CHAPTER 6

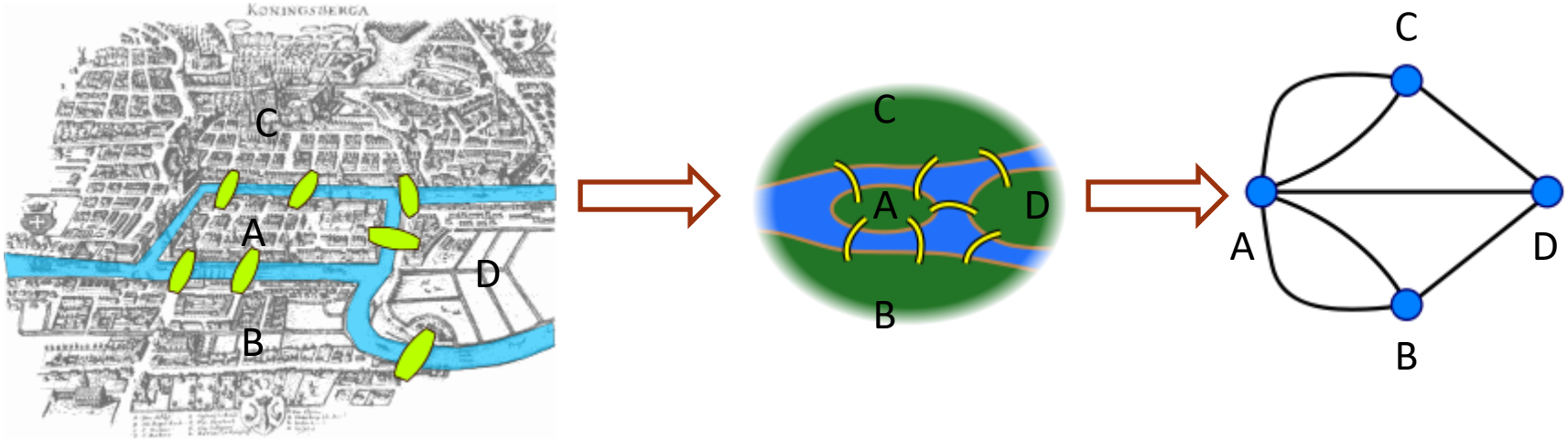
Graphs

Outline

- **The Graph Abstract Data Type（圖形抽象資料型態）**
- Elementary Graph Operations（基本的圖形運算）
- Minimum Cost Spanning Trees（最小成本生成樹）
- Shortest Paths（最短路徑）

Königsberg Bridge Problem

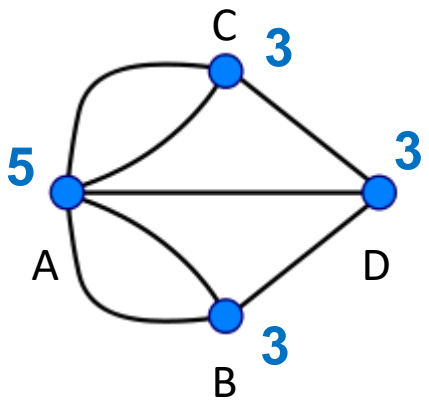
- Can we walk across all the bridges **exactly once** in returning back to the starting land area ?
  (從某一地區出發, 使否可能在經過每一橋樑恰好一次以後回到原來的出發點?)



- Transferring to "graph" model
  (轉換成圖的模式)
  - Land → vertex (地區→頂點、節點)
  - Bridge → edge (橋→邊)

Eulerian Walk

- A walk that does this is called **Eulerian**.
  （我們稱這種走法為**尤拉路線**）
- Euler showed that the graph contains an Eulerian walk **if and only if (iff) the degree of each vertex is even.**
  （尤拉證明圖形存在尤拉路線若且唯若每一頂點的是偶數的）
- He answered the Königsberg problem in the negative.
  （他證明哥尼斯堡七橋問題不存在尤拉路線）
- There is no Eulerian walk for Königsberg problem as all four vertices are of odd degree
  （因為四個頂點的分支度都是奇數，
  所以科尼格茲堡七橋問題不存在尤拉路線）



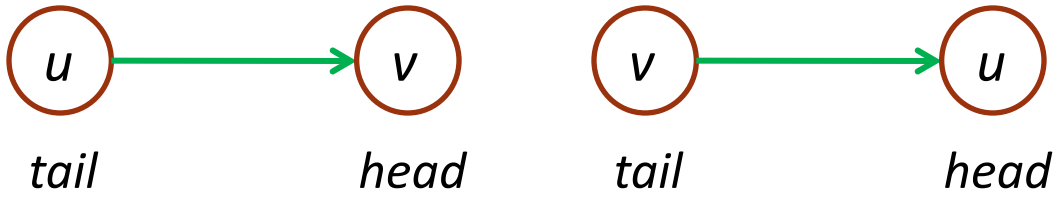
degree（分支度）：進入頂點的邊數

Definition of A graph

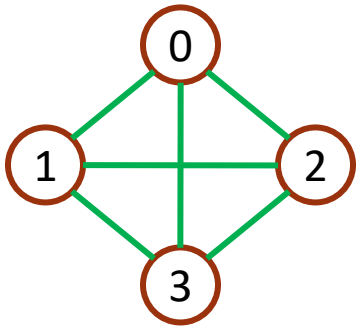
- A **graph**, **G**, consists of two sets, *V* and *E*
  (圖形 *G* 包含兩個集合, *V* 和 *E*)
  - *V* is a finite, nonempty set of vertices.
    (*V* 是有限, 非空的點集合)
  - *E* is a set of pairs of vertices; these pairs are called edges.
    (*E* 是由成對的點所形成的集合, 我們稱這些成對的點為邊)
  - *V*(*G*) and *E*(*G*) will present the sets of vertices and edges, respectively, of *G*.
    (*V*(*G*)和*E*(*G*)分別來表示圖形 *G* 中的頂點集合和邊集合)
  - We will also write ***G* = (*V*, *E*)** to represent a graph.
    (或者可寫成 *G* = (*V*, *E*) 來表示一個圖形)

Definition of A graph

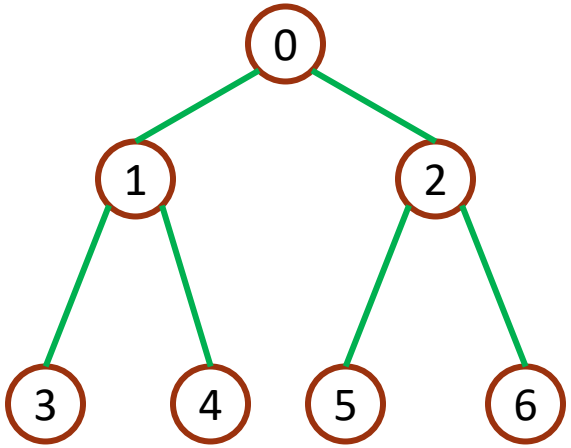
- Graphs have two different types:
  (圖形可分為兩種不同型態)
  - **Undirected graph**: **(*u*, *v*)** and **(*v*, *u*)** represent the same edge
    (無向圖形中，(*u*, *v*) 和 (*v*, *u*) 代表同一邊)
  - **Directed graph**: a directed pair **<*u*, *v*>** has *u* as the tail and the *v* as the head. So <*u*, *v*> and <*v*, *u*> indicate different edges.
    (有向圖形中，序對 <*u*, *v*> 代表尾部是 *u* , 頭部是 *v* 的邊. 所以, <*u*, *v*> 和 <*v*, *u*> 代表不同邊)



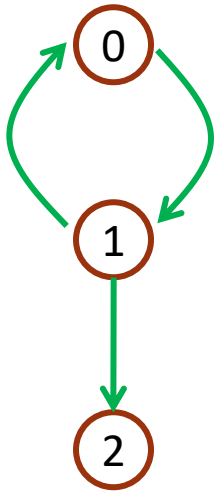
Three Sample Graphs



G_1



G_2



G_3

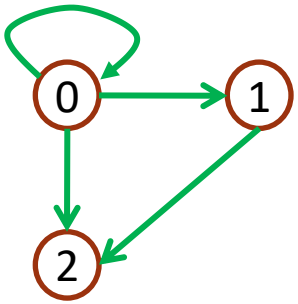
$V(G_1)=\{0, 1, 2, 3\}$
$E(G_1)=\{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$

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

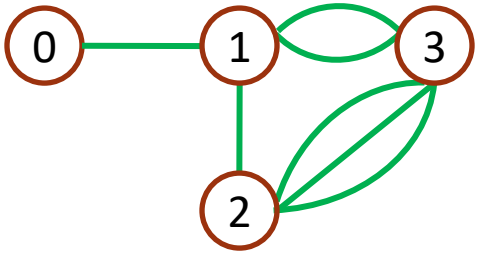
$V(G_3)=\{0, 1, 2\}$
$E(G_3)=\{<0, 1>, <1, 0>, <1, 2>\}$

Figure 6.3: Examples of graphlike structures

- **We impose the following restrictions on graphs**:
  (我們對圖形增加了下列的限制)
  - A graph may not have an edge from a vertex, *v*, back to itself. That is, **edges of the form (*v*, *v*) and <*v*, *v*> are not legal.** Such edges are known as **self edges** or **self loops**. (圖形不可以有一個邊從頂點 *v* 連到自身. 也就是說邊 (*v*, *v*) 和 <*v*, *v*> 是不允許的. 我們稱這些邊為**自我邊**或**自我迴路(自我迴圈)**.)
  - A graph may not have multiple occurrences of the same edges. Such graphs are known as **multigraph**.
    (圖形的同一邊不可以重覆. 我們稱這些有重邊的圖為**重邊圖形**)



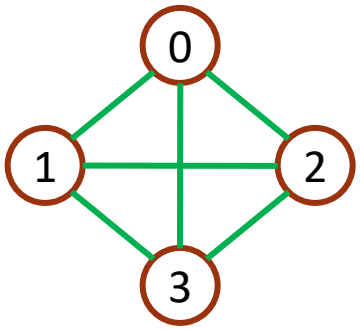
Graph with a self edge



Multigraph

Complete Graph（完全圖）

- The number of distinct unordered pairs (u, v) with $u \neq v$ in a graph with n vertices: $n(n\text{-}1)/2$.
  （一個具有 n 個頂點的無向圖最多有 $n(n\text{-}1)/2$ 個邊）
- An n-vertex, **undirected graph** with exactly $n(n\text{-}1)/2$ edges is said to be **complete**.
  （一個有$n(n\text{-}1)/2$ 個邊的無向圖為完全無向圖）
- In the case of a **directed graph** on n vertices, the maximum number of edges is $n(n\text{-}1)$.
  （一個具有 n 個頂點的有向圖最多有 $n(n\text{-}1)$ 個邊）



A complete graph with 4 vertices and 6 edges.

Subgraph and Path

- If (u, v) is an edge in $E(G)$, then the vertices u and v are **adjacent** and that the edge (u, v) is **incident** on vertices u and v.
  （如果 (u, v) 是圖上的一個邊，則 u 和 v 是**相鄰的**，且邊 (u, v) **附著於**頂點 u 和 v ）

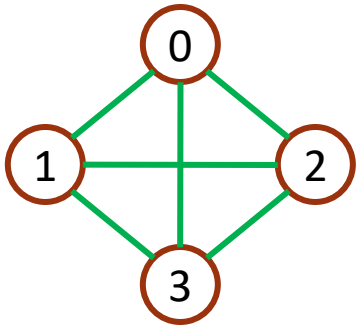
- A **subgraph**（**子圖**）of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.
  （如果 $V(G') \subseteq V(G)$ 且 $E(G') \subseteq E(G)$，我們稱 G' 為 G 的子圖）

- A **path**（**路徑**）from vertex u to vertex v in graph G is a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$, such that $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$ are edges in $E(G)$.
  （在圖形 G 中從頂點 u 到 v 的一個路徑是指一連串的頂點 $u, i_1, i_2, \ldots, i_k, v$，其中 $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$ 都是無向圖中的邊）

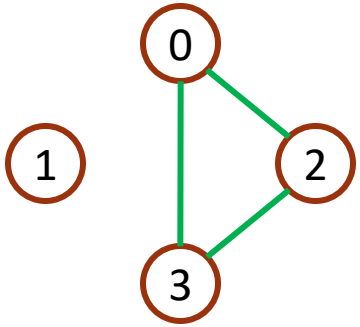
Subgraph, Path and Cycle

- If G is **directed**, the path consists of $<u, i_1>, <i_1, i_2>, \ldots, <i_k, v>$ edges in $E(G)$.
  (如果是有向圖形, 則路徑包含 $<u, i_1>, <i_1, i_2>, \ldots, <i_k, v>$)
- The **length** of a path is **the number of edges on it**.
  (路徑的長度等於其上所有的邊數)
- A **simple path** a path in which all vertices, except possibly the first and the last, are distinct.
  (簡單路徑是除了第一個節點與最後一個節點外的**所有點都不相同**的路徑)
- A **cycle** is a **simple path** in which **the first and last vertices are the same**.
  (迴路/迴圈是一個簡單路徑, 其中第一個節點與最後一個節點相同)

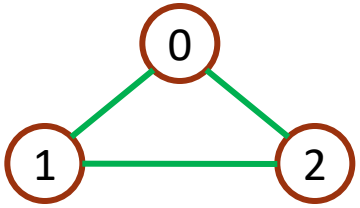
Some subgraphs



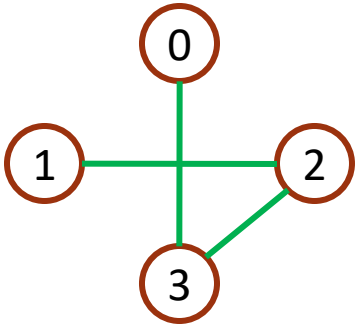
G_1



(a)



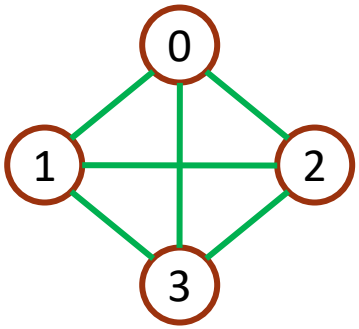
(b)



(c)

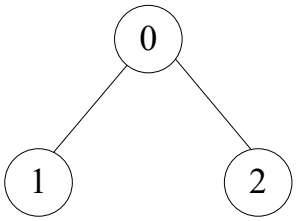
Graphs (a), (b) and (c) are the subgraph of G_1

Some subgraphs

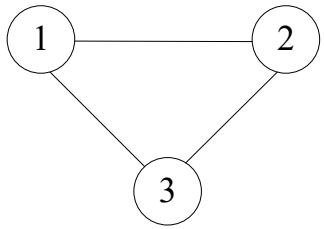


G_1

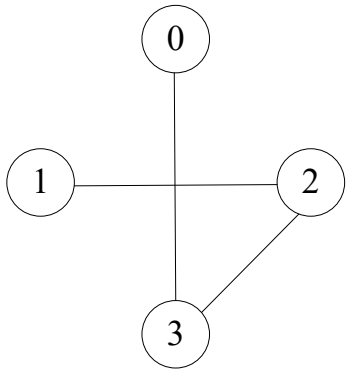
(i)



(ii)

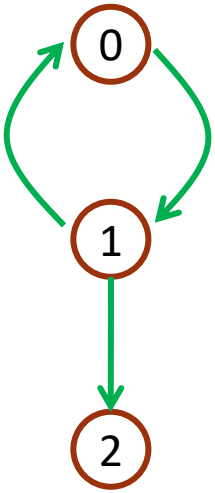


(iii)



(iv)

(a) 一些 G_1 的子圖

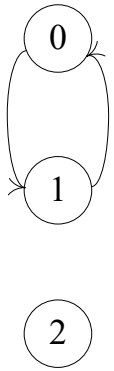


G_3

(i)

(ii)

(iii)



(iv)

(b) 一些 G_3 的子圖

Connected and Connected Component

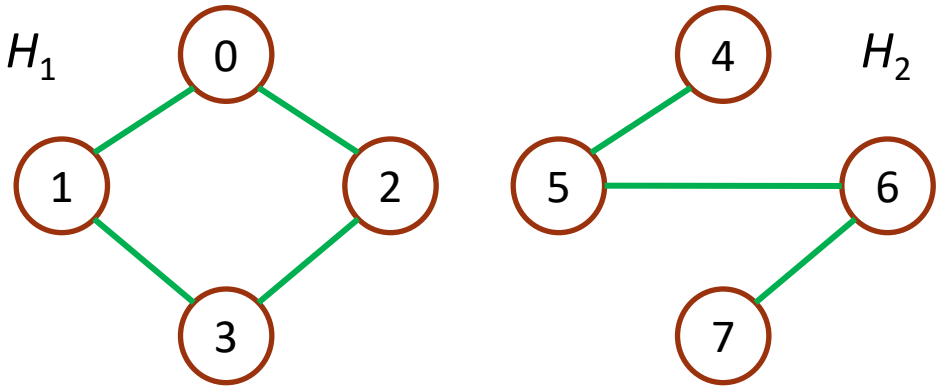
- In an undirected graph, *G*, two vertices ***u* and *v*** are **connected iff** there is a path in *G* from *u* to *v*.
  （在無向圖 *G* 中，存在一個 *u* 到 *v* 的路徑若且唯若 *u* 和 *v* 是連通的）
- An undirected **graph** is **connected iff** for every pair of district vertices *u* and *v* in *V*(*G*) there is a path from *u* to *v* in *G*.
  （一個無向圖是連通的若且唯若在 *G* 中的任意兩點 *u* 和 *v* 都有一條路徑）
- A **connected component** (or simply a component), *H*, of an undirected graph is a **maximal** connected subgraph.
  （一個無向圖的連通元件是其中最大的連通子圖）
- A **tree** is a connected acyclic (i.e., has no cycles) graph.
  （樹是連通但沒有迴路的圖形）

