1650
4	∞	∞	2450	1250	0	250	1150	1650
5	3350	∞	2450	1250	0	250	1150	1650

加入頂點7，與其相連的有頂點0
頂點7->頂點0，距離1650+1700=3350（新增）

Dijkstra's algorithm

□ 步驟6

步驟	0	1	2	3	4	5	6	7
init	∞	∞	∞	∞	0	∞	∞	∞
1	∞	∞	∞	1500	0	250	∞	∞
2	∞	∞	∞	1250	0	250	1150	1650
3	∞	∞	∞	1250	0	250	1150	1650
4	∞	∞	2450	1250	0	250	1150	1650
5	3350	∞	2450	1250	0	250	1150	1650
6	3350	3250	2450	1250	0	250	1150	1650

加入頂點2，與其相連的有頂點1，0
頂點2->頂點0，距離2450+1000=3450（不更新）
頂點2->頂點1，距離2450+800=3250（新增）

Dijkstra's algorithm

□ 步驟7

步驟	0	1	2	3	4	5	6	7
init	∞	∞	∞	∞	0	∞	∞	∞
1	∞	∞	∞	1500	0	250	∞	∞
2	∞	∞	∞	1250	0	250	1150	1650
3	∞	∞	∞	1250	0	250	1150	1650
4	∞	∞	2450	1250	0	250	1150	1650
5	3350	∞	2450	1250	0	250	1150	1650
6	3350	3250	2450	1250	0	250	1150	1650
7	3350	3250	2450	1250	0	250	1150	1650

加入頂點1，與其相連的有頂點0
頂點1->頂點0，距離3250+300=3550（不更新）

Dijkstra's algorithm

☐ 步驟8

步驟	0	1	2	3	4	5	6	7
init	∞	∞	∞	∞	0	∞	∞	∞
1	∞	∞	∞	1500	0	250	∞	∞
2	∞	∞	∞	1250	0	250	1150	1650
3	∞	∞	∞	1250	0	250	1150	1650
4	∞	∞	2450	1250	0	250	1150	1650
5	3350	∞	2450	1250	0	250	1150	1650
6	3350	3250	2450	1250	0	250	1150	1650
7	3350	3250	2450	1250	0	250	1150	1650

加入頂點0

Single Source to All Destinations

```
void shortestpath(int v, int
   cost[][MAX_ERXTICES], int distance[], int n,
   short int found[])
{
   int i, u, w;
   for (i=0; i<n; i++) {
      found[i] = FALSE;
      distance[i] = cost[v][i];         O(n)
   }
   found[v] = TRUE;
   distance[v] = 0;
```

```
for (i=0; i<n-2; i++) {determine n-1 paths from v
  u = choose(distance, n, found);
  // return u which is minimum distance to v
  found[u] = TRUE;
  for (w=0; w<n; w++)
    if (!found[w])   與u相連的端點w
      if (distance[u]+cost[u][w]<distance[w])
        distance[w] = distance[u]+cost[u][w];
}
}                     O(n^2)
```

```c
int choose(int distance[], int n, short int found[])
{
  /* 找出還沒確認最短距離的點 */
  int i, min, minpos;
  min = INT_MAX;
  minpos = -1;
  for (i = 0; i < n; i++) {
    if(distance[i] < min && !found[i])
    {
      min = distance[i];
      minpos = i;
    }
  }
  return minpos;
}
```

Shortest paths with **negative edge** lengths
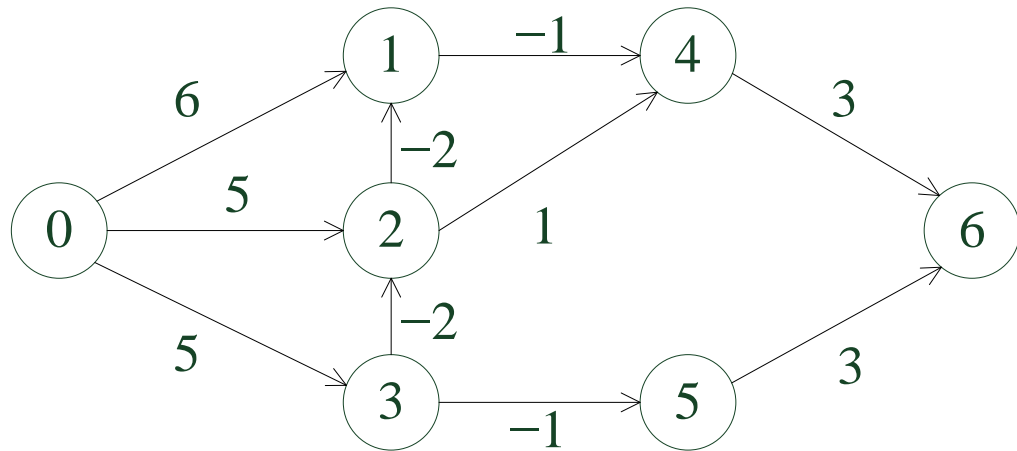
(a) 有向圖

k	$dist^k[7]$						
	0	1	2	3	4	5	6
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Graph with Negative Cycle
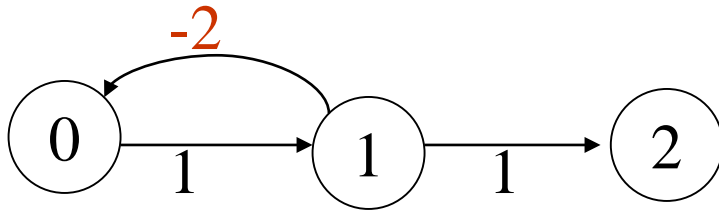
(a) Directed graph

$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

(b) A^{-1}

0, 1, 0, 1, 0, 1, …, 0, 1, 2

The length of the shortest path from vertex 0 to vertex 2 is $-\infty$.

Bellman and Ford algorithm to compute shortest paths

```
void BellmanFord(int n, int v)
{
  /* 計算單一起點/所有終點的最短路徑，其中邊長允許是負值 */
  for (int i = 0; i < n; i++)
      dist[i] = length[v][i];
         /* 對dist做初始化 */

  for (int k = 2; k <= n-1; k++)
    for (每個u滿足u!=v 且u至少有一個進到它的邊)
      for(每個圖上的邊<i,u>)
        if(dist[u] > dist[i] + length[i][u])
          dist[u] = dist[i] + length[i][u];
}
```