Strongly Connected Graph

- A **directed graph** *G* is said to be **strongly connected iff** for every pair of district vertices *u* and *v* in *V*(*G*), there is directed path from *u* to *v* and also from *v* to *u*.
  (一個有向圖是強連通的若且唯若在 *G* 中的任意兩點 *u* 和 *v* 都有一條 ***u* 到 *v* 的路徑**而且**有一條 *v* 到 *u* 的路徑**)

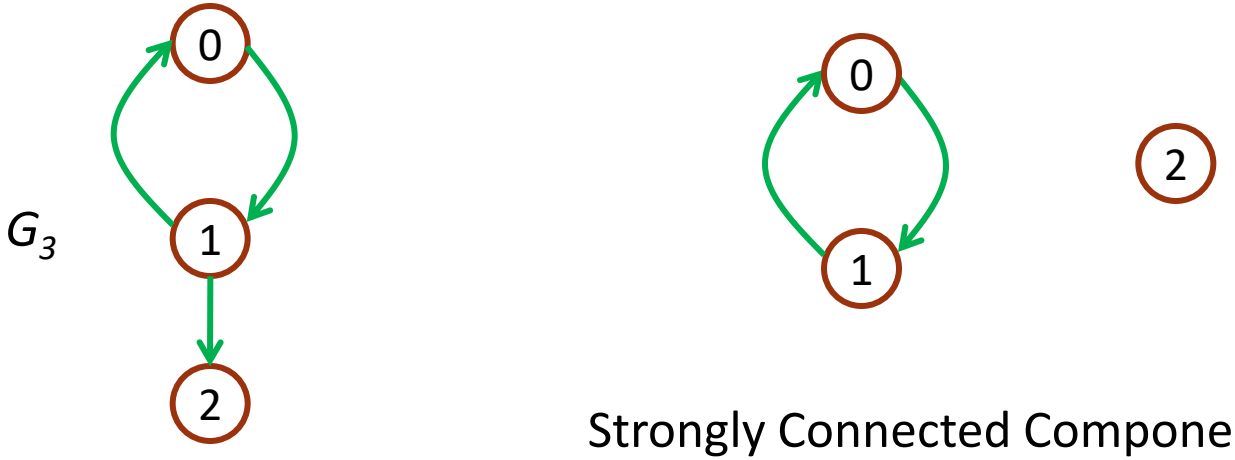
- A **strongly connected component** is a maximal subgraph that is strongly connected.
  (一個強連通元件是一個強連通的最大子圖)

Connected Components



G_4

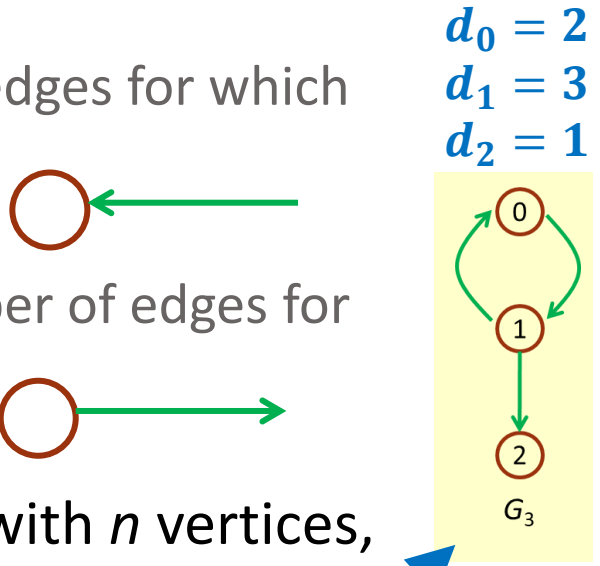
A Graph with two Connected Components



Strongly Connected Components of G_3

Degree of a Vertex

- The **degree** of a vertex is the number of edges incident to that vertex.
  (一個頂點的**分支度**是附著於該頂點的邊數)
- If *G* is a directed graph:
  (如果 *G* 是個有向圖)
  - The **in-degree** of a vertex is the number of edges for which vertex is head.
    (頂點的 *v* 的入分支度是以頂點 *v* 為頭部的邊數)
  - And, the **out-degree** of a vertex is the number of edges for which the vertex is the tail.
    (頂點的 *v* 的出分支度是以頂點 *v* 為尾部的邊數)
- If d_i is the degree of the vertex *i* in a graph with *n* vertices, then the number of edges is $(\sum_{i=0}^{n-1} d_i)/2$.
  (如果 d_i 是頂點 *i* 的分支度, 則一個有 *n* 個頂點的圖, 它的邊總數為 $\sum_{i=0}^{n-1} d_i)/2$



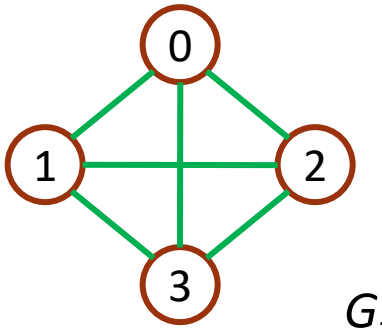
Graph Representations

- There are two most commonly used representations for graph:
  (常用的圖形表示法有兩種)
  - **Adjacency matrices**
    (相鄰矩陣)
  - **Adjacency lists**
    (相鄰串列)
- The choice of a particular representation will depend upon the application one has in mind and the functions one expects to perform on the graph. (不同的應用選擇的表示法也不同)

Adjacency Matrix（相鄰矩陣）

- The **adjacency matrix** of *G* is a two-dimensional *n* x *n* array *a*, with the property that
  （圖形 *G* 的**相鄰矩陣**是一個 *n* x *n* 的二維矩陣, 具有下列性質）
  - *a*[*i*][*j*] = 1 iff the edge (*i*, *j*) is in *E*(*G*)
    （*a*[*i*][*j*] = 1 若且唯若邊 (*i*, *j*) 在 *E*(*G*) 中）
  - *a*[*i*][*j*] = 0 iff there is no such edge in *G*
    （*a*[*i*][*j*] = 0 若且唯若邊 (*i*, *j*) 不在 *E*(*G*) 中）
  - **The adjacency matrix for an undirected graph is symmetric.**
    （無方向圖形的相鄰矩陣是對稱的）
- For an **undirected graph** the **degree** of any vertex *i* is its row sum.（對於無向圖, 任一頂點的分支度為其各列之和）
- For an **directed graph** the row sum is its **out-degree** and the column sum is its **in-degree**.
  （對於有向圖, 各列之和為出分支度, 各行之和為入分支度）

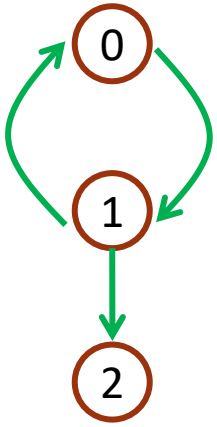
Figure 6.7: Adjacency matrices 1/2



G_1

	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

The adjacency matrix of G_1



G_3

	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

The adjacency matrix of G_3

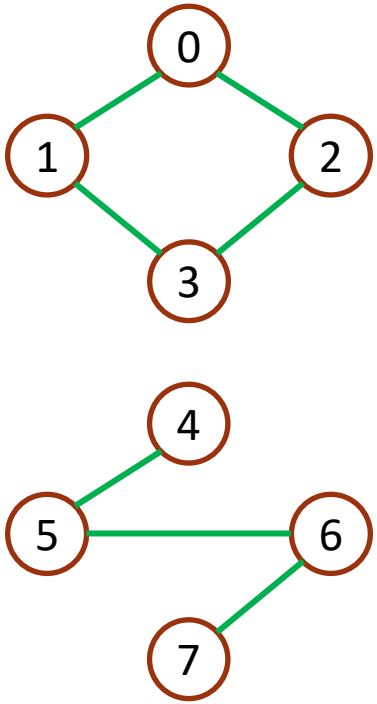
The adjacency matrix for an undirected graph is symmetric.（無方向圖形的相鄰矩陣是對稱的）

For an **undirected graph** the **degree** of any vertex *i* is its row sum.（對於無向圖，任一頂點的分支度為其各列之和）

For an **directed graph** the row sum is its **out-degree** and the column sum is its **in-degree**.

（對於有向圖，各列之和為出分支度，各行之和為入分支度）

Figure 6.7: Adjacency matrices 2/2



G_4

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

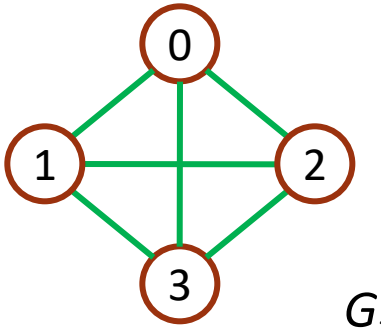
The adjacency matrix of G_4

The adjacency matrix for an undirected graph is symmetric.（無方向圖形的相鄰矩陣是對稱的）
For an **undirected graph** the **degree** of any vertex *i* is its row sum.（對於無向圖，任一頂點的分支度為其各列之和）
For an **directed graph** the row sum is its **out-degree** and the column sum is its **in-degree**.
（對於有向圖，各列之和為出分支度，各行之和為入分支度）

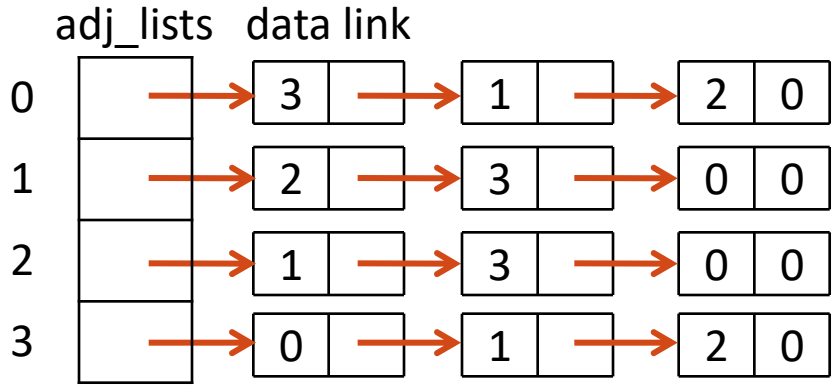
Adjacency Lists（相鄰串列）

- The *n* rows of the adjacency matrix are represented as *n* chains.
  （我們用*n* 個鏈結串列取代相鄰矩陣中的 *n* 個列）
- There is one chain for each vertex in *G*.
  （圖形 *G* 中的每一頂點各有一個串列）
- The node in chain *i* represent the vertices that are adjacent from vertex *i*.
  （對任一串列 *i*，**串列中的節點包含與頂點 *i* 相鄰的頂點**）
- The **data** field of a chain node stores the index of an adjacent vertex.
  （節點中的資料欄位儲存相鄰節點的編號）

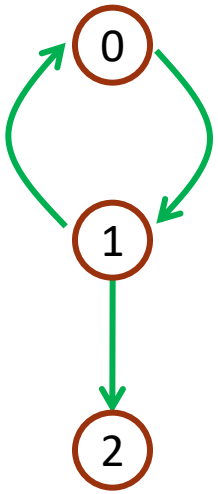
Figure 6.8: Adjacent lists



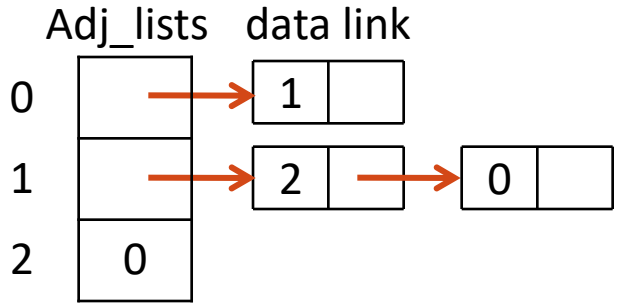
G_1



Adjacent list of G_1



G_3



Adjacent list of G_3

Weighted Edges

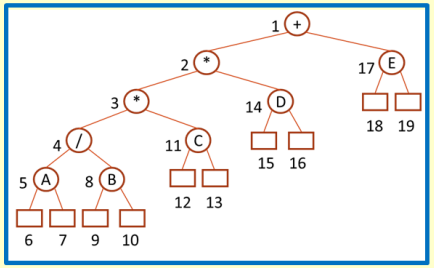
- In many applications, the edges of a graph have **weights** associated with them.
  (在許多應用上, 圖形的各邊會被指定一個**權重**)
- These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex. (這些權重可以表示頂點到其相鄰頂點的距離或一個頂點到其相鄰頂點之間的成本)
  - The adjacency matrix entries a[i][j] would keep this information.
    (相鄰矩陣可以使用 a[i][j] 來儲存這項資訊)
  - When adjacency lists are used, the weight information may be kept in the list nodes by **including an additional field, *weight***.
    (使用相鄰串列時, 可以在節點構造中增加權重欄位)

Outline

- The Graph Abstract Data Type（圖形抽象資料型態）
- **Elementary Graph Operations（基本的圖形運算）**
- Minimum Cost Spanning Trees（最小成本生成樹）
- Shortest Paths（最短路徑）

Elementary Graph Operations

- Given an undirected graph, $G = (V, E)$, and a vertex v in $V(G)$, we wish to visit all vertices in G that are reachable from v.
  (給定一個無向圖, 我們想要找出從頂點 v 可以到達的所有 G 中的頂點)
- We shall look at two ways of doing this:
  (我們將探討完成這項工作的兩種方法)
  - **Depth First Search** (DFS) is similar to a **preorder tree traversal**.
    (深度優先搜尋類似樹的**前序走訪**)
  - **Breadth First Search** (BFS) is similar to a **level-order tree traversal**.
    (廣度/寬度優先搜尋類似樹的**階序走訪**)
- In the following discussion, we shall assume that the linked adjacency list representation for graphs is used.
  (接下來的討論, 我們將假設採用圖形的鏈結相鄰串列表示)



Inorder traversal: A/B*C*D+E (LVR)
Preorder traversal: +/ABCDE (VLR)**
Postorder traversal: AB/C*D*E+ (LRV)
Level-order traversal: +*E*D/CAB

Depth First Search (DFS)$_{1/2}$

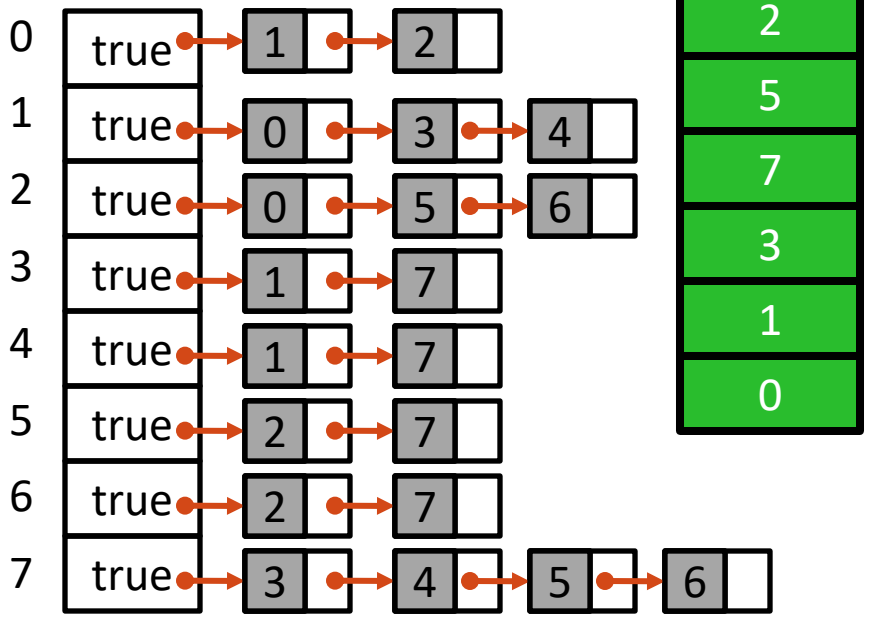
- We begin the search by visiting the start vertex, *v*.
  （我們從走訪起始點 *v* 開始搜尋的工作）
- Next, we select an unvisited vertex, *w*, from *v*'s adjacency lists and carry out a depth first search on *w*.
  （接著, 從 *v* 的相鄰串列中選取一個未經走訪的頂點 *w*, 由 *w* 繼續深度優先搜尋）
- We preserve our current position in *v*'s adjacency list by placing it on a **stack**.
  （將目前在 *v* 相鄰串列中的頂點放到堆疊中）
- Eventually our search reaches a vertex, *u*, that has no unvisited vertices on its adjacency list.
  （直到搜尋到達一個頂點 *u*, 且 *u* 的相鄰串列中沒有未經走訪的頂點為止）