Bellman and Ford algorithm

☐ 取得graph的關係後來實作Bellman and Ford algorithm

☐ 實作步驟

1. 取得圖的關係

Number of vertices in graph: 7
Number of edges in graph: 10
Source vertex number: 0
Source, destination, weight of edg
     0 1 6
     0 2 5
     0 3 5
     3 2 -2
     2 1 -2
     1 4 -1
     2 4 1
     3 5 -1
     4 6 3
     5 6 3
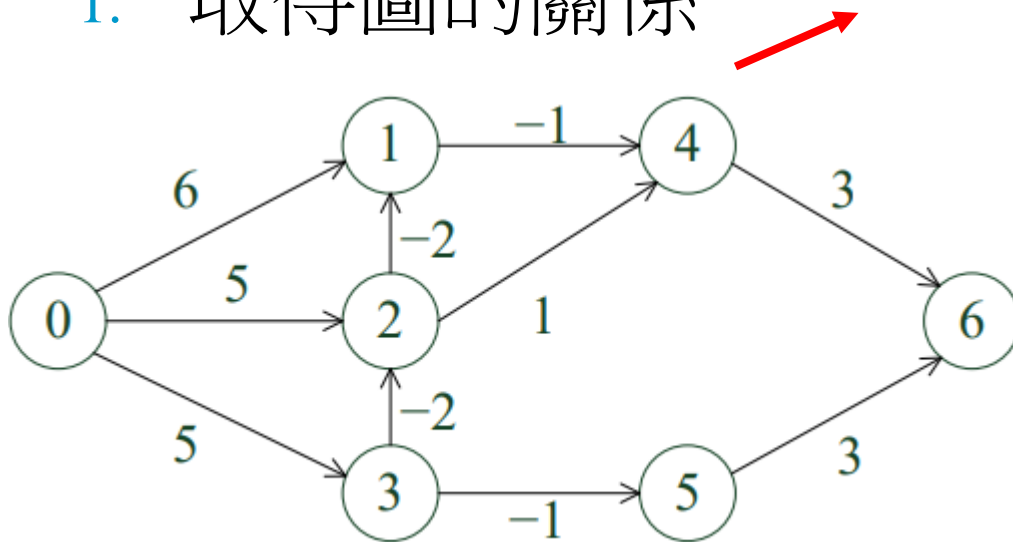
Bellman and Ford algorithm

init

Source =u	Destination =v	weight
0	1	6
0	2	5
0	3	5
3	2	-2
2	1	-2
1	4	-1
2	4	1
3	5	-1
4	6	3
5	6	3

StoreDistance[]

	0	1	2	3	4	5	6
	0	∞	∞	∞	∞	∞	∞

```
if (StoreDistance[u] + weight < StoreDistance[v])
    StoreDistance[v] = StoreDistance[u] + weight;
```

Bellman and Ford algorithm

Source =u	Destination =v	weight
0	1	6
0	2	5
0	3	5
3	2	-2
2	1	-2
1	4	-1
2	4	1
3	5	-1
4	6	3
5	6	3

StoreDistance[]

0	1	2	3	4	5	6
0	∞	∞	∞	∞	∞	∞

0	6	∞	∞	∞	∞	∞

0 + 6 < ∞

```
if (StoreDistance[u] + weight < StoreDistance[v])
    StoreDistance[v] = StoreDistance[u] + weight;
```

StoreDistance[v] = 0 + 6

Bellman and Ford algorithm

Source =u	Destination =v	weight
0	1	6
0	2	5
0	3	5
3	2	-2
2	1	-2
1	4	-1
2	4	1
3	5	-1
4	6	3
5	6	3

StoreDistance[]

0	1	2	3	4	5	6
0	6	∞	∞	∞	∞	∞

0	6	5	∞	∞	∞	∞

0 + 5 < ∞

```
if (StoreDistance[u] + weight < StoreDistance[v])
    StoreDistance[v] = StoreDistance[u] + weight;
```

StoreDistance[v] = 0 + 5

Bellman and Ford algorithm

Source =u	Destination =v	weight
0	1	6
0	2	5
0	3	5
3	2	-2
2	1	-2
1	4	-1
2	4	1
3	5	-1
4	6	3
5	6	3

StoreDistance[]

0	1	2	3	4	5	6
0	6	5	∞	∞	∞	∞

0	6	5	5	∞	∞	∞

0   +   5   <   ∞

```
if (StoreDistance[u] + weight < StoreDistance[v])
    StoreDistance[v] = StoreDistance[u] + weight;
```

StoreDistance[v] =   0   +   5

Bellman and Ford algorithm

Source =u	Destination =v	weight
0	1	6
0	2	5
0	3	5
3	2	-2
2	1	-2
1	4	-1
2	4	1
3	5	-1
4	6	3
5	6	3

StoreDistance[]

0	1	2	3	4	5	6
0	6	5	5	∞	∞	∞

0	6	3	5	∞	∞	∞

5   +   -2 <   5

```
if (StoreDistance[u] + weight < StoreDistance[v])
    StoreDistance[v] = StoreDistance[u] + weight;
```

StoreDistance[v] =   5   +   -2

Bellman and Ford algorithm

Source =u	Destination =v	weight
0	1	6
0	2	5
0	3	5
3	2	-2
2	1	-2
1	4	-1
2	4	1
3	5	-1
4	6	3
5	6	3

StoreDistance[]

0	1	2	3	4	5	6
0	6	3	5	∞	∞	∞

⬇

0	1	3	5	∞	∞	∞

3   +   -2 <   6

```
if (StoreDistance[u] + weight < StoreDistance[v])
    StoreDistance[v] = StoreDistance[u] + weight;
```

StoreDistance[v] =   3   +   -2

Bellman and Ford algorithm

Source =u	Destination =v	weight
0	1	6
0	2	5
0	3	5
3	2	-2
2	1	-2
1	**4**	**-1**
2	4	1
3	5	-1
4	6	3
5	6	3

StoreDistance[]

0	1	2	3	4	5	6
0	1	3	5	∞	∞	∞

⬇

0	1	3	5	0	∞	∞

$$1 \; + \; -1 \; < \; \infty$$

```
if (StoreDistance[u] + weight < StoreDistance[v])
    StoreDistance[v] = StoreDistance[u] + weight;
```

StoreDistance[v] = 1 + -1

Bellman and Ford algorithm

Source =u	Destination =v	weight
0	1	6
0	2	5
0	3	5
3	2	-2
2	1	-2
1	4	-1
2	4	1
3	5	-1
4	6	3
5	6	3

StoreDistance[]

0	1	2	3	4	5	6
0	1	3	5	0	∞	∞

3   +   1 <   0

```
if (StoreDistance[u] + weight < StoreDistance[v])
    StoreDistance[v] = StoreDistance[u] + weight;
```

不成立

Bellman and Ford algorithm

Source =u	Destination =v	weight
0	1	6
0	2	5
0	3	5
3	2	-2
2	1	-2
1	4	-1
2	4	1
3	5	-1
4	6	3
5	6	3

StoreDistance[]

0	1	2	3	4	5	6
0	1	3	5	0	∞	∞

⬇

0	1	3	5	0	4	∞

$$5 \quad + \quad -1 \quad < \quad \infty$$

```
if (StoreDistance[u] + weight < StoreDistance[v])
    StoreDistance[v] = StoreDistance[u] + weight;
```

StoreDistance[v] = 5 + -1

Bellman and Ford algorithm

Source =u	Destination =v	weight
0	1	6
0	2	5
0	3	5
3	2	-2
2	1	-2
1	4	-1
2	4	1
3	5	-1
4	6	3
5	6	3

StoreDistance[]

0	1	2	3	4	5	6
0	1	3	5	0	4	∞

0	1	3	5	0	4	3

0   +   3   <   ∞

```
if (StoreDistance[u] + weight < StoreDistance[v])
    StoreDistance[v] = StoreDistance[u] + weight;
```

StoreDistance[v] =   0   +   3

Bellman and Ford algorithm

Source =u	Destination =v	weight
0	1	6
0	2	5
0	3	5
3	2	-2
2	1	-2
1	4	-1
2	4	1
3	5	-1
4	6	3
5	6	3

StoreDistance[]

0	1	2	3	4	5	6
0	1	3	5	0	4	3

4   +   3 < 3

```
if (StoreDistance[u] + weight < StoreDistance[v])
    StoreDistance[v] = StoreDistance[u] + weight;
```

不成立

All Pairs Shortest Paths

□Find the shortest paths between all pairs of vertices.