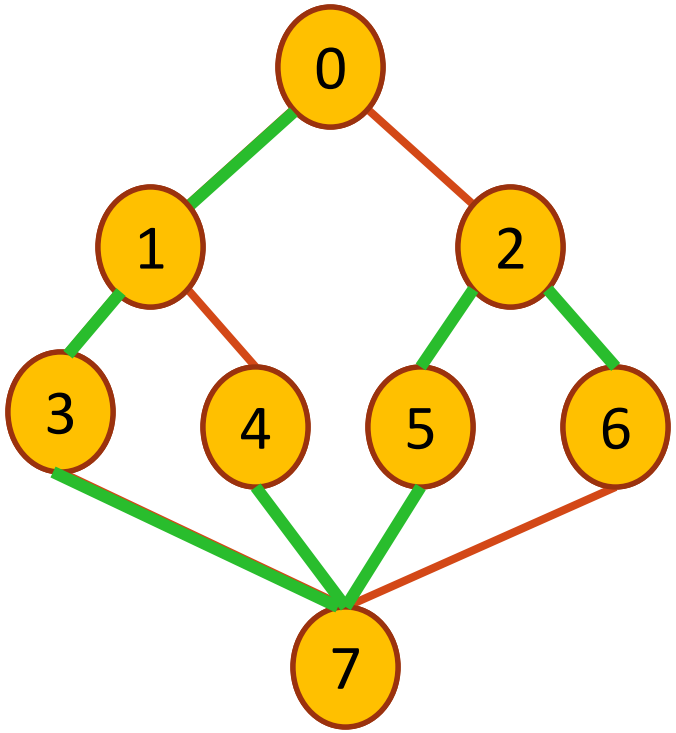
Depth First Search (DFS)$_{2/2}$

- At this point, we remove a vertex from **stack** and continue processing its adjacency list.
  (此時, 從堆疊中取出一個頂點, 並繼續處理其相鄰串列)
- Previously visited vertices are discarded; unvisited vertices are visited and placed on the stack.
  (**先前已走訪過的頂點可以省略不處理**; 未經走訪過的頂點則走訪它並放入堆疊)
- The search terminates when the stack is empty.
  (當堆疊變成空的時, 搜尋結束)

Depth First Search

- 類似樹的**前序走訪**
- 必須記錄之前的路徑
  - Stack
  - Recursive

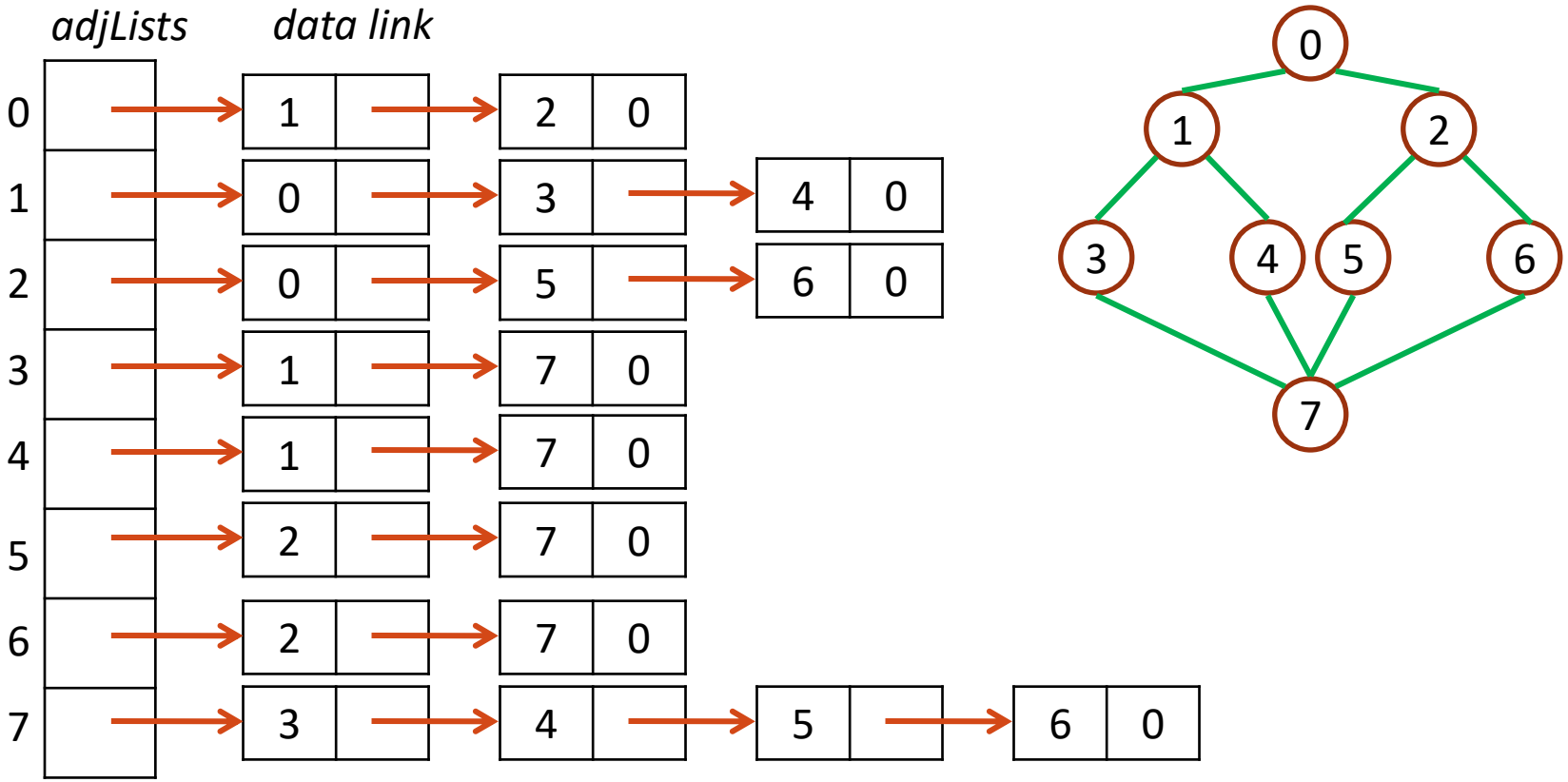




0→1→3→7→4→5→2→6

Depth First Search Order

- **The depth first search order: v_0, v_1, v_3, v_7, v_4, v_5, v_2, v_6.**

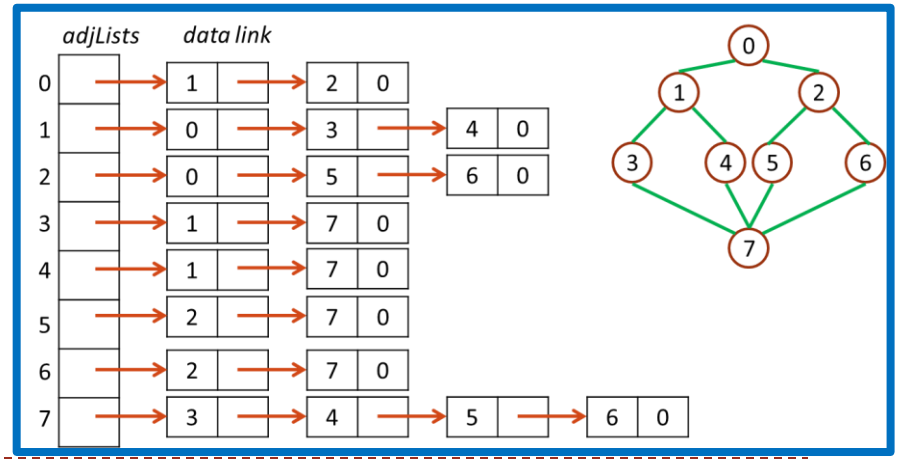


Program 6.1: Depth First Search

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

void dfs(int v)
{
    /* depth first search of a graph beginning with vertex v. */
    nodePointer w;
    visited[v] = true;
    printf("%5d",v);
    for(w = graph[v]; w; w = w->link)
        if (!visited[w->vertex])
                dfs(w->vertex);
}
```

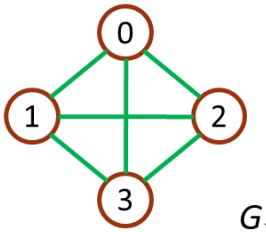
w不為NULL



Analysis of DFS

- If we represent *G* by its adjacency lists, then we can determine the vertices adjacent to *v* by following a chain of links.
  (如果 *G* 以相鄰串列表示, 則我們可以沿著串列指標, 找出與 *v* 相鄰的頂點)
- Since DFS examines each node in the adjacency lists at most once, the time to complete the search is $O(e)$. (e為總邊數)
  (因為 DFS 對相鄰串列中的各節點至多檢查一次, 完成搜尋所需的時間為$O(e)$)
- If we represent *G* by its adjacency matrix, then determining all vertices adjacent to *v* requires $O(n)$ time.
  (*G* 以相鄰矩陣表示, 則找出所有與 *v* 相鄰的頂點需要 $O(n)$)
  - Because we visit at most *n* vertices, the total time is $O(n^2)$.
    (因為我們最多走訪 *n* 個頂點, 所以全部的時間為$O(n^2)$)

Example.



G_1

0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

The adjacency matrix of G_1

Breadth First Search (BFS)$_{1/2}$

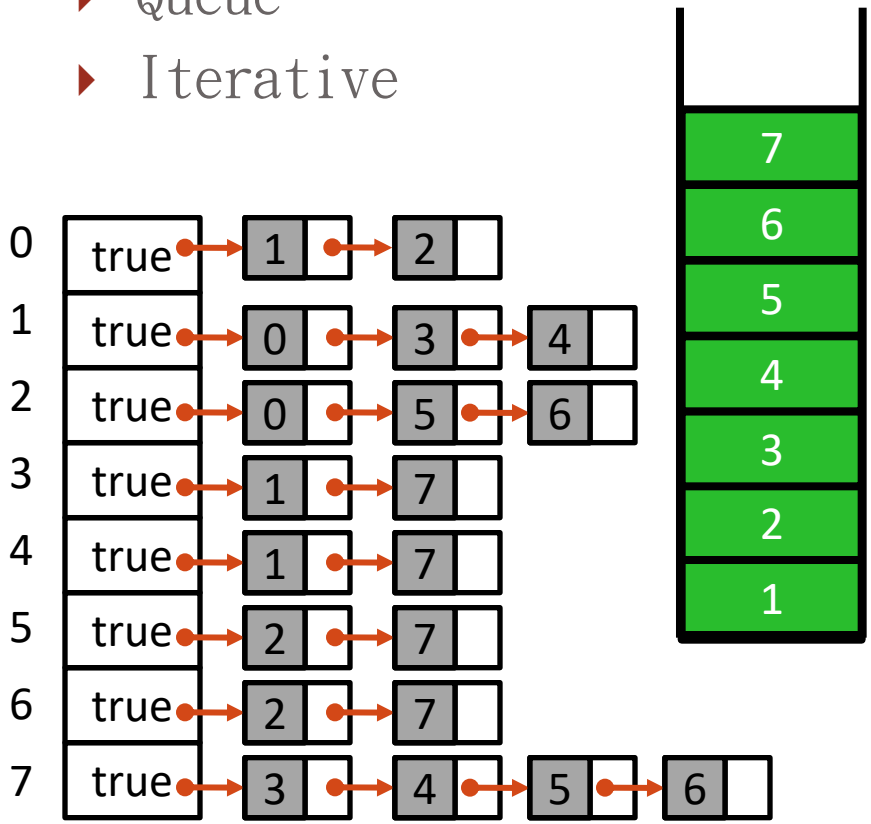
- Breadth First Search starts at vertex *v* and marks it as visited.
  (廣度優先搜尋由頂點 *v* 開始, 並標記它為已走訪)
- Then visiting each of the vertices on *v*'s adjacency list.
  (然後走訪與 *v* 相鄰的每一個頂點)
- When we have visited all the vertices on *v*'s adjacency list, we visit all the unvisited vertices that are adjacent to the first vertex on *v*'s adjacency list.
  (當與 *v* 相鄰的每一個頂點都已經走訪過了, 我們走訪與 *v* 的相鄰串列上的第一個頂點相鄰但未經走訪的頂點)
- To implement this scheme, as we visit each vertex we place the vertex in a **queue**.
  (要實作這個方法, 在走訪每一頂點時將它放到一個佇列中)

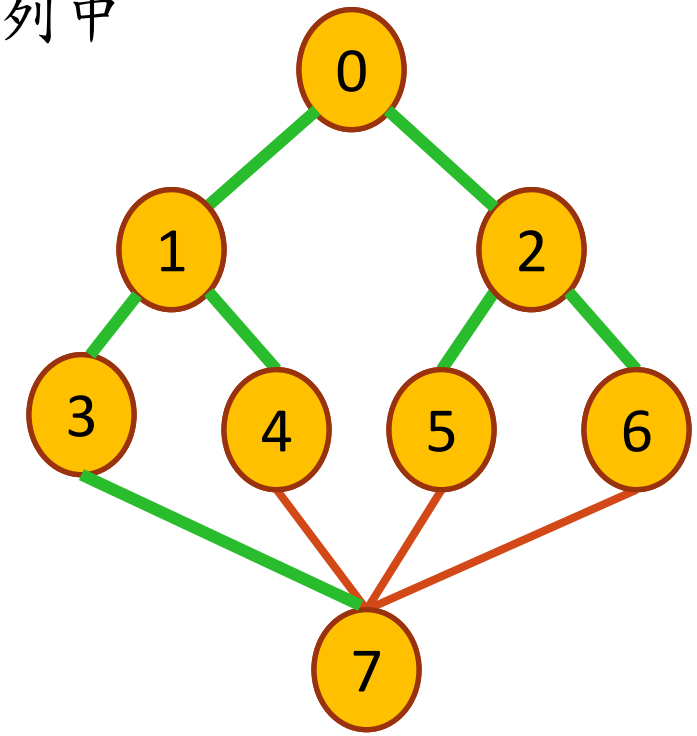
Breadth First Search (BFS)$_{2/2}$

- When we have exhausted an adjacency list, we remove a vertex from the queue and proceed by examining each of the vertices on its adjacency list.
  （當處理完一個相鄰串列時, 在從佇列中取出一個頂點, 並繼續處理其相鄰串列上的每一個頂點）
- **Unvisited vertices are visited and placed on the queue; visited are ignored.**
  （未經走訪的頂點被走訪後放入佇列中, 不需處理已經走訪的頂點）
- We have finished the search when the queue is empty.
  （當佇列變成空的時, 搜尋結束）

Breadth First Search

- 類似樹的**階序走訪**
- 未經走訪的頂點被走訪後放入佇列中
  - Queue
  - Iterative

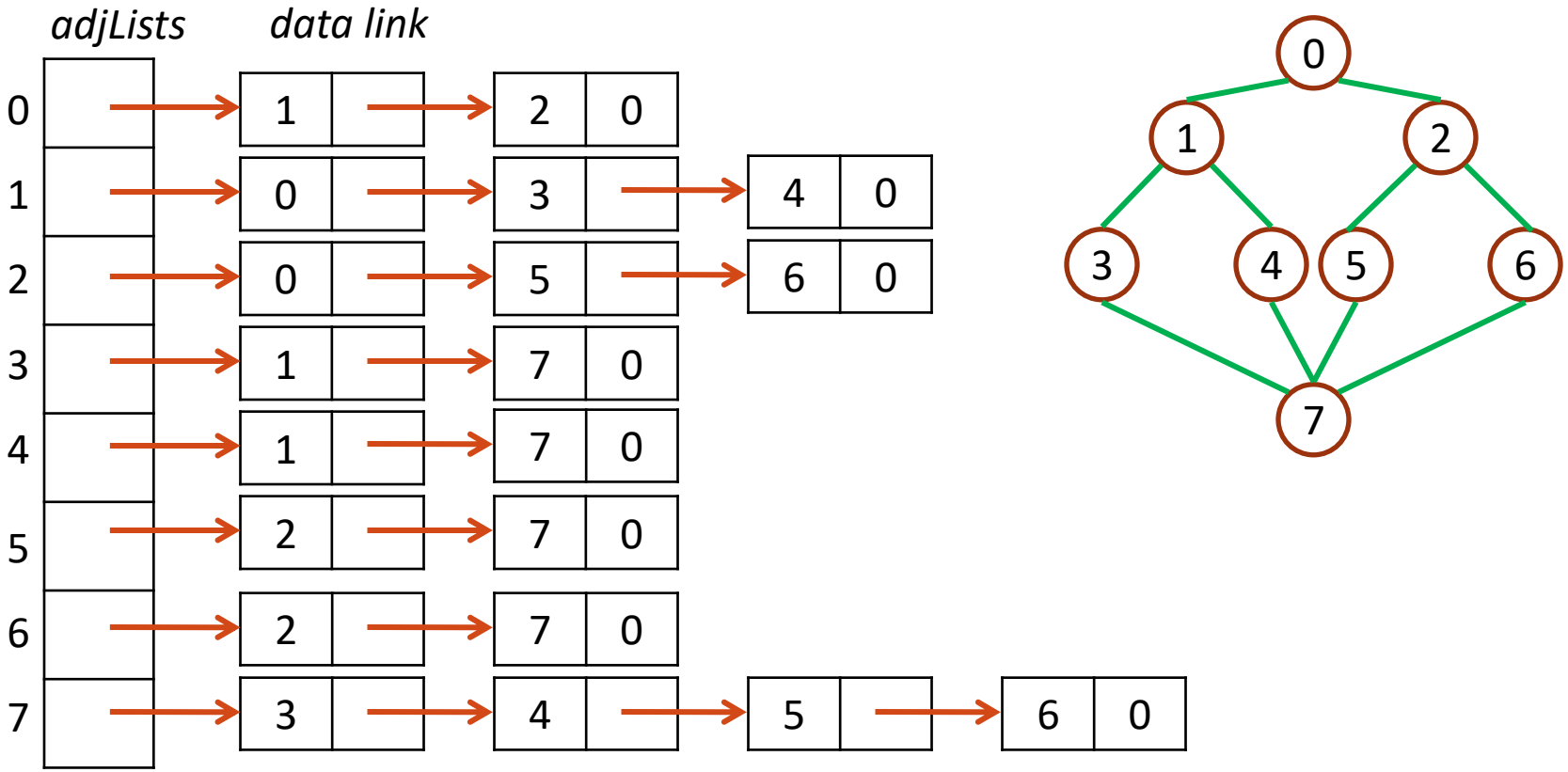




0→1→2→3→4→5→6→7

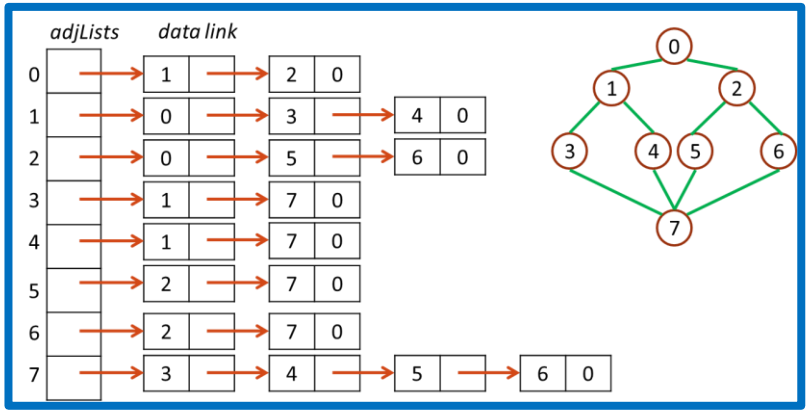
Breadth First Search Order

- **The breadth first search order: v0, v1, v2, v3, v4, v5, v6, v7.**



Program 6.2: Breadth First Search of a Graph