□Solution 1

– Apply shortest path n times with each vertex as source.

$$O(n^3)$$

□Solution 2

– Represent the graph G by its <u>cost adjacency matrix</u> with cost[i][j]

– If the edge <i,j> is not in G, the cost[i][j] is set to some sufficiently large number

– A[i][j] is the cost of the shortest path form i to j, using only those intermediate vertices with an index <= k

All Pairs Shortest Paths *(Continued)*

- The cost of the shortest path from i to j is $A^{n-1}[i][j]$, as no vertex in G has an index greater than n-1

- $A^{-1}[i][j]=cost[i][j]$

- Calculate the A^0, A^1, A^2, ..., A^{n-1} from A^{-1} iteratively

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

Algorithm for All Pairs Shortest Paths

```c
void allcosts(int cost[][MAX_VERTICES],
          int distance[][MAX_VERTICES], int n)
{
  int i, j, k;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
         distance[i][j] = cost[i][j];
  for (k=0; k<n; k++)
    for (i=0; i<n; i++)
      for (j=0; j<n; j++)
        if (distance[i][k]+distance[k][j]
            < distance[i][j])
           distance[i][j]=
              distance[i][k]+distance[k][j];
}
```

* Figure 6.33: Directed graph and its cost matrix
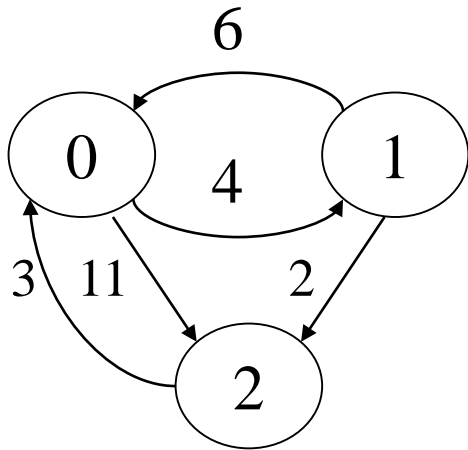
(a)Directed graph G

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

(b)Cost adjacency matrix for G
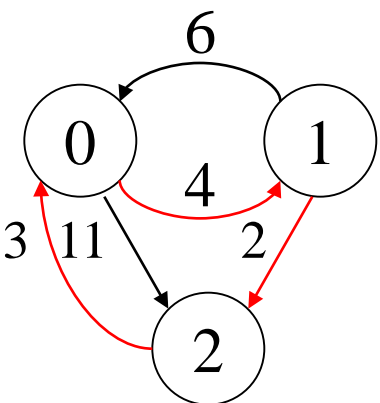
A^{-1}

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

A^{0}

	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A^{-1}

V0 加入

A^{1}

	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^{0}

V1 加入

A^{2}

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

A^{1}

V2 加入
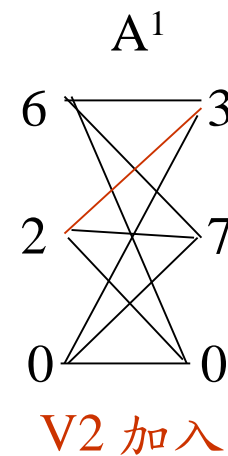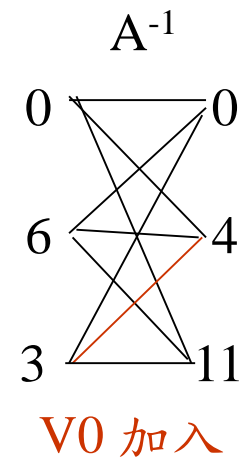
Transitive Closure

Goal: given a graph with unweighted edges, determine if there is a path from i to j for all i and j.
(1) Require positive path (> 0) lengths.   transitive closure matrix
(2) Require nonnegative path (≥ 0) lengths.   reflexive transitive closure matrix
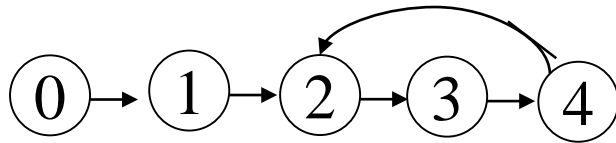
$$
\begin{array}{c}
0 \\ 1 \\ 2 \\ 3 \\ 4
\end{array}
\begin{bmatrix}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0
\end{bmatrix}
$$

(a) Digraph G   (b) Adjacency matrix A for G

$$
\begin{array}{c}
0 \\ 1 \\ 2 \\ 3 \\ 4
\end{array}
\begin{bmatrix}
0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1
\end{bmatrix}
$$
cycle

$$
\begin{array}{c}
0 \\ 1 \\ 2 \\ 3 \\ 4
\end{array}
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1
\end{bmatrix}
$$
reflexive

(c) transitive closure matrix A^+ (d) reflexive transitive closure matrix A^*

There is a path of length > 0 There is a path of length ≥ 0

Activity on Vertex (AOV) Network

- Definition: A directed graph in which the vertices represent tasks or activities and the edges represent precedence relations between tasks.

- Predecessor (successor): vertex i is a predecessor of vertex j iff there is a directed path from i to j.
  - j is a successor of i.

- Partial order: a precedence relation which is both transitive ($\forall i, j, k, i \bullet j \,\&\, j \bullet k => i \bullet k$) and irreflexive ($\forall x \,\neg x \bullet x$).