```
void bfs(int v)
{
    /* breadth first traversal of a graph, starting with node v
    the global array visited is initialized to 0, the queue
    operations are similar to those described in Chapter 4. */
    nodePointer w;
    front = rear = NULL;  /* initialize queue */
    printf("%5d",v);
    visited[v] = TRUE;
    addq(v);
    while(front) {
        v = deleteq();
        for(w = graph[v]; w ; w->link)
            if(!visited[w->vertex]) {
                printf("%5d", w->vertex);
                addq(w->vertex);
                visited[w->vertex]=TRUE;
            }
    }
}
```



Analysis of BFS

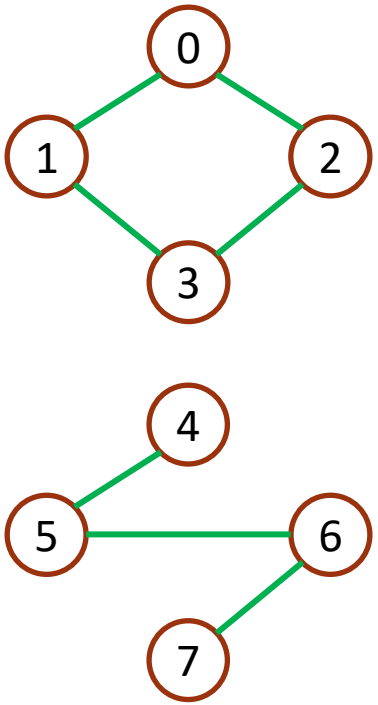
- Since each vertex is placed on the queue exactly once, the **while** loop is iterated at most n times.
  (因為每一頂點都被放入佇列中一次, while 迴圈最多執行 n 次)
- For the **adjacency list** representation, this loop has a total cost of $d_0 + \ldots + d_{n-1} = O(e)$, where $d_i = degree(v_i)$.
  (對於相鄰串列表示法而言, 迴圈的成本為 $d_0 + \ldots + d_{n-1} = O(e)$, 其中 $d_i = degree(v_i)$)
- For the **adjacency matrix** representation, the while loop takes O(n) time for each vertex visited.
  (對於相鄰矩陣表示法而言, 每走訪一個頂點while 迴圈需要 $O(n)$ 時間)
  - Therefore, the total time is $O(n^2)$.
    (因此, 全部的時間為$O(n^2)$)
- As was true of DFS, all vertices visited, together with all edges incident to them, from a connected of G. (正如 DFS 一般, 所有已走訪的頂點與所有附著於其上的邊, 形成圖形 G 上的連通元件)

Connected Components

- Let us look at the problem of determining whether or not an undirected graph is connected.
  (對於判斷一個無向圖形是否為連通的問題)
  - We can implement this operation by calling either *dfs* (0) or *bfs* (0) and then determining if there are any unvisited vertices.
    (**我們可以呼叫 *dfs* (0) 或 *bfs* (0) 然後判斷是否有未經走訪的頂點**)
- A related problem is that of listing the connected components of a graph.
  (另一個相關的問題為列出一個圖形的連通元件)
  - This can be done by making repeated calls to either *dfs* (*v*) or *bfs* (*v*) where *v* is an unvisited vertex.
    (只要藉由重覆呼叫 *dfs* (*v*) 或 *bfs* (*v*) 即可完成, 其中 *v* 是未經走訪的頂點)

Program 6.3: Connected Components

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



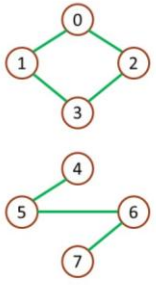
G_4

Analysis of Connected

- If *G* is represented by its adjacency lists, then the total time taken by DFS is *O*(*e*).
  (如果 *G* 以相鄰串列表示, 則 DFS 所需的全部時間為*O*(*e*))
- Since the **for** loop takes *O*(*n*) time, the total time needed to generate all the connected components is *O*(*n*+*e*).
  (因為 for 迴圈需要*O*(*n*) 時間, 要產生所有的連通元件所需的時間為*O*(*n*+*e*))
- If *G* is represented by its adjacency matrix, then the time needed to determine the connected components is *O*(n^2).
  (如果 *G* 以相鄰矩陣表示, 則決定連通元件所需的時間為*O*(n^2))

Program 6.3: Connected Components

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



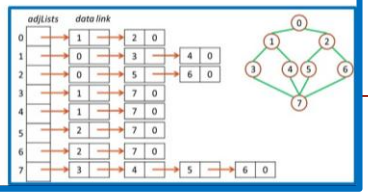
G_4

Program 6.1: Depth First Search

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

void dfs(int v)
{
   /* depth first search of a graph beginning with vertex v. */
   nodePointer w;
   visited[v] = true;
   printf("%5d",v);
   for(w = graph[v]; w; w = w->link)
      if (!visited[w->vertex])
            dfs(w->vertex);
}
```

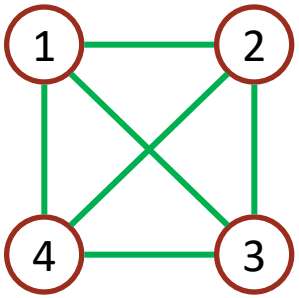
w不為NULL



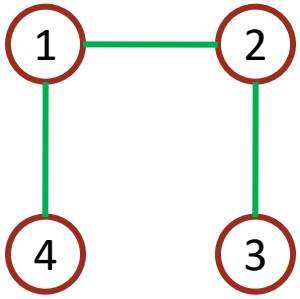
Spanning Trees

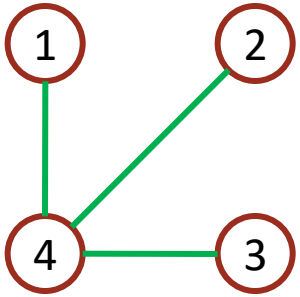
- A tree *T* is said to be a **spanning tree** of a connected graph *G* if *T* is a subgraph of *G* and *T* contains all vertices of *G*.
  (如果樹 *T* 是連通圖形 *G* 上的一個子圖且 *T* 包含圖形 *G* 上所有的頂點, 則我們稱 *T* 是圖形 *G* 的**生成樹**)
- When graph *G* is connected, a **DFS** or **BFS** implicitly partitions the edges in *G* into two sets:
  (當圖是連通時, 執行 DFS 或 BFS 時會將圖形上的邊分成兩組)
  - Tree edges: the set of edges used or traversed during the search.
    (樹邊：在搜尋時會用到或經過的邊集合)
  - Nontree edges : the set of remaining edges.
    (非樹邊：剩下的邊)

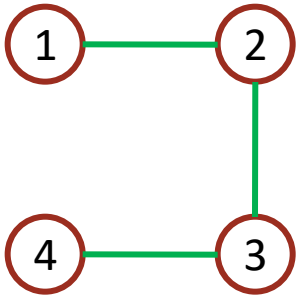
Example.
A complete Graph and Three of its Spanning Trees



G_1



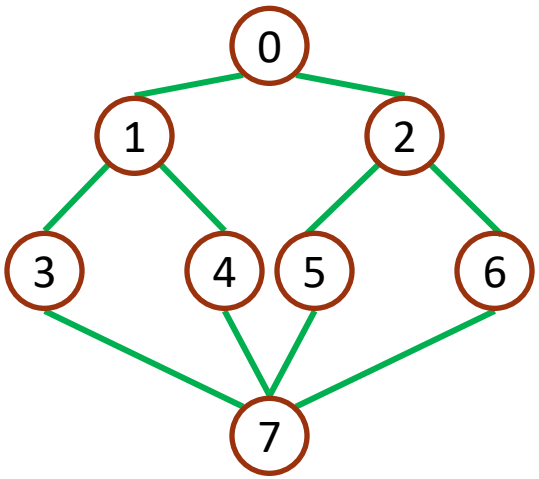


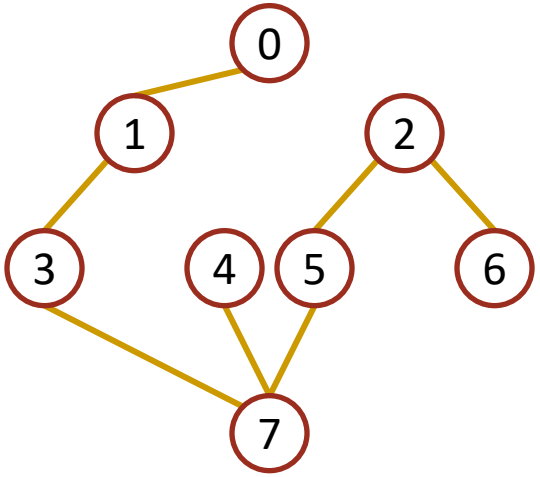


Three spanning trees of G_1.

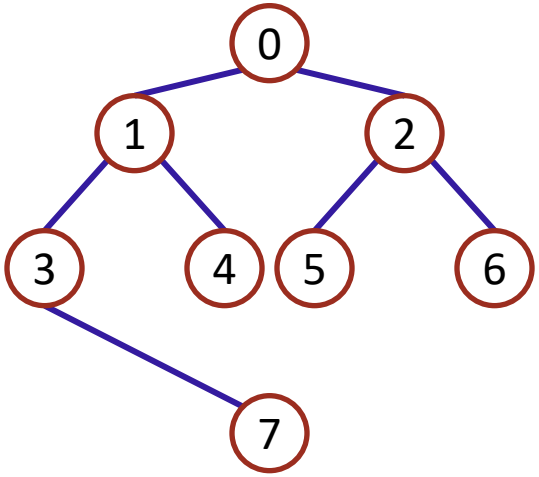
Depth-First and Breadth-First Spanning Trees

- 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**.





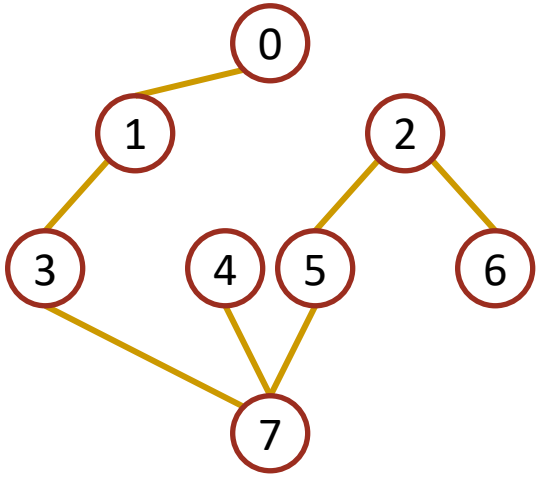
DFS (0)
spanning tree



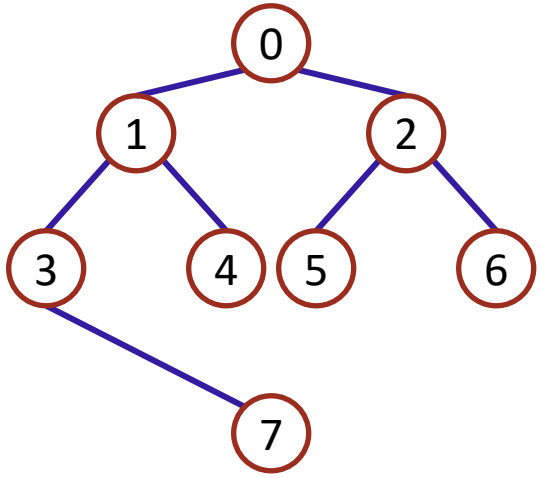
BFS (0)
spanning tree

Properties of spanning trees

- **Property 1:** Suppose we add a nontree edge, (*v*, *w*), into any spanning tree, *T*. The result is a cycle that consists of the edge (*v*, *w*) and all the edges on the path from *w* to *v* in *T*.
  (在任何生成樹 *T* 中加入一個非樹邊 (*v*, *w*). 所得到的結果是一個迴路, 它包含了邊 (*v*, *w*), 以及 *T* 中由 *w* 到 *v* 的路徑上所有的邊)



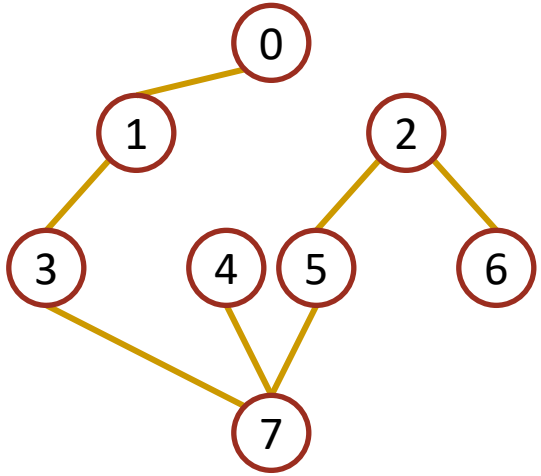
DFS (0)
spanning tree



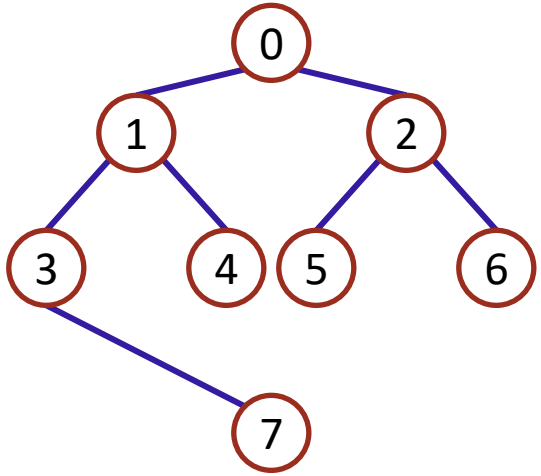
BFS (0)
spanning tree

Properties of spanning trees

- **Property 2:** A spanning tree is a minimal subgraph, G', of G such that $V(G') = V(G)$ and G' is connected.
  （一個生成樹是一個圖形 G 的最小子圖, G', 使的 $V(G') = V(G)$ 且 G' 是連通的）
  - We define a minimal subgraph as one with the fewest number of edge.（最小子圖是具有最少的**邊數**之子圖）
  - **A spanning tree has *n*-1 edges**.（生成樹有*n*-1個邊）



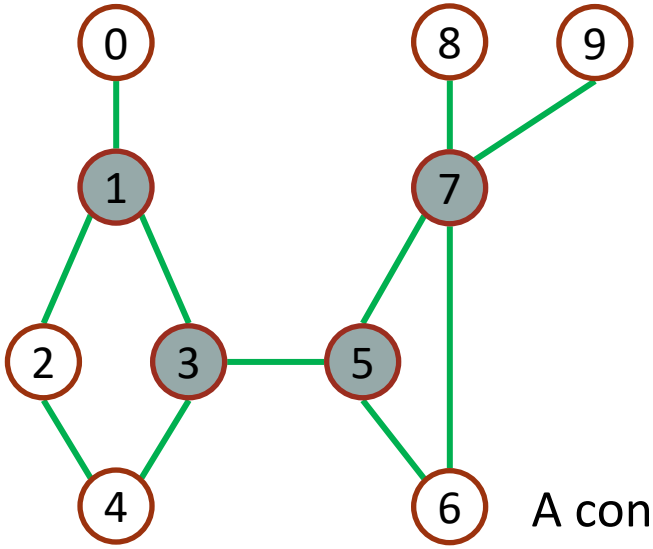
DFS (0)
spanning tree



BFS (0)
spanning tree

Articulation Points and Biconnected Graph

- An **articulation point** is a vertex *v* of *G* such that the deletion of *v*, together with all edges incident on *v*, produces a graph, *G'*, that has at least 2 connected components.
  (如果將 *v* 以及所有附著於 *v* 的邊刪除, **會產生兩個以上的連通元件**, 我們就稱 *v* 為 **連接點**)
  - Fig 6.19 has four articulation points, vertices 1, 3, 5, and 7.
    (圖 6.19 有 4 個連接點, 頂點 1, 3, 5, 以及 7)