- Acylic graph: a directed graph with no directed cycles

*Figure 6.37: An AOV network
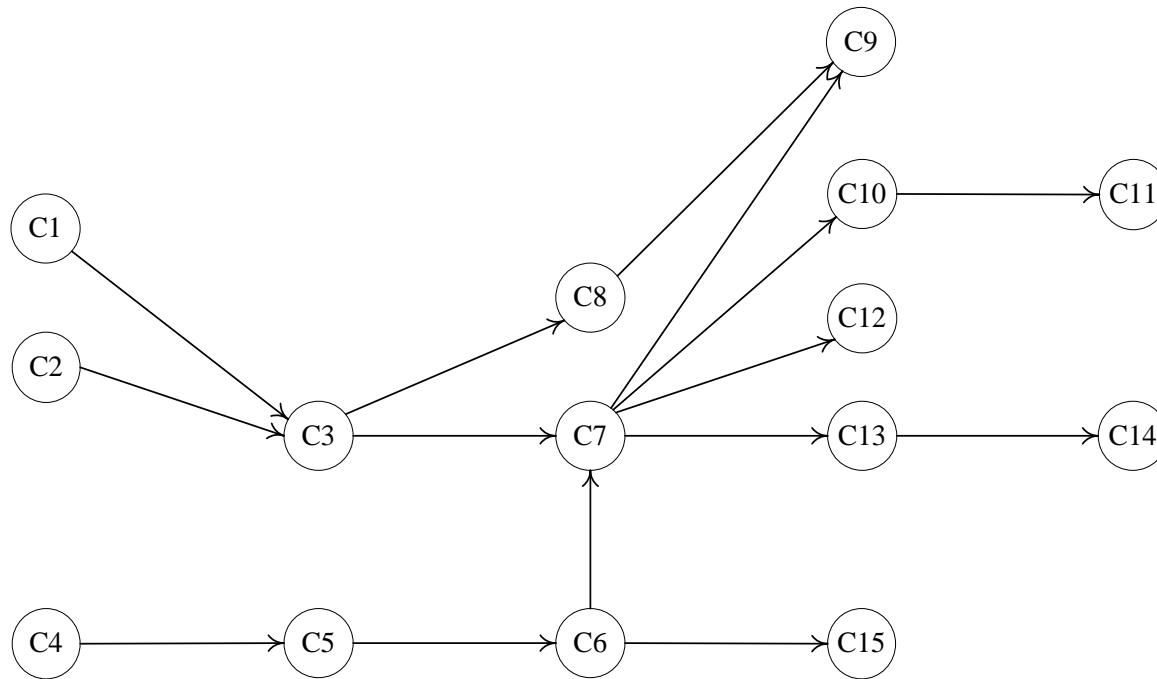
Topological order:
linear ordering of vertices of a graph
$\forall i, j$ if i is a predecessor of j, then i precedes j in the linear ordering

C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9

C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C13, C12, C14

課程編號	課程名稱	先修課程
C1	程式I	無
C2	離散數學	無
C3	資料結構	C1, C2
C4	微積分I	無
C5	微積分II	C4
C6	線性代數	C5
C7	演算法分析	C3, C6
C8	組合語言	C3
C9	作業系統	C7, C8
C10	程式語言	C7
C11	編譯器設計	C10
C12	人工智慧	C7
C13	計算機理論	C7
C14	平行演算法	C13
C15	數值分析	C5

*Program 6.13: Topological sort

```
for (i = 0; i <n; i++) {
    if every vertex has a predecessor {
        fprintf(stderr, "Network has a cycle. \n " );
        exit(1);
    }
    pick a vertex v that has no predecessors;
    output v;
    delete v and all edges leading out of v
    from the network;
}
```

*Figure 6.38: Simulation of Program 6.13 on an AOV network

1. v0 no predecessor
delete v0->v1, v0->v2, v0->v3

2. v1, v2, v3 no predecessor
select v3
delete v3->v4, v3->v5

3. select v2
delete v2->v4, v2->v5
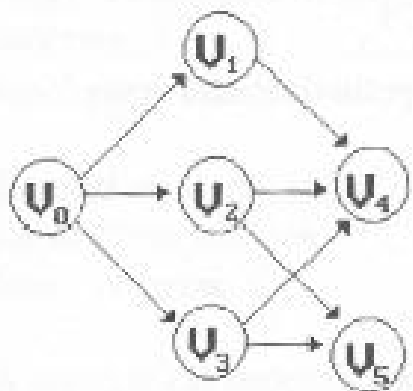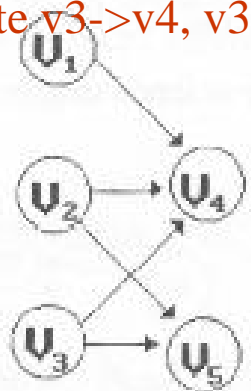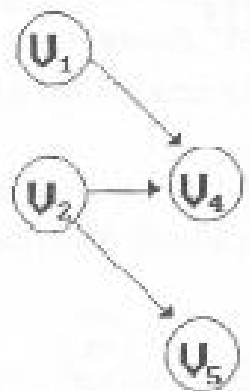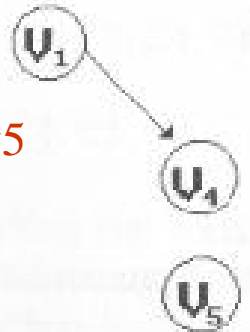
4. select v5

5. select v1
delete v1->v4

(a) initial

(b) v_0

(c) v_3

(d) v_2

(e) v_5

(f) v_1

(g) v_4

Topological order generated: $v_0, v_3, v_2, v_5, v_1, v_4$

Issues in Data Structure Consideration

☐ Decide whether a vertex has any predecessors.

  – Each vertex has a count.

☐ Decide a vertex together with all its incident edges.

  – Adjacency list

*Figure 6.39: Internal representation used by topological sorting algorithm
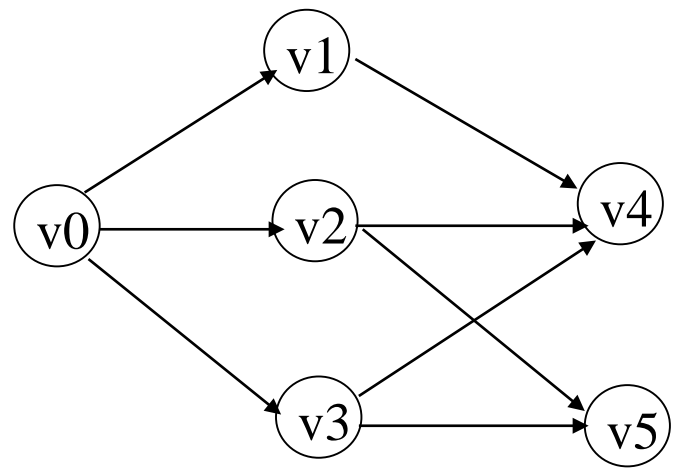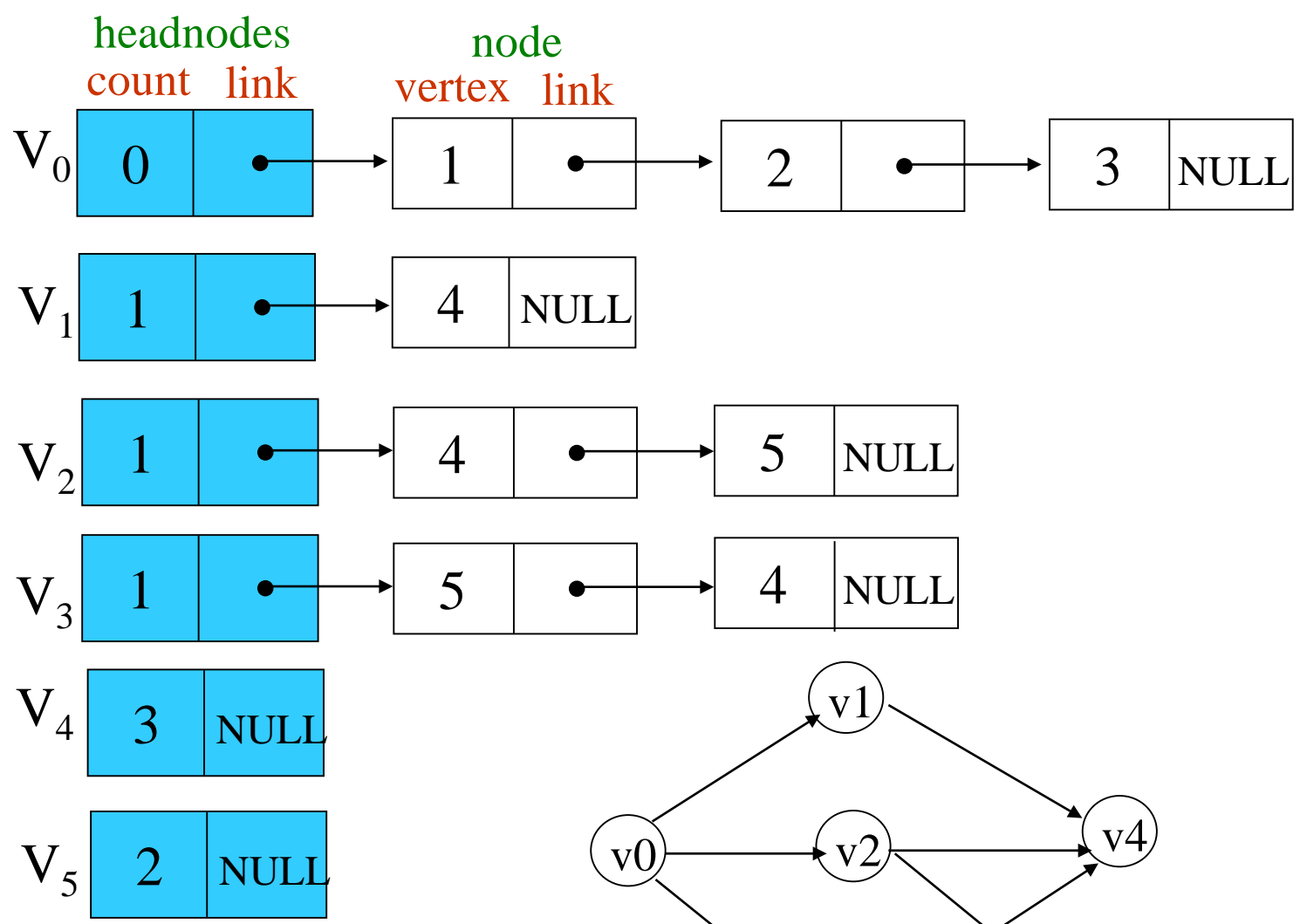
```c
typedef struct node *node_pointer;
typedef struct node {
        int vertex;
        node_pointer link;
        };
typedef struct {
        int count;
        node_pointer link;
        } hdnodes;
hdnodes graph[MAX_VERTICES];
```

*Program 6.14: Topological sort

```
void topsort (hdnodes graph [] , int n)
{
  int i, j, k, top;
  node_pointer ptr;
  /* create a stack of vertices with no predecessors */
  top = -1;
  for (i = 0; i < n; i++)
     if (!graph[i].count) {   no predecessors, stack is linked through count field
        graph[i].count = top;
        top = i;
  }
  for (i = 0; i < n; i++)
    if (top == -1) {
       fprintf(stderr, "\n Network has a cycle. Sort terminated. \n");
       exit(1);
     }
```

O(n)

```
        }
    else {
        j = top; /* unstack a vertex */
        top = graph[top].count;
        printf("v%d, ", j);
        for (ptr = graph [j].link; ptr ; ptr = ptr ->link ){
        /* decrease the count of the successor vertices of j */
            k = ptr ->vertex;
            graph[k].count --;
            if (!graph[k].count) {
            /* add vertex k to the stack*/
                graph[k].count = top;
                top = k;
            }
        }
    }
}
```

O(e)

O(e+n)

Activity on Edge (AOE) Networks

☐ Directed edge

– tasks or activities to be performed

☐ Vertex

– events which signal the completion of certain activities

☐ Number

– time required to perform the activity
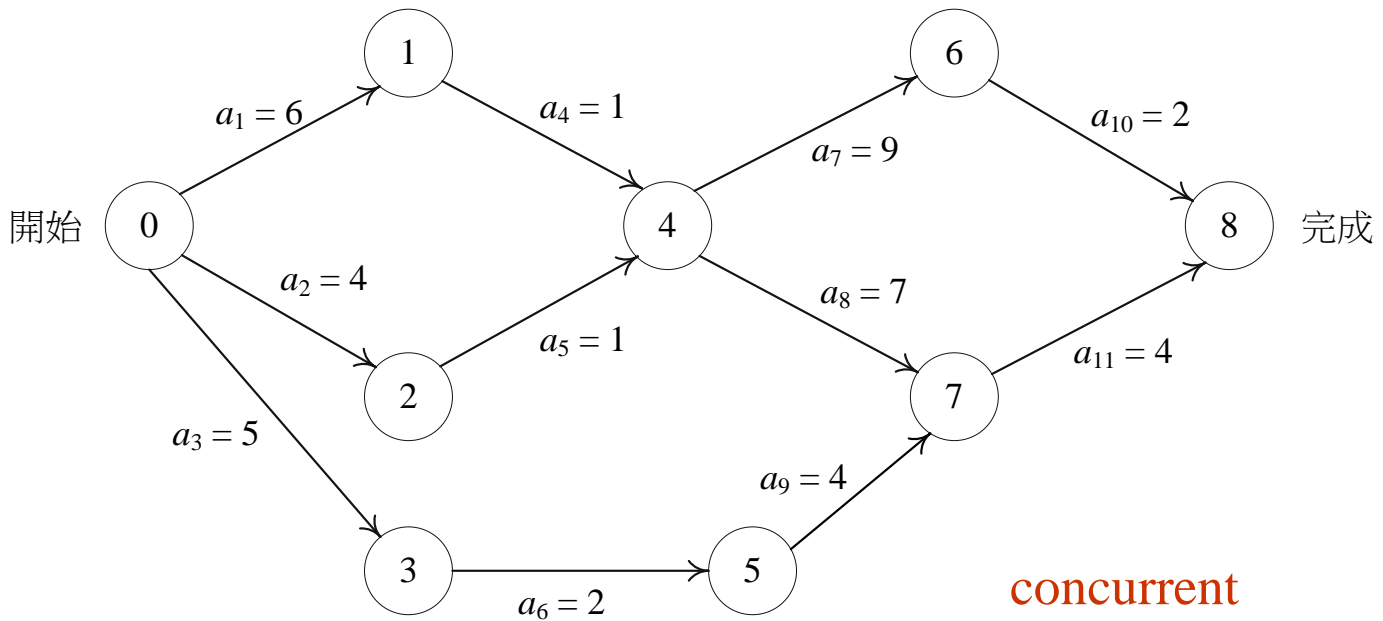
(Fig. 6.40)