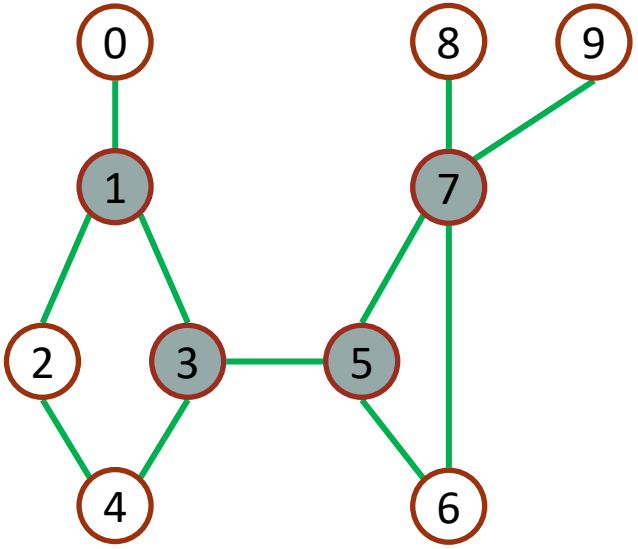


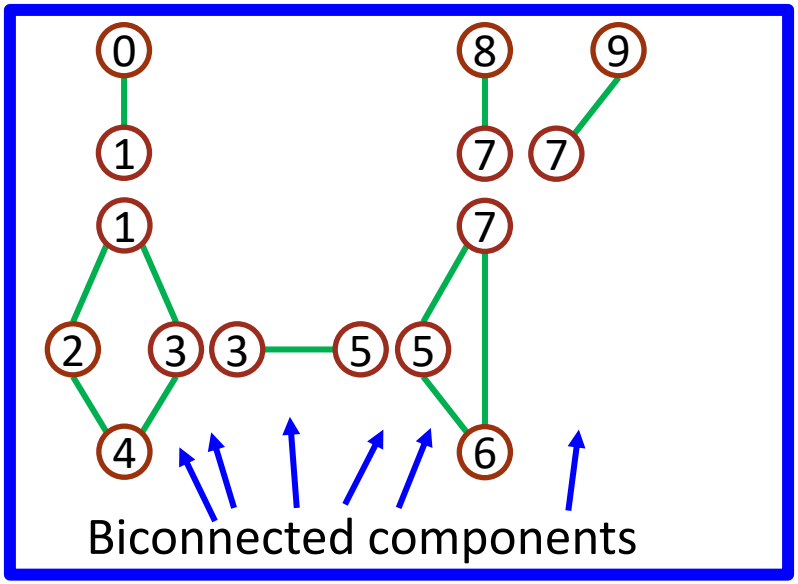
A connected graph

How to find articulation points?

Articulation Points and Biconnected Graph

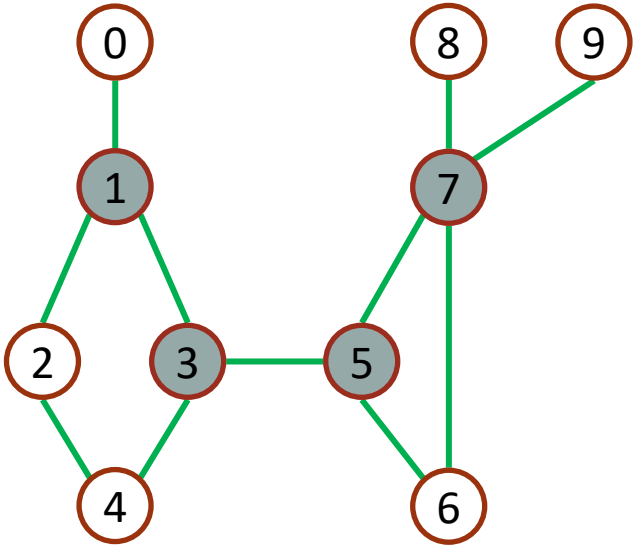
- A **biconnected graph** is a connected graph that has no articulation points.
  （二連通圖是一個沒有連接點的連通圖形）
- A **biconnected component** of a connected graph *G* is a **maximal biconnected subgraph** *H* of *G*.
  （一個連通圖形 ***G*** 的二連通元件 ***H*** 是一個最大的二連通子圖）
  - By maximal, we mean that *G* contains no other subgraph that is both biconnected and properly contains *H*.

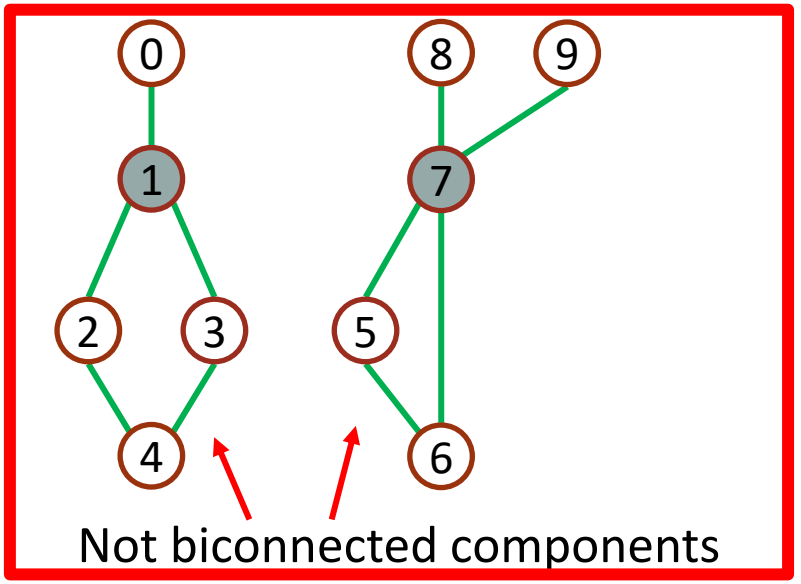




Articulation Points and Biconnected Graph

- A **biconnected graph** is a connected graph that has no articulation points.
  （二連通圖是一個沒有連接點的連通圖形）
- A **biconnected component** of a connected graph *G* is a **maximal biconnected subgraph** *H* of *G*.
  （一個連通圖形 ***G*** 的二連通元件 ***H*** 是一個最大的二連通子圖）
  - By maximal, we mean that *G* contains no other subgraph that is both biconnected and properly contains *H*.

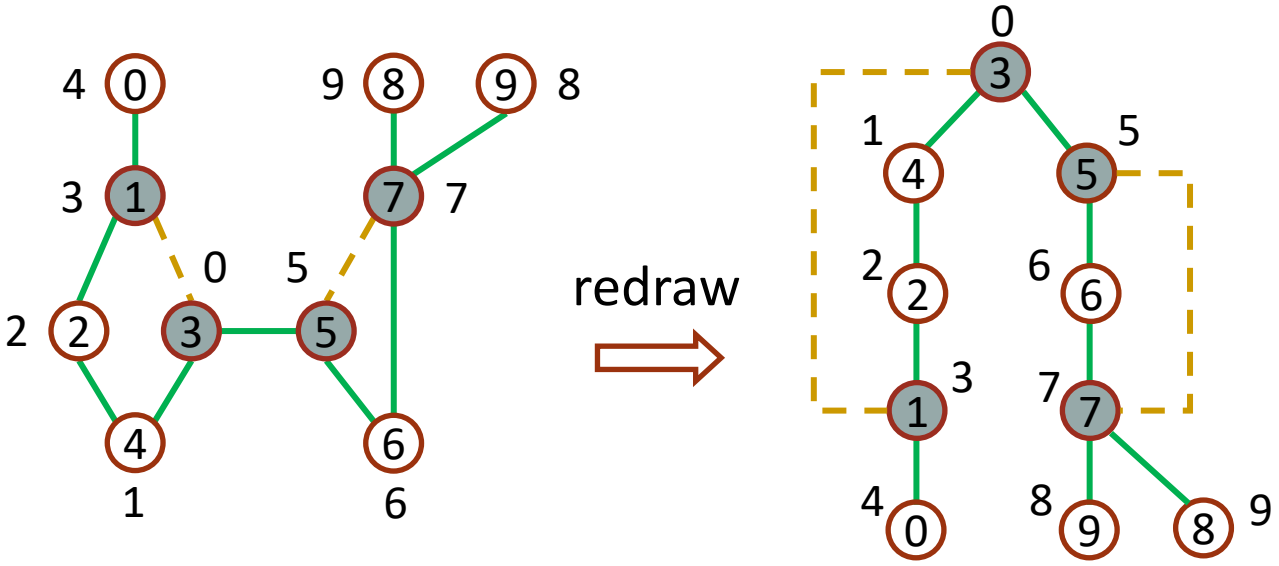




Find the articulation points 1/3

- We can find the biconnected components of *G* by using any depth first spanning tree of *G*.
  (藉由深度優先樹, 我們可以找出圖形 *G* 的二連通元件)

- The broken lines in Fig 6.20(b) represent nontree edges. We now call these nontree edges the **back edges**.
  (圖6.20(b)的虛線代表非樹邊, **我們現在先稱這些非樹邊為後退邊**)

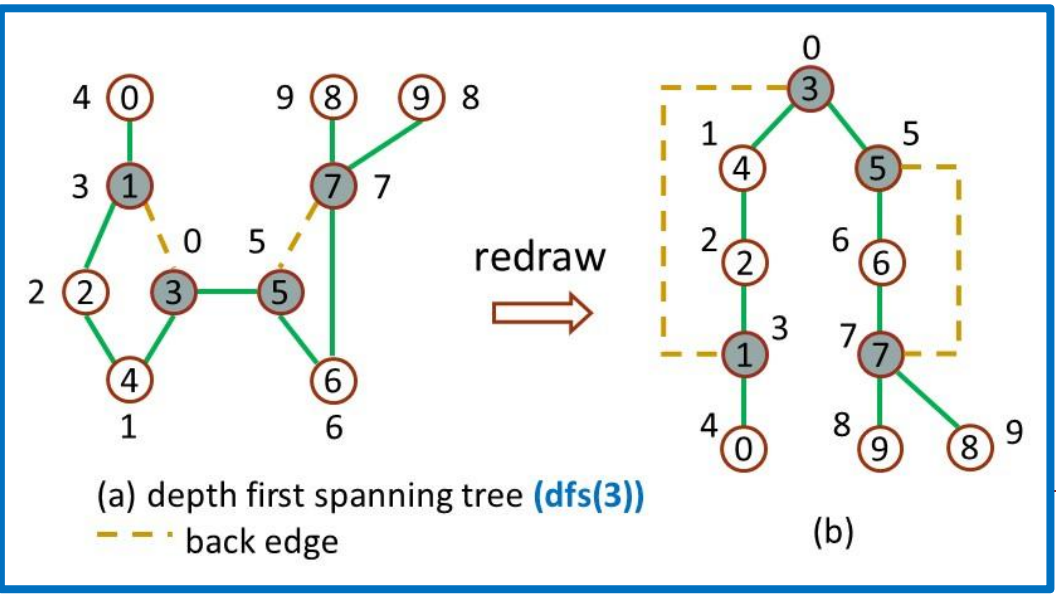


(a) depth first spanning tree **(dfs(3))**
back edge

(b)

(Two Observations)

- The root of a depth first spanning tree is an articulation point **iff** it has at least two children.
  (深度優先樹的根節點為一個連接點若且唯若它至少有兩個子節點)
- 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 that consists of only *w*, descendants of *w*, and a single back edge.
  (任一其他頂點 *u* 為一個連接點, 若且唯若它至少有一個子頂點 *w*, 使得我們無法僅用 *w*, *w* 的後代和單一個後退邊而到達 *u* 的某個祖先)



頂點 *5* 是一個連接點：存在子頂點*6*, 我們無法僅用頂點*6*, 頂點*6* 的後代和單一個後退邊到達頂點 *5* 的祖先。

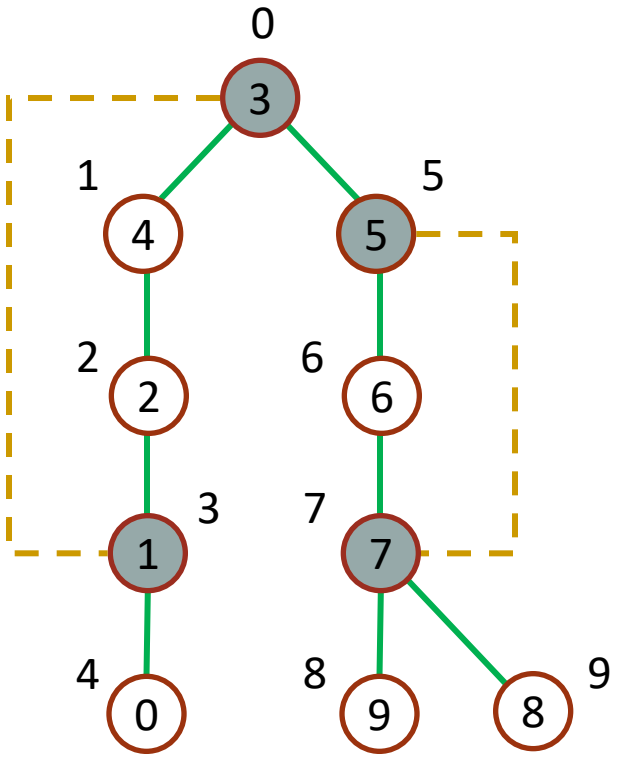
頂點 *6*不是一個連接點：所有頂點6的子頂點(只有頂點*7*)， 我們可以用頂點*7*, 頂點*7* 的後代和單一個後退邊到達頂點 *5*(頂點6的某個祖先)。

Find the articulation points 2/3

- The **depth first numbers**, or **dfn**, of the vertices give the sequence in which the vertices are visited during the depth first search.
  (頂點的**深度優先序號**為在頂點在深度優先搜尋時, 頂點被走訪的順序)
  - If *u* is an ancestor of *v* in the depth first spanning tree, then *dfn*(*u*) < *dfn*(*v*).
    (如果在深度優先樹中*u* 是 *v* 的祖先, 則 *dfn*(*u*) < *dfn*(*v*))
- The **low (*u*)** value of vertex *u* is the lowest depth first number that we can reach from *u* using a path of descendants followed by at most 1 back edge:
  (*low*(*u*)是利用 *u* 的後代以及至多一個後退邊**可達到的頂點**之最小深度優先序號)

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

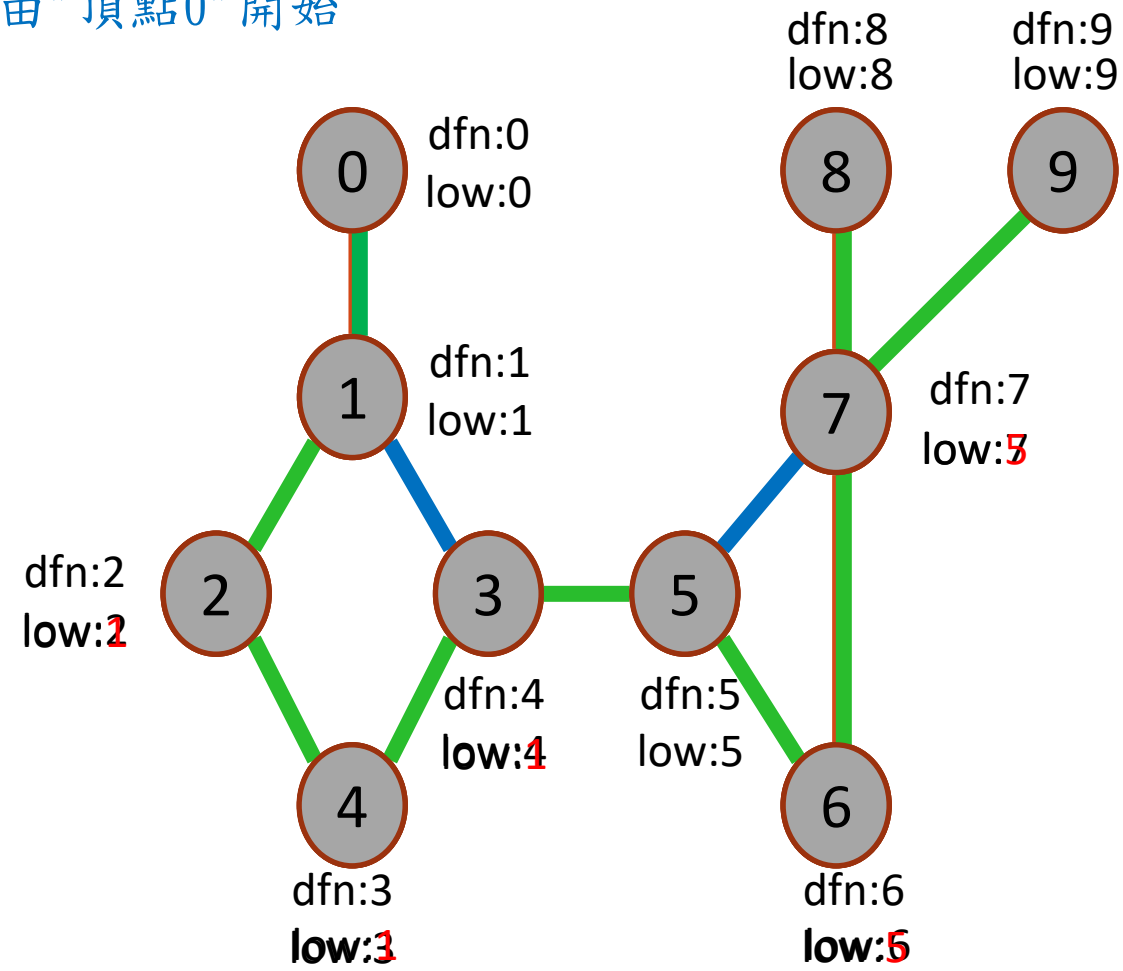
Figure 6.21:
dfn and *low* values for *dfs* spanning tree with root=3