事件	解釋
0	計劃開始
1	活動a_1完成
4	活動a_4和a_5完成
7	活動a_8和a_9完成
8	計畫完成

Application of AOE Network

☐ Evaluate performance
- – minimum amount of time
- – activity whose duration time should be shortened
- – …

☐ Critical path
- – **a path that has the longest length**
- – minimum time required to complete the project
- – v0, v1, v4, v7, v8 or v0, v1, v4, v6, v8

AOE

- Earliest time that vi can occur
  - the length of the longest path from v0 to vi
  - the earliest start time for all activities leaving vi
  - early(7) = early(8) = 7
- Latest time of activity
  - the latest time the activity may start without increasing the project duration
  - late(6) = 8, late(8) = 7

Critical activity

  - an activity for which early(i)=late(i)
  - early(7)=late(7)=14
- late(i)-early(i)
  - measure of how critical an activity is
  - late(5)-early(5)=10-7=3

earliest, early, latest, late

153
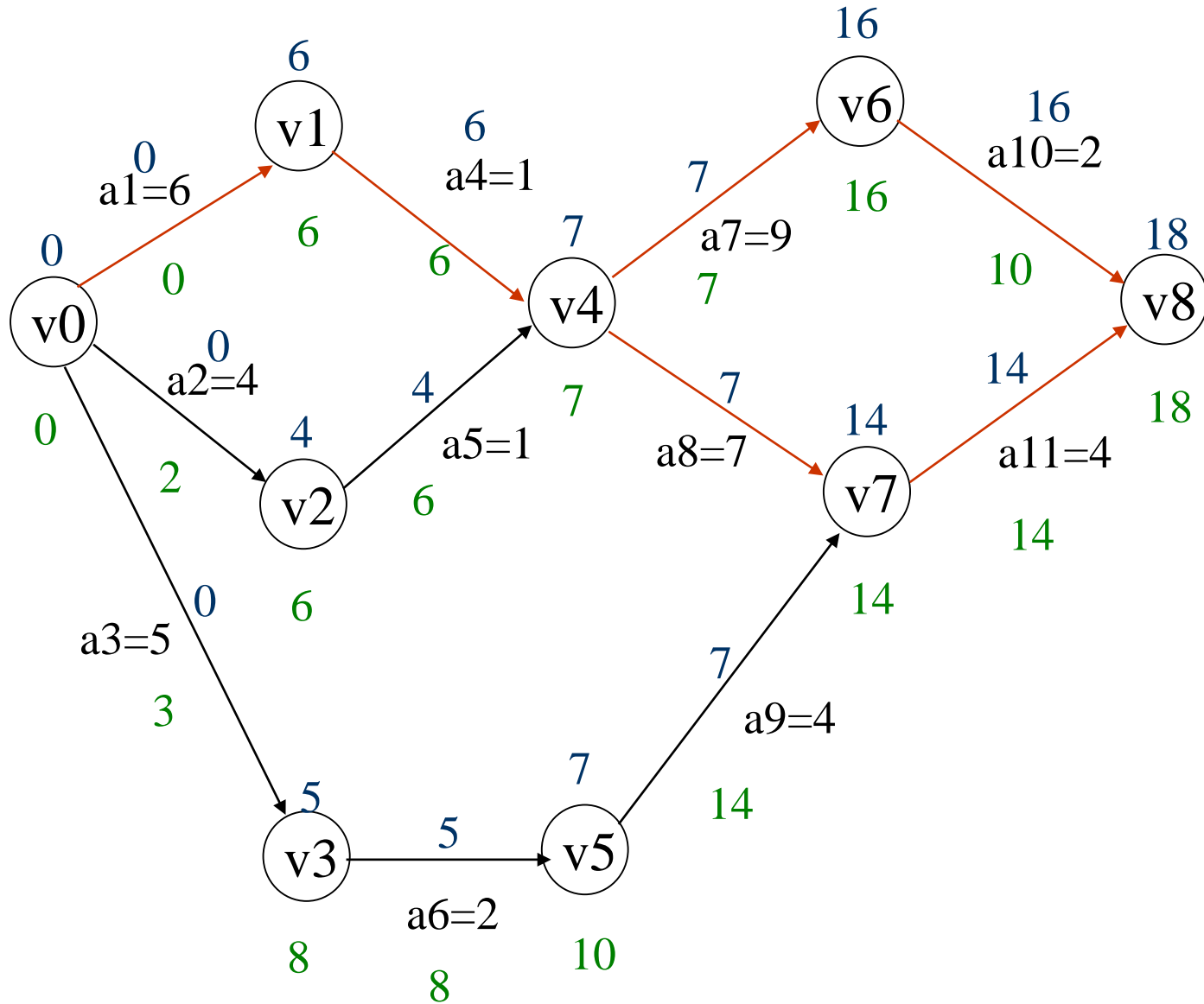
Determine Critical Paths

☐ Delete all noncritical activities

☐ Generate all the paths from the start to finish vertex.

Calculation of Earliest Times

☐ earliest[j]

– the earliest event occurrence time

earliest[0]=0

earliest[j]=$\max_{i \in p(j)}$ {earliest[i]+duration of <i,j>}

☐ latest[j]

– the latest event occurrence time

$$v_k \xrightarrow{a_i} v_l$$
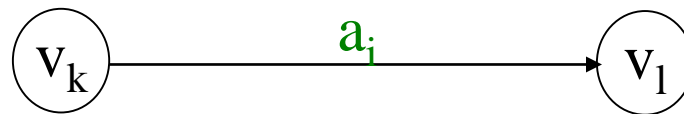
early(i)=earliest(k)

late(i)=latest(l)-duration of a_i

forward stage
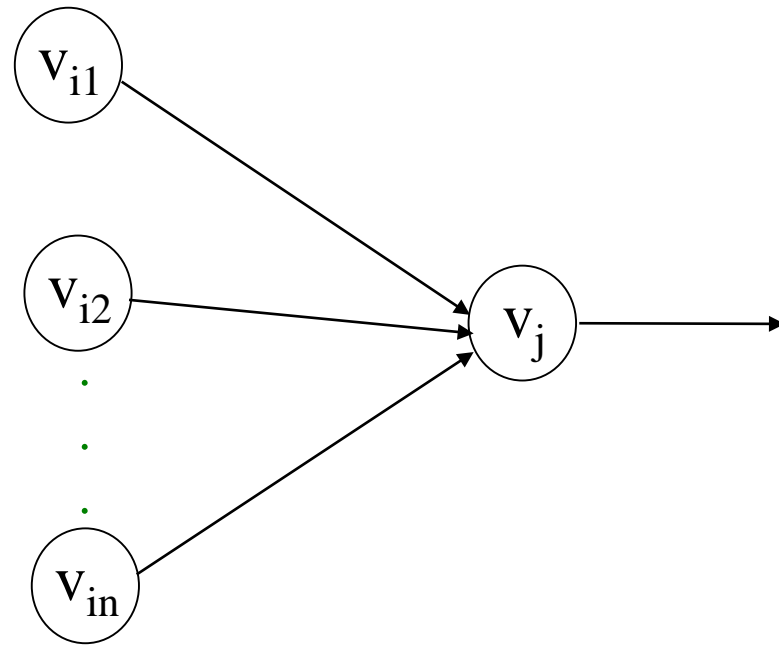
if (earliest[k] < earliest[j]+ptr->duration)
earliest[k]=earliest[j]+ptr->duration

count first

```
[0]  0  →  [1 | 6 | →] → [2 | 4 | →] → [3 | 5 | 0]
[1]  1  →  [4 | 1 | 0]
[2]  1  →  [4 | 1 | 0]
[3]  1  →  [5 | 2 | 0]
[4]  2  →  [6 | 9 | →] → [7 | 7 | 0]
[5]  1  →  [7 | 4 | 0]
[6]  1  →  [8 | 2 | 0]
[7]  2  →  [8 | 4 | 0]
[8]  2  0
```

(a) 圖 6.40(a) 的相鄰串列

開始 0

$a_1 = 6$, $a_2 = 4$, $a_3 = 5$, $a_4 = 1$, $a_5 = 1$, $a_6 = 2$, $a_7 = 9$, $a_8 = 7$, $a_9 = 4$, a_{10}, a_{11}
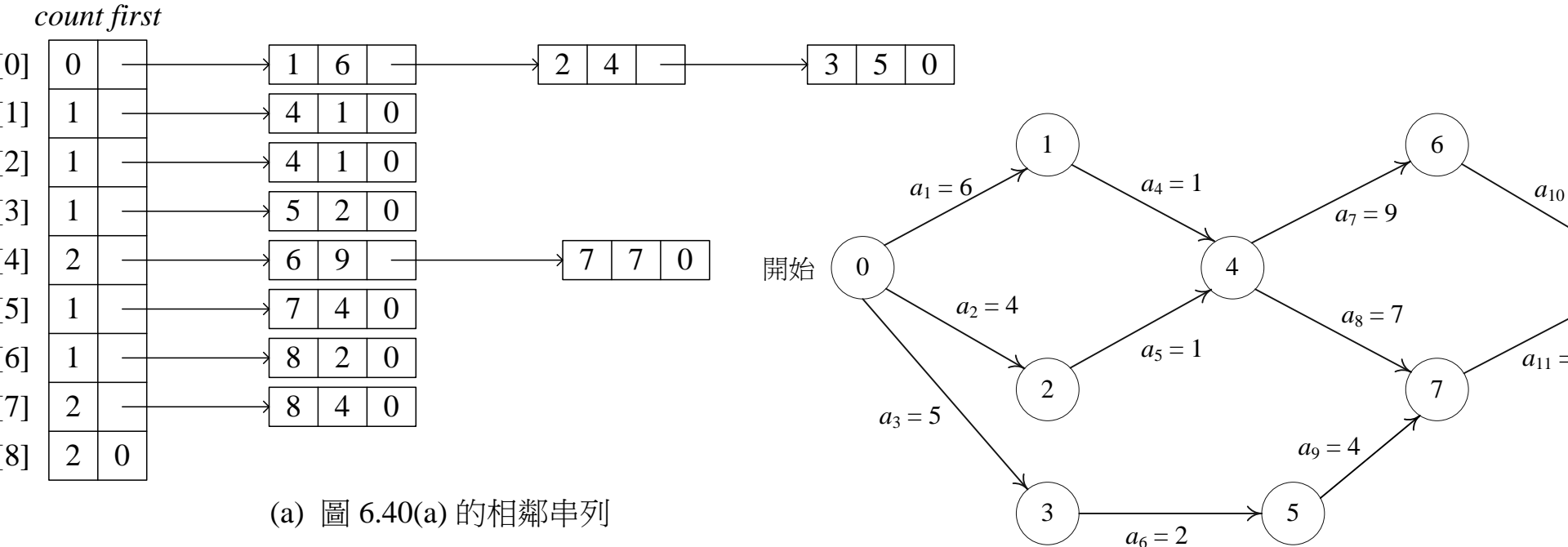
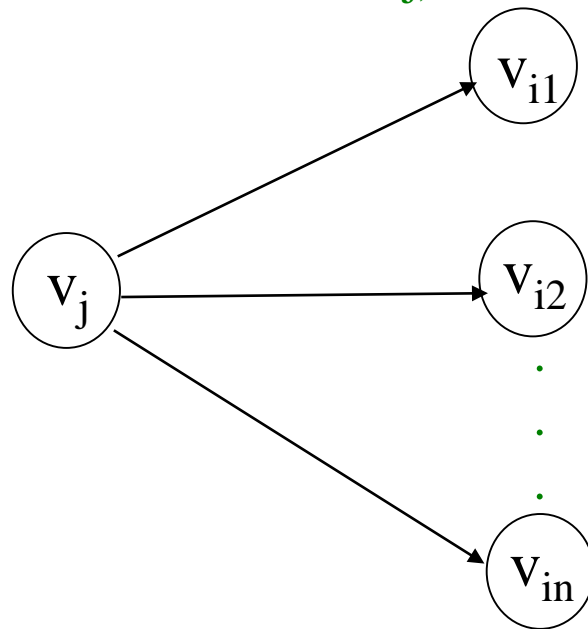| ee	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	堆疊
起始	0	0	0	0	0	0	0	0	0	[0]
輸出 0	0	6	4	5	0	0	0	0	0	[3, 2, 1]
輸出 3	0	6	4	5	0	7	0	0	0	[5, 2, 1]
輸出 5	0	6	4	5	0	7	0	11	0	[2, 1]
輸出 2	0	6	4	5	5	7	0	11	0	[1]
輸出 1	0	6	4	5	7	7	0	11	0	[4]
輸出 4	0	6	4	5	7	7	16	14	0	[7, 6]
輸出 7	0	6	4	5	7	7	16	14	18	[6]
輸出6	0	6	4	5	7	7	16	14	18	[8]
輸出 8										

(b) *ee* 的計算

Calculation of Latest Times

☐ latest[j]

   – the latest event occurrence time

$$latest[j] = \min_{i \in s(j)} \{latest[i] - \text{duration of } <j,i>\}$$

backward stage

if (latest[k] > latest[j]-ptr->duration)
    latest[k]=latest[j]-ptr->duration

(a) Inverted adjacency lists for AOE network of Figure 6.41(a)