Vertex	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

Determining *dfn* and *low*

假設這次由"頂點0"開始



Program 6.4: Determining *dfn* and *low*

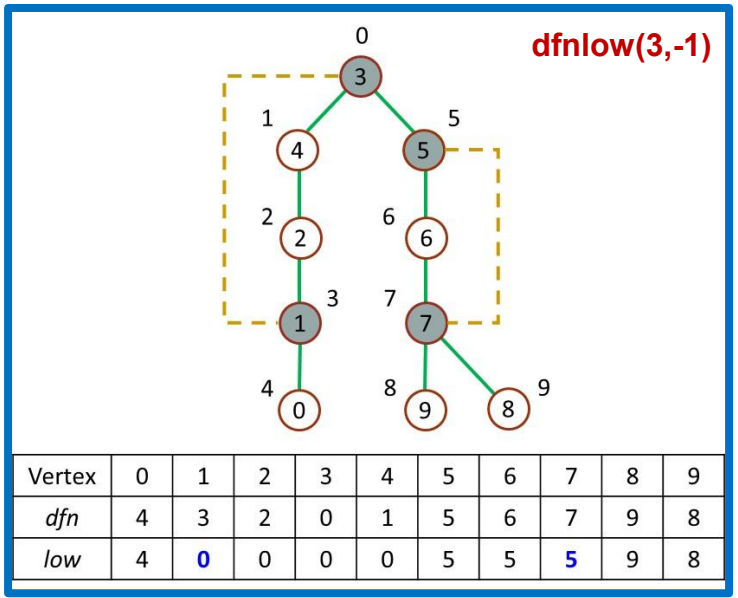
```
void init(void)
{
        int i;
        for(i = 0; i < n; i++) {
                visited[i] = FALSE;
                dfn[i] = low[i] = -1;
        }
        num = 0;
}
```

-
```
#define MIN2 (x, y) ((x) < (y) ? (x) : (y))
 short int dfn [MAX_VERTICES];
 short int low[MAX_VERTICES];
 int num = 0;

void dfnlow(int u, int v)
{
   /* 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++;
      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]);
          }
          else if (w != v)
              low[u] = MIN2(low[u], dfn[w] );
      }
 }
```

用法：呼叫 dfnlow(x, -1)



Vertex	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

- Time Complexity: $O(e)$

Find the articulation points 3/3

- We say that *u* is an articulation point **iff**
  (*u* 是一個**連接點**若且唯若)
  - *u* is the root of the spanning tree and has two or more children
    (*u* 是一個根節點且至少有兩個子節點)
  - or *u* is not the root of the spanning tree and *u* has a child *w* such that $low(w) \geq dfn(u)$
    (或 *u* 不是一個根節點但有一個子節點 *w* 使得 $low(w) \geq dfn(u)$)

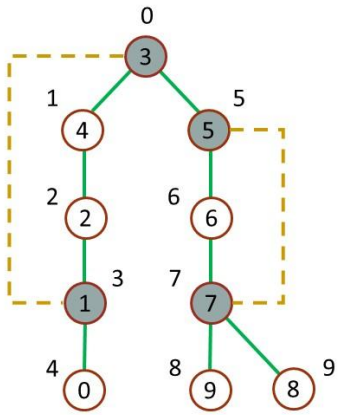
等同於

"它有一個子頂點 w，使得我們無法僅用 w, w 的後代和單一個後退邊而到達 u 的某個祖先"

- For example,
  - **vertex 1**: $low(0) = \mathbf{4} \geq \mathbf{3} = dfn(1)$
  - **vertex 7**: $low(8) = \mathbf{9} \geq \mathbf{7} = dfn(7)$
  - **vertex 5**: $low(6) = \mathbf{5} \geq \mathbf{5} = dfn(5)$
  - **vertex 3**: it has two children

  連接點
  - **vertex 2**: $low(1) = \mathbf{0} < \mathbf{2} = dfn(2)$
  - **vertex 6**: $low(7) = \mathbf{5} < \mathbf{6} = dfn(6)$

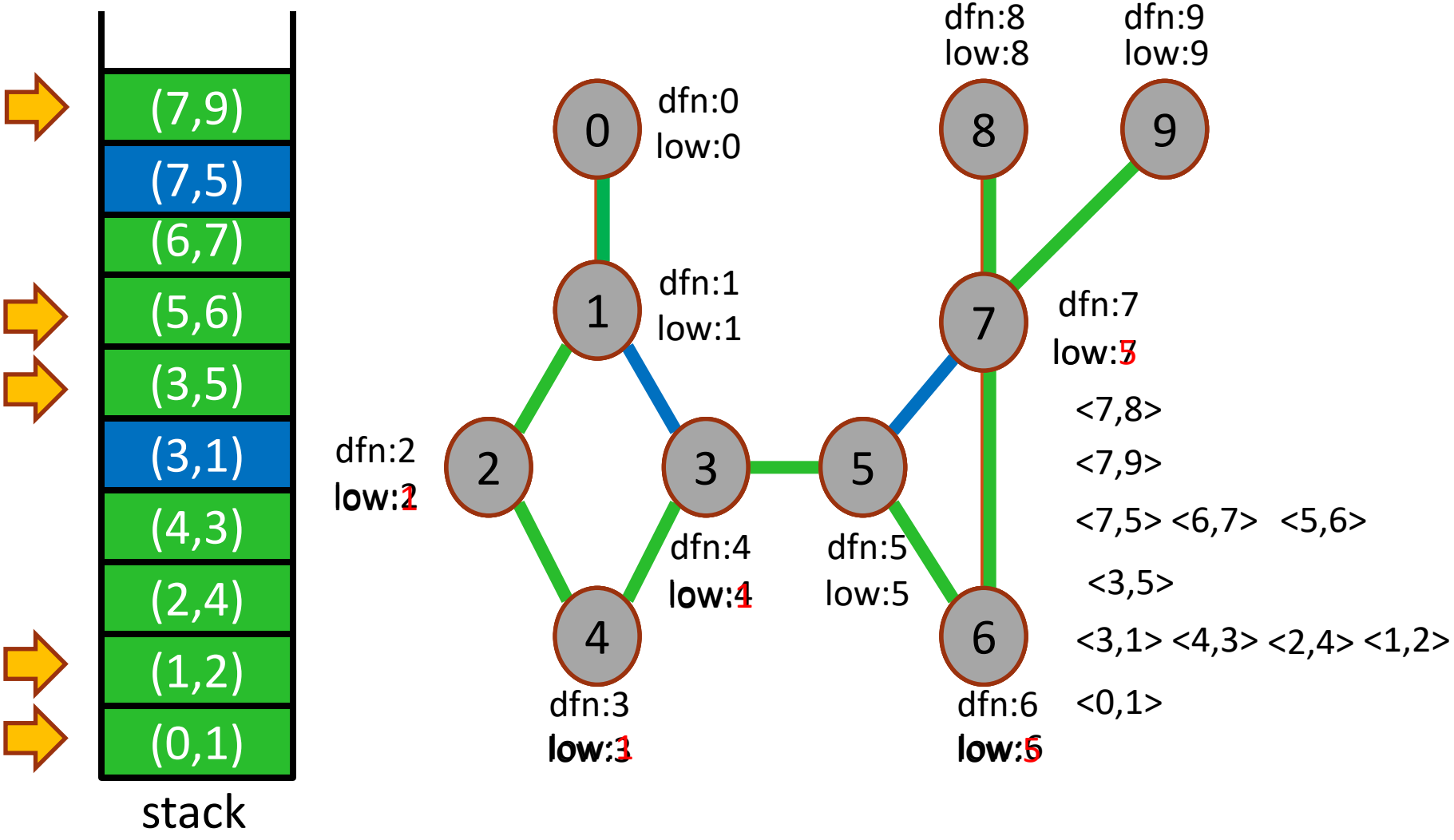
  不是連接點



Vertex	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

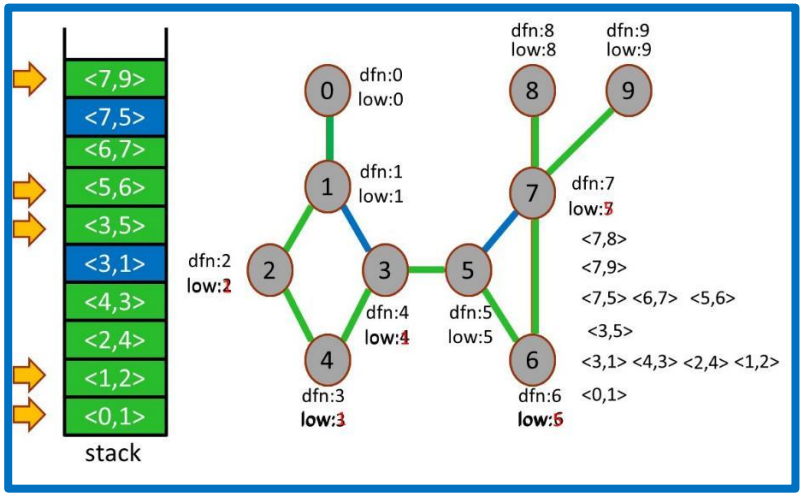
從dfnlow(u,w)返回時，若low[w]≥dfn[u]，則辨認出一個新的biconnected component

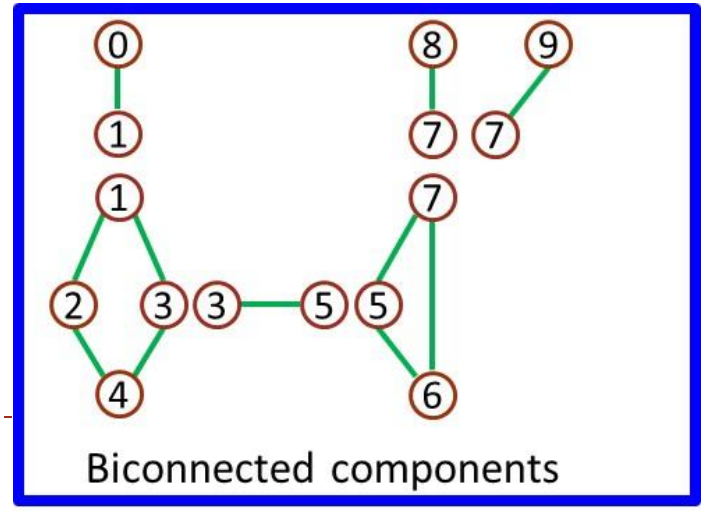
Biconnected Graph演算法



Program 6.6: Biconnected components of a graph

```
Void bicon(int u, int v)
{   /* 計算dfn與low並且依據G的雙連通元件輸出邊。 v是所產生的生成樹裡u的父節點（如果v存在的話）。
       我們假設dfn[]初始值為-1，num初始值為0， s是一個初值為空的堆疊。 */
    nodePointer 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); /* 把邊加進堆疊 */
            if (dfn[w] < 0) { /* w沒被走訪過 */
                bicon(w, u);
                low[u] = MIN2(low[u],low[w]);
                if (low[w] >= dfn[u]) {
                    printf("New biconnected component: ");
                    do { /* 從堆疊裡刪除一個邊 */
                        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]);
    }
}
```





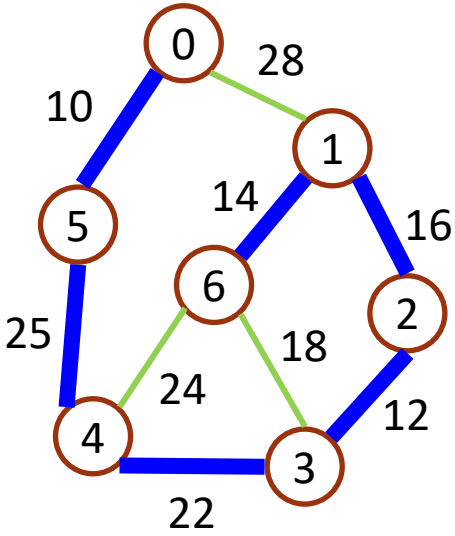
- Time Complexity: $O(n + e)$

Outline

- The Graph Abstract Data Type（圖形抽象資料型態）
- Elementary Graph Operations（基本的圖形運算）
- **Minimum Cost Spanning Trees（最小成本生成樹）**
- Shortest Paths（最短路徑）

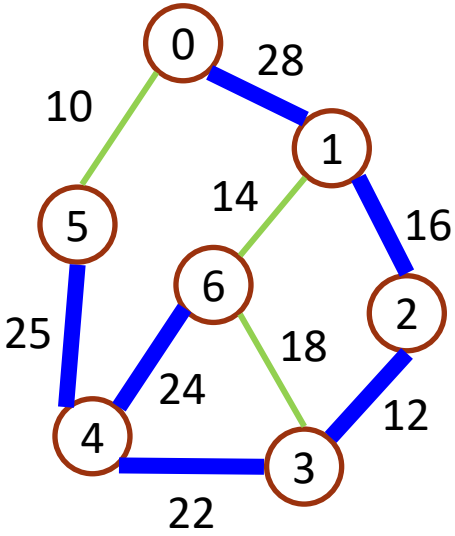
Minimal Cost Spanning Tree 1/2

- The **cost** of a spanning tree of a weighted undirected graph is the sum of the weights of the edges in the spanning tree.
  （加權的無向圖形的生成樹成本為各邊的權重之和）
- A **minimum-cost spanning tree** is a spanning tree of least cost.
  （最小成本生成樹是具有最少成本的生成樹）



(a) minimum-cost spanning tree.

cost=10+25+22+12+16+14=99



(b) spanning tree.

cost=28+16+12+22+24+25=127

Minimal Cost Spanning Tree 2/2

- Three different algorithms can be used to obtain a minimum cost spanning tree of a connected undirected graph, and all three use **greedy method**:
  (有三種不同的演算法可用來找出連通無向圖形最小成本生成樹, 這三種都採用稱為**貪婪法則**的演算法設計策略)
  1. Kruskal's algorithm
  2. Prim's algorithm
  3. Sollin's algorithm

Greedy Method 1/2

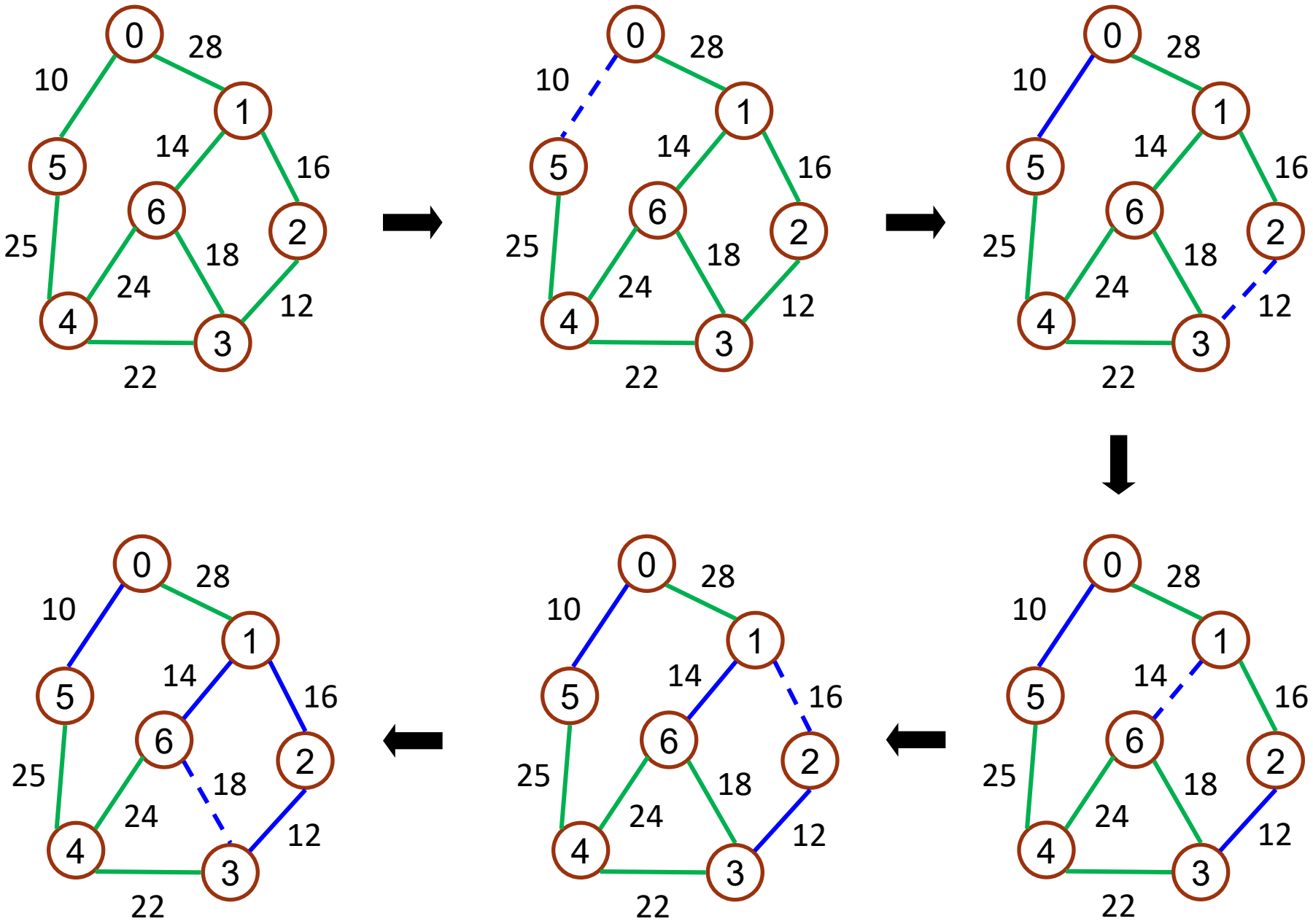
- In the greedy method, we construct an optimal solution in stages.
  (在貪婪法則中, 我們逐步建立最佳解法)
- A feasible solution is one which works within the constrains specified by the problem.
  (我們根據問題所規定的條件產生可行的解法)
- At each stage, we make a decision that is the best decision at that time.
  (**在每一步驟中, 我們找出現階段最佳的決策**)
- Typically, The selection of an item at each stage is based on either a least cost or a highest profit criterion.
  (**決策通常是根據最小成本或最大利益**)

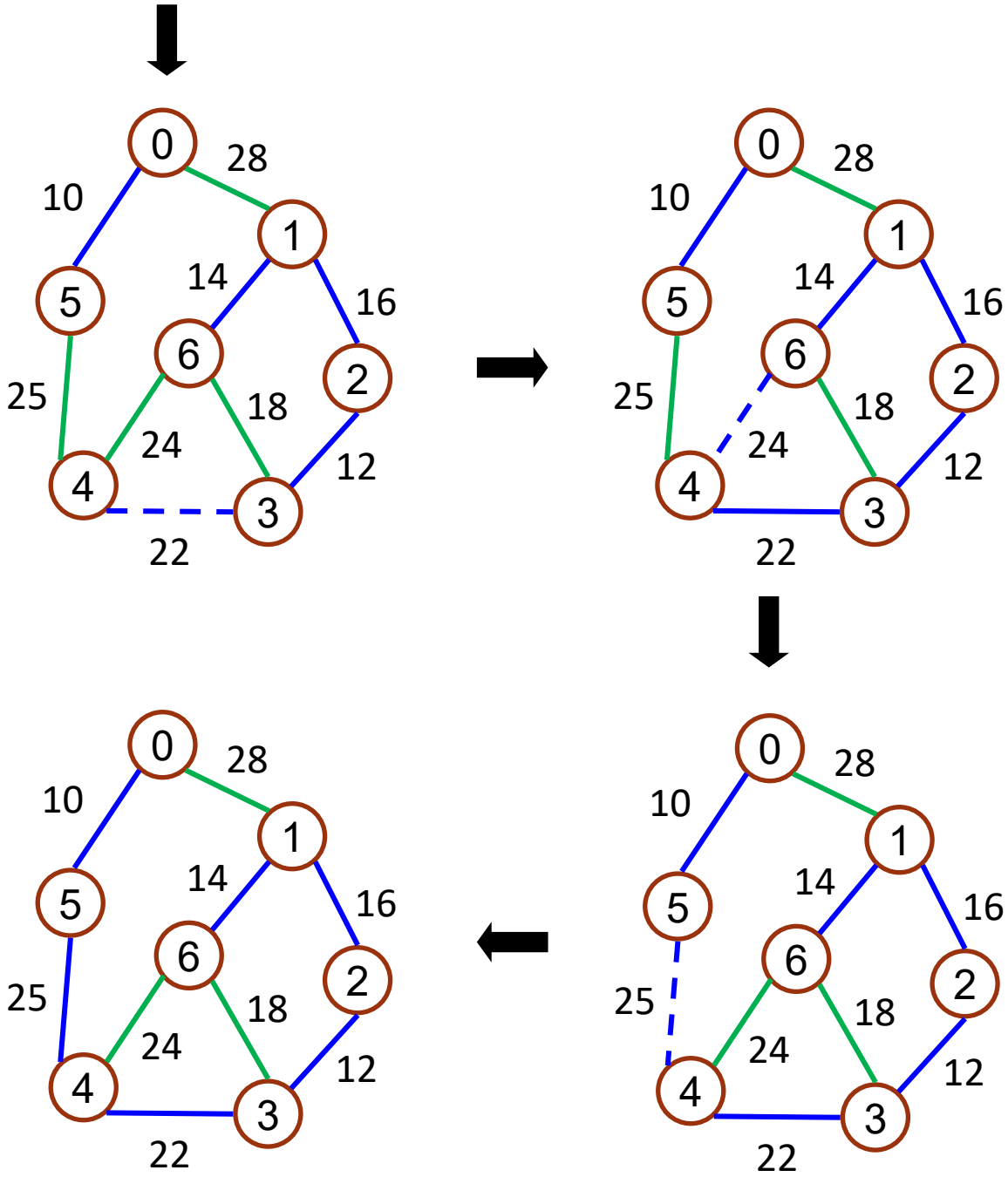
Greedy Method 2/2

- For minimum spanning trees, we use a least cost criterion. Our solution must satisfy the following constrains:
  (對於最小成本生成樹而言, 我們希望成本最小. 我們解答必須符合以下限制)
  - **We must use only edges within the graph**
    (我們只能使用圖形內的邊)
  - **We must exactly n-1 edges**
    (我們只能使用恰好 *n*-1 邊)
  - **We may not use edges that would produce a cycle**
    (我們不可使用會形成迴路的邊)

1. Kruskal's Algorithm 1/2

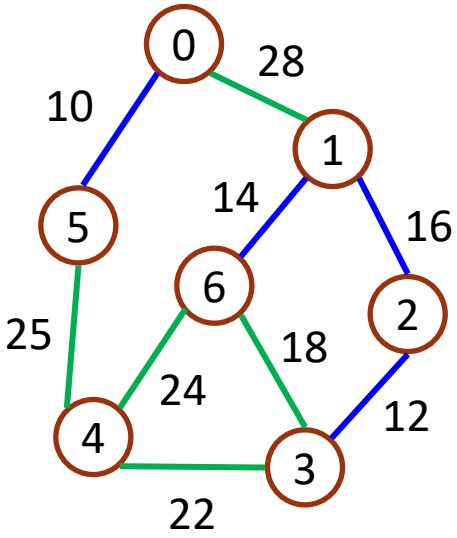
- Build a minimum cost spanning tree *T* by adding edges to *T* one at a time.
  (每次在 *T* 中加入一個邊, 以形成最小成本生成樹 *T*)
- Select the edges for inclusion in *T* in nondecreasing order of their cost.
  (根據各邊的成本, 以遞增的方式選擇要加入 *T* 的邊)
- An edge is added to *T* if it does not form a cycle with the edges that are already in *T*.
  (一個邊若與 *T* 原來已有的邊不會形成迴路, 即可加入 *T* 中)
- Exactly n-1 edges will be selected for inclusion in *T*.
  (恰有 *n-1* 個邊會被選入 *T* 中)
- **Time complexity: *O*(*e* log *e*)** (chap 7: sorting)


0
1
2
3
4
5
6
10
28
14
16
25
18
24
12
22


0
28
10
1
5
14
16
6
2
25
18
24
12
4
3
22

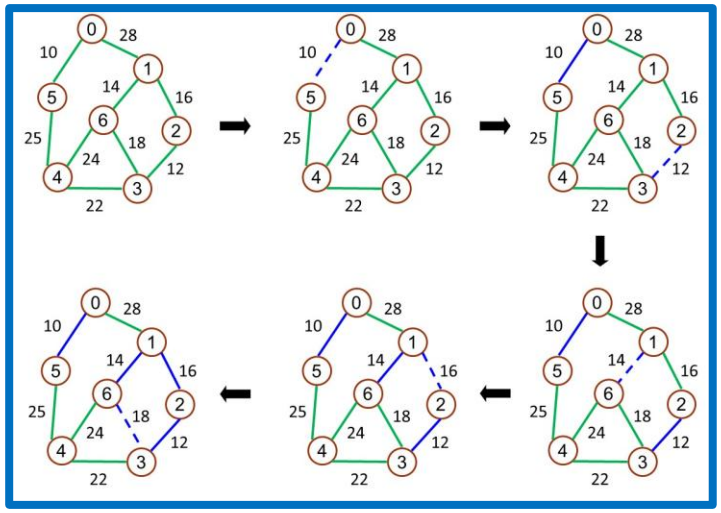
Union-find operations

- The union-find operations are used to determine whether or not adding an edge would cause a cycle.
  (chap 5: union and find operation)
- For example, the sets in the following would be {0, 5}, {1, 2, 3, 6}, {4}. (下圖的集合為 {0, 5}, {1, 2, 3, 6}, {4})
- Since vertices 3 and 6 are already in the same set, the edge (3, 6) are rejected. (**因為頂點 3 和頂點 6 在同一個集合,所以不加入邊 (3, 6)**)



1. Kruskal's Algorithm 2/2

```
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");
```



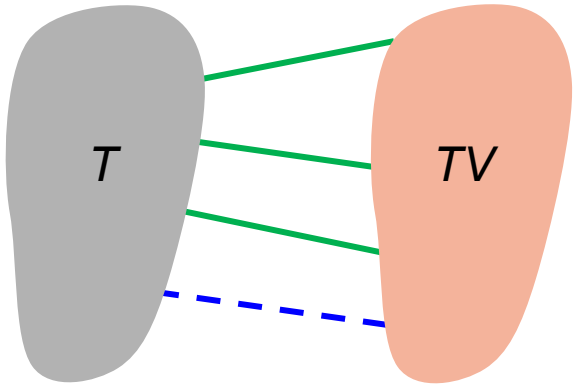
- As we shall see in Chapter 7, we can sort the edges in E in $O(e \log e)$ time.
  (在第 7 章有說明如何將圖形上的邊, 在$O(e \log e)$ 時間由小到大排列)
- Since the union-find operations require less time than choosing and deleting an edge, the total computing time is $O(e \log e)$.
  (因為 union-find所需時間較排序來的小, 所以時間複雜度為$O(e \log e)$)

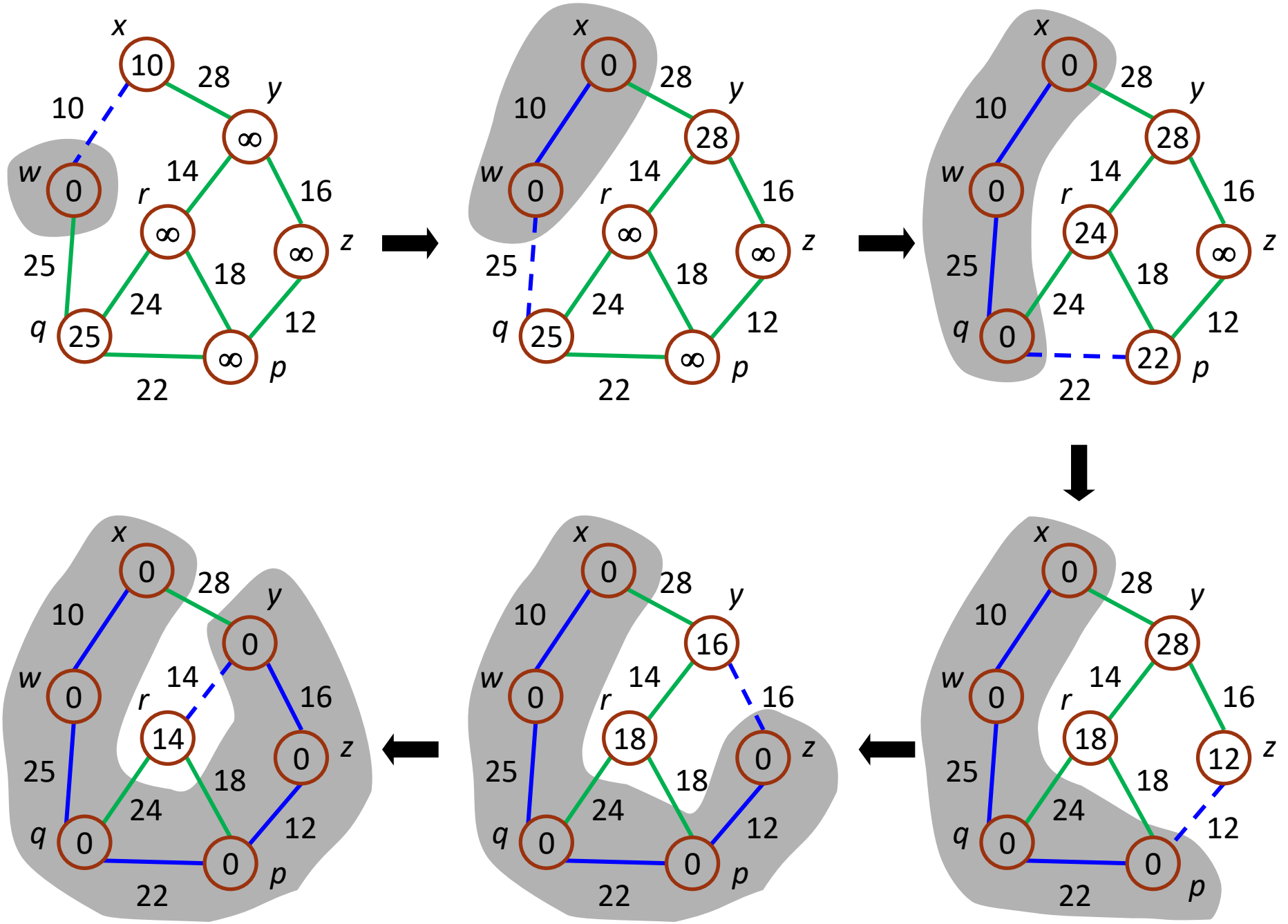
2. Prim's Algorithm 1/3

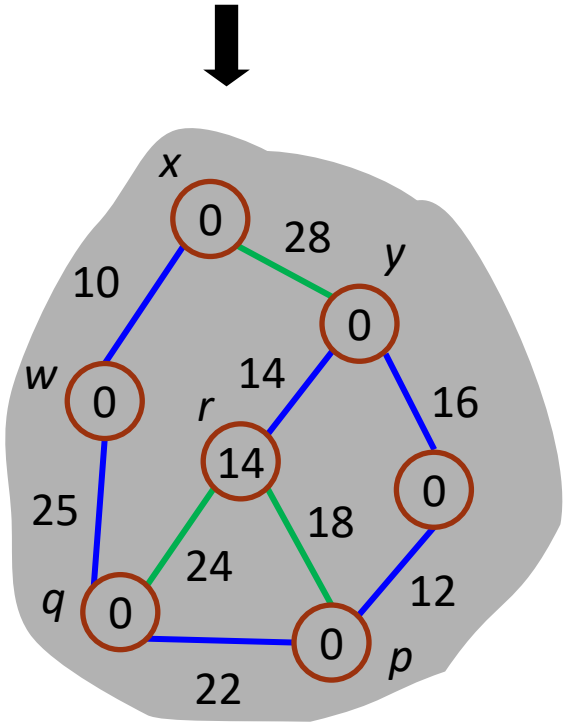
- Prim's algorithm, like Kruskal's, constructs the minimum-cost spanning tree one edge at a time.
  (**類似 Kruskal 演算法**, Prim 演算法以一次加入一個邊的方式建立最小成本生成樹)
- The main difference :
  (主要的差別在於)
  - The set of selected edges forms a tree at all times in Prim's algorithm.
    (**在 Prim 演算法**的每一步驟中, 所選擇的邊集合形成**一個樹狀結構**)
  - The set of selected edges in Kruskal's algorithm forms a forest at each stage.
    (**在 Kruskal 演算法**的每一步驟中, 所選擇的邊集合形成一個**樹林**)

2. Prim's Algorithm 2/3

- Prim's algorithm begins with a tree, *T*, that contains an arbitrary single vertex.
  (Prim 演算法由包含單一節點的樹 *T* 開始, **頂點可以是任意頂點**)
- Next, we add a least cost edge (u, v) to *T* such that $T \cup \{(u, v)\}$ is also a tree.
  (接著, 選擇滿足 $T \cup \{(u, v)\}$ **是樹狀結構的條件下具有最小成本的邊** (u, v) 加入 *T*)
- We repeat this edge addition until *T* contains *n*-1 edges.
  (重覆加邊的步驟**直到 *T* 有 *n-1* 個邊**)
- **Time complexity: $O(n^2)$.**