Latest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	18	18	18	18	18	18	18	18	18	[8]
output v_8	18	18	18	18	18	18	16	14	18	[7, 6]
output v_7	18	18	18	18	7	10	16	14	18	[5, 6]
output v_5	18	18	18	18	7	10	16	14	18	[3, 6]
output v_3	3	18	18	8	7	10	16	14	18	[6]
output v_6	3	18	18	8	7	10	16	14	18	[4]
output v_4	3	6	6	8	7	10	16	14	18	[2, 1]
output v_2	2	6	6	8	7	10	16	14	18	[1]
output v_1	0	6	6	8	7	10	16	14	18	[0]

(b) Computation of *latest*

***Figure 6.43(continued):**Computing latest of AOE network of Figure 6.41(a)

latest[8]=earliest[8]=18
latest[6]=min{le[8] - 2}=16
latest[7]=min{le[8] - 4}=14
latest[4]=min{le[6] - 9; le[7] -7}= 7
latest[1]=min{le[4] - 1}=6
latest[2]=min{le[4] - 1}=6
latest[5]=min{le[7] - 4}=10
latest[3]=min{le[5] - 2}=8
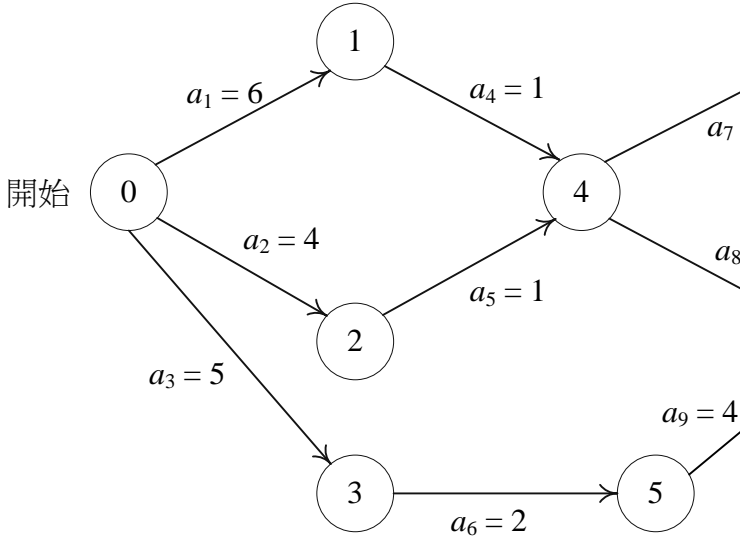latest[0]=min{le[1] - 6; le[2]- 4; le[3] -5}=0

(c) Computation of latest from Equation (6.3) using a reverse topological order
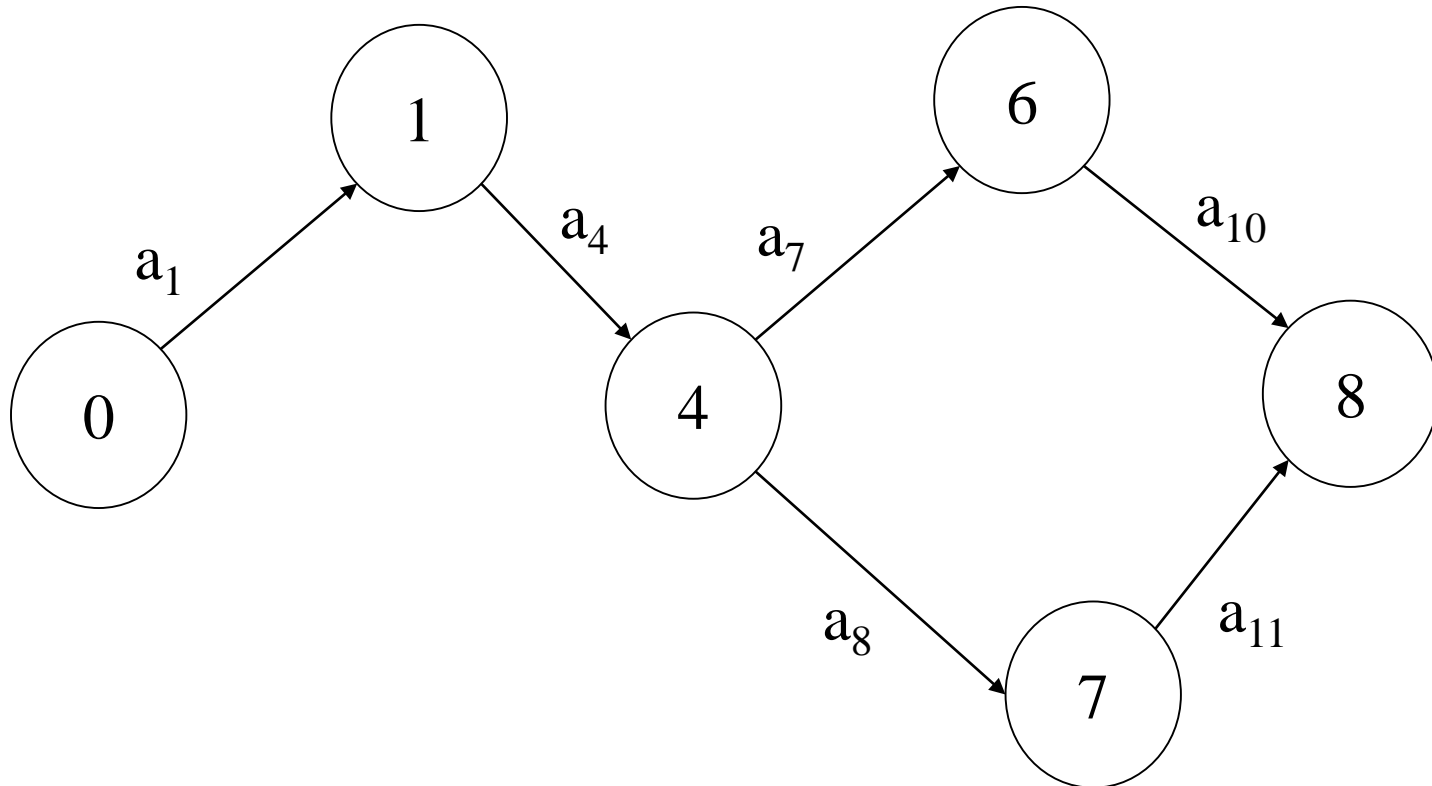
*Figure 6.42: Early, late and critical values

Activity	Early	Late	Late-Early	Critical
a_1	0	0	⓪	Yes
a_2	0	2	2	No
a_3	0	3	3	No
a_4	6	6	⓪	Yes
a_5	4	6	2	No
a_6	5	8	3	No
a_7	7	7	⓪	Yes
a_8	7	7	⓪	Yes
a_9	7	10	3	No
a_{10}	16	16	⓪	Yes
a_{11}	14	14	⓪	Yes

$l - e = 0$

162

*Figure 6.43:Graph with noncritical activities deleted

Figure 6.45: AOE network with unreachable activities