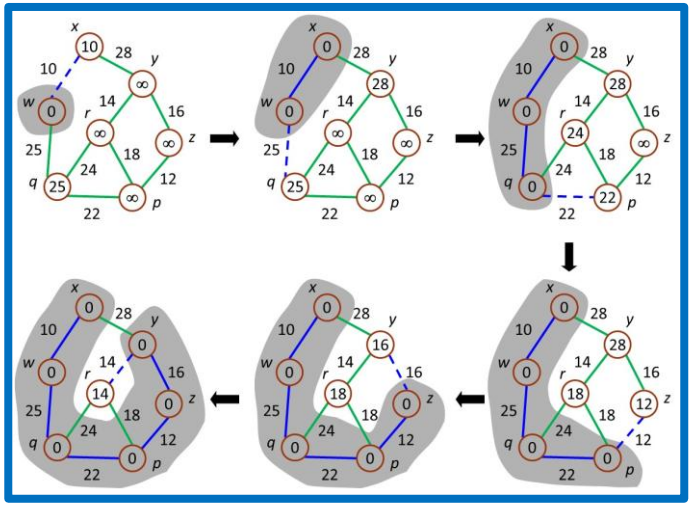



x
y
w
r
z
q
p
10
28
10
∞
14
16
0
∞
∞
25
18
24
12
25
∞
22
0
28
10
28
14
16
0
∞
∞
25
18
24
12
25
∞
22
0
28
10
28
14
24
16
0
∞
∞
25
18
24
12
0
22
22
0
28
10
16
14
16
0
18
0
25
18
24
12
0
0
22
0
28
10
28
14
18
16
0
18
0
25
18
24
12
0
0
22
0
28
10
0
14
16
0
14
0
25
18
24
12
0
0
22


x
0
28
y
10
0
w
0
14
r
16
14
0
25
18
24
12
q
0
0
p
22

2. Prim's Algorithm 3/3

```
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");
```



- Each vertex *v* that is not in TV has a companion vertex, *near*(v), such that *near*(*v*) ∈ TV and cost(*near*(*v*), *v*) is minimum over all such choices for *near*(*v*).
- Therefore, it takes $O(n)$ time to choose an edge.
- **We can implement Prim's algorithm in $O(n^2)$ time.**

3. Sollin's Algorithm 1/2

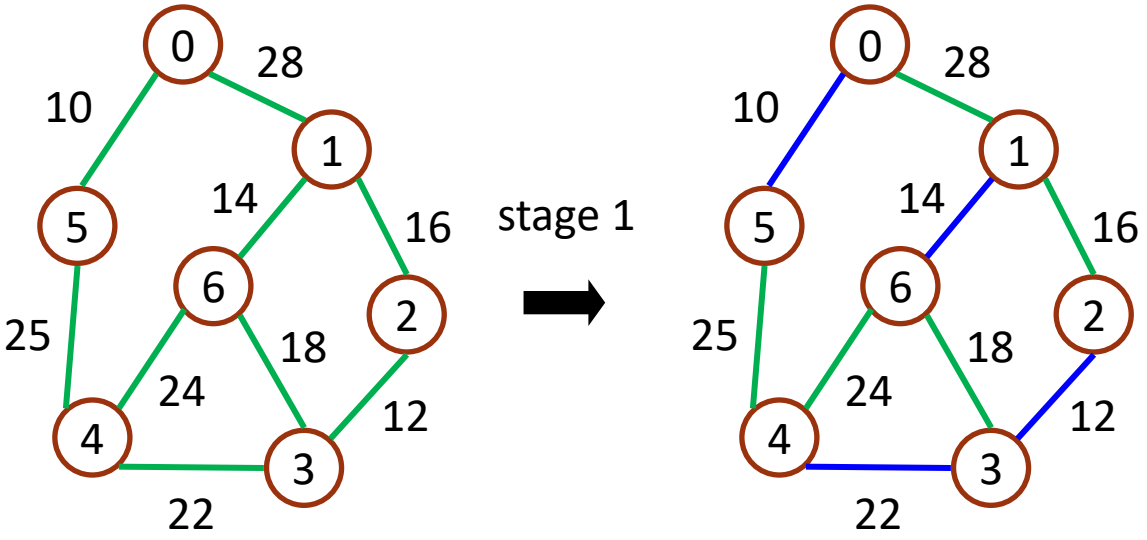
- Sollin's algorithm selects **several edges** for inclusion in *T* at each stage.
  (Sollin 演算法在每一步驟中**選擇數個可加入 *T* 的邊**)
- At the start of a stage, the selected edges, together with all the *n* vertices, form a spanning forest.
  (在每一步驟開始時, 所選擇的邊以及圖形上的 *n* 個頂點**形成一個生成樹林**)
- During a stage, we select one edge for each tree in the forest. This edge is a minimum cost edge that has exactly one vertex in the tree.
  (在每一步驟中, 我們**為每一顆樹選擇一個邊**, 這個邊是在恰好只有一個頂點在樹中的條件下, 成本最小的邊)

3. Sollin's Algorithm 2/2

- Since two trees in the forest could select the same edge, we need to eliminate multiple copies of edges.
  (因為樹林中的兩個樹可能選到相同的邊, **我們必須消除重覆的邊**)
- At the start of the first stage the set of selected edges is empty.
  (在第一步驟開始時, 並沒有任何的邊被選擇)
- The algorithm terminates when there is only one tree at the end of a stage or no edges remain for selection.
  (當**只有一個樹或沒有邊可以選擇的時候**, 停止演算法)
- **We can implement Prim's algorithm in *O*(e log *v*) time.**

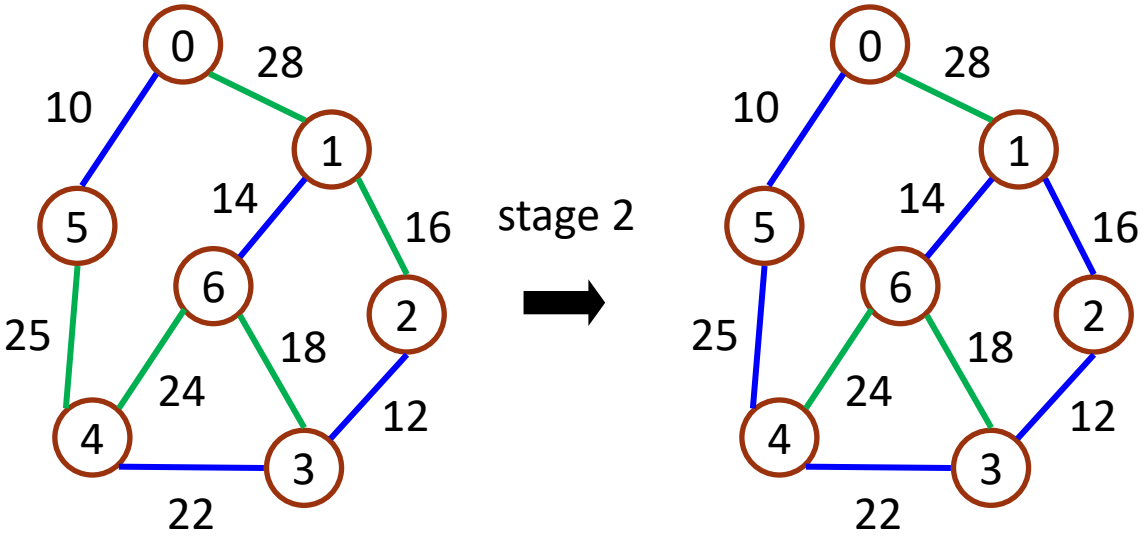
An example 1/2

- In stage 1, the selected edges are (0,5), (1, 6), (2, 3), (3, 2), (4, 3), (5, 0), and (6, 1).
- After eliminating the duplicate edges, we are left with edges (0,5), (1, 6), (2, 3), and (4, 3).



An example 2/2

- In stage 2, the selected edges are (5,4), (1, 2), and (2, 1).
- After eliminating the duplicate edges, we are left with edges (5, 4), and (2, 1).



Outline

- The Graph Abstract Data Type（圖形抽象資料型態）
- Elementary Graph Operations（基本的圖形運算）
- Minimum Cost Spanning Trees（最小成本生成樹）
- **Shortest Paths（最短路徑）**

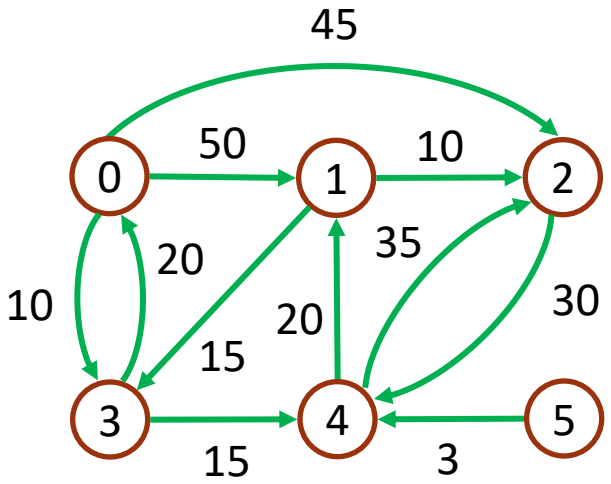
Shortest Paths

- How to find a shortest path from NTOU to Taipei 101 ?
  - Model this problem via a graph.
  - Vertices → important locations/bus or railway stations
  - Edges → highway/railway/roads
  - Each edge has a weight representing the distance between the two stations connected by the edge.
- There are two questions:
  - **Is there a path** from NTOU to 101 ?
  - If there is more than one path, **which path is the shortest** ?
- The starting vertex of the path will be referred to as the **source** and the last vertex the **destination**.
  (路徑開始的點稱為起點, 最後一個頂點稱為目的)

Single Source/All Destinations: Nonnegative Edge Costs

- We are given a directed graph, $G=(V, E)$, a weighting function, $w(e)$, $w(e)>0$, for the edges of G, and a source vertex v_0. If v_i and v_j are not adjacent, then $e(v_i, v_j) = \infty$.
  (給定一個有向圖形, $G=(V, E)$, 一個加權函數 $w(e)$, 圖形中的每一邊 $w(e) > 0$, 以及一個起始點 v_0 . **如果 v_i 與 v_j 不相連, 則 $e(v_i, v_j) = \infty$**)
- We wish to determine a shortest path from v_0 to each of the remaining vertices of G.
  (我們想找出從 v_0 出發到其餘各點的最短路徑)

	Path	***Length***
1	0, 3	10
2	0, 3, 4	25
3	0, 3, 4, 1	45
4	0, 2	45



Dijkstra's Algorithm

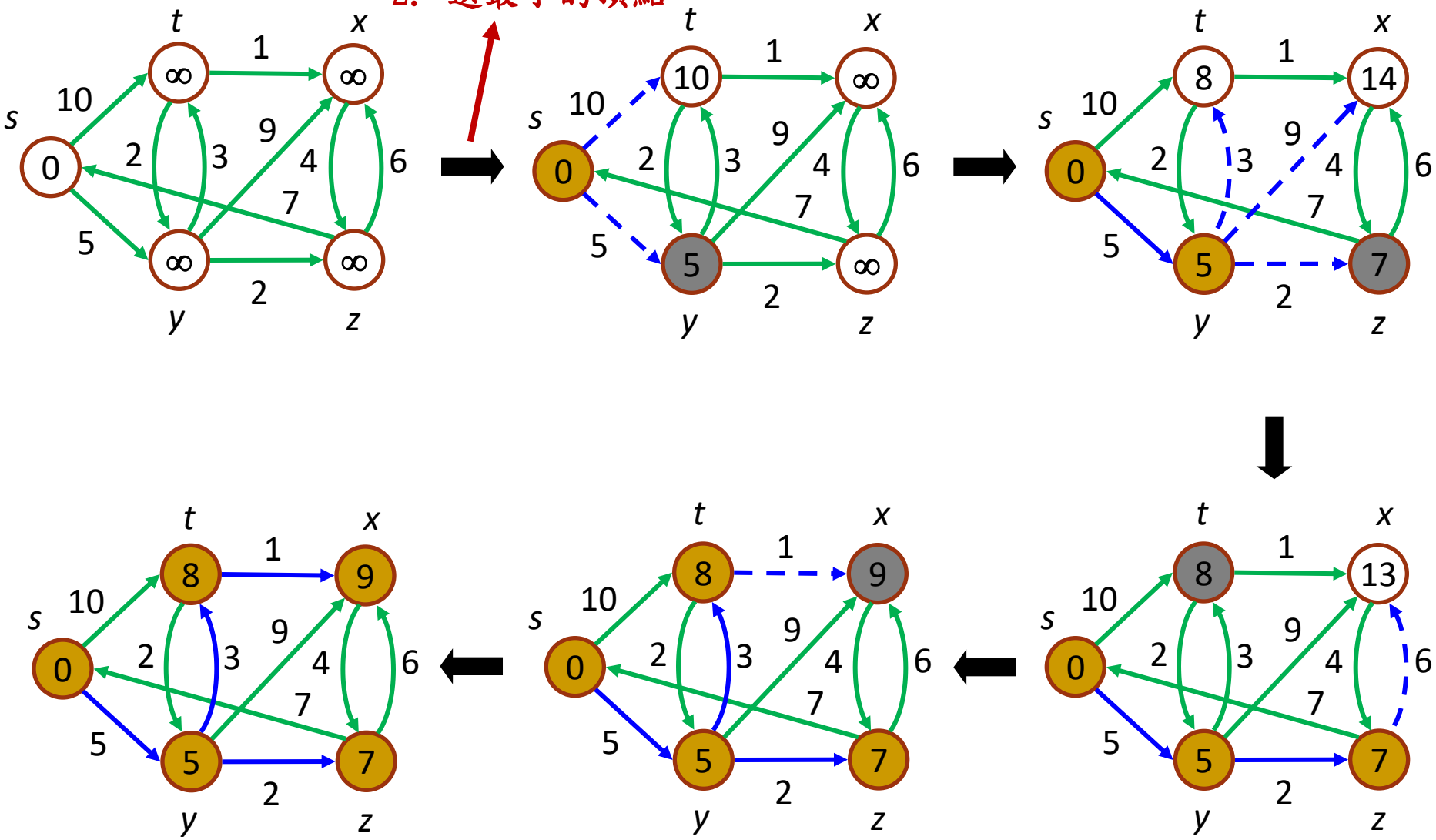
- We may use a greedy algorithm to generate the shortest paths in the order indicated in the above example.
  (我們可以使用貪婪演算法加上前一個例子所示的順序來產生最短路徑)
- Let *S* denote the set of vertices, including v_0, whose shortest paths have been found.
  (假設 *S* 代表包括 v_0 在內的頂點集合, 其最短路徑已經找出)
- For *v* not in *S*, let *dist*[*v*] be the length of the shortest path starting from v_0, going through vertices only in *S*, and ending in *v*.
  (對於不在 *S* 中的點 *v*, 讓 *dist*[*v*] 代表從 v_0 開始, 只能經過 *S* 中的頂點而到達 *v* 的路徑長度.)

Dijkstra's Algorithm

- At the first stage, we add v_0 to S, set $dist[v_0] = 0$ and determine $dist[v]$ for **each** v not in S.
  (一開始, 我們將 v_0 加入 S, 設定 $dist[v_0] = 0$, 對於不在 S 中的 v, 計算其$dist[v]$)
- Next, at each stage, vertex w is chosen so that it has the minimum distance, $dist[w]$, among all the vertices not in S.
  (接著, 在每一步驟中, 我們從不在 S 當中的頂點選擇 w, 使得它具有最小的 $dist[w]$)
- Adding w to S, and updating $dist[v]$ for v, that is not currently in S.
  (將 w 加入 S 當中, 對於目前不在 S 當中的點 v, 重新計算其 $dist[v]$)
- We repeat this vertex addition until $S = V(G)$.
  (執行加點的步驟直到 $S = V(G)$)
- **Time complexity: $O(n^2)$.**

每回合都做兩件事：

1. 更新剩餘的頂點值
2. 選最小的頂點



Dijkstra's Algorithm

- $S \leftarrow \{v_0\}$;

  $dist[v_0] \leftarrow 0$;

  for each v in V - $\{v_0\}$ do dist[v] $\leftarrow$ e(v_0, v);

  while $S \neq V$ do

  choose a vertex w in V-S such that $dist$[w] is a minimum;

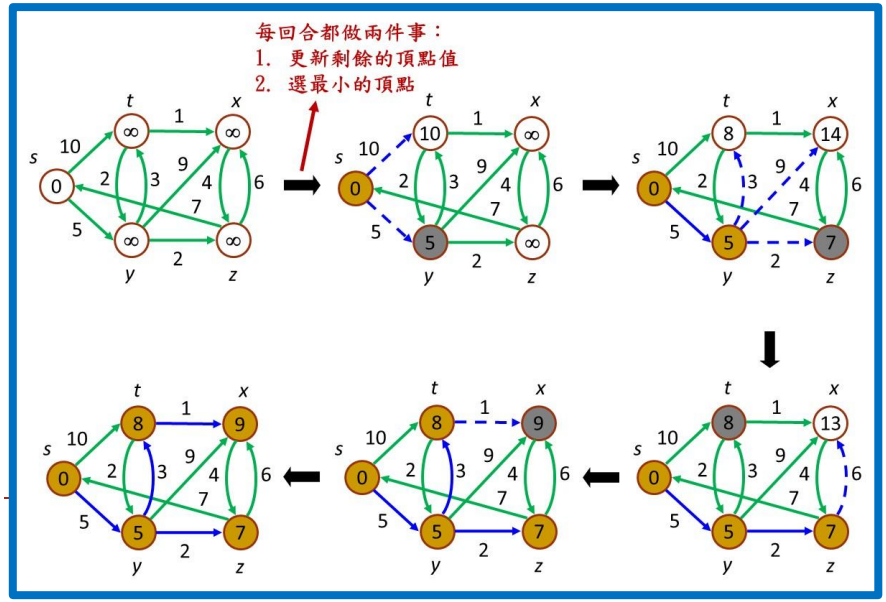
  add w to S;

  for each v in V-S do

  $dist[v] \leftarrow$ min($dist[v]$, $dist[w]$+e(w, v));

  endfor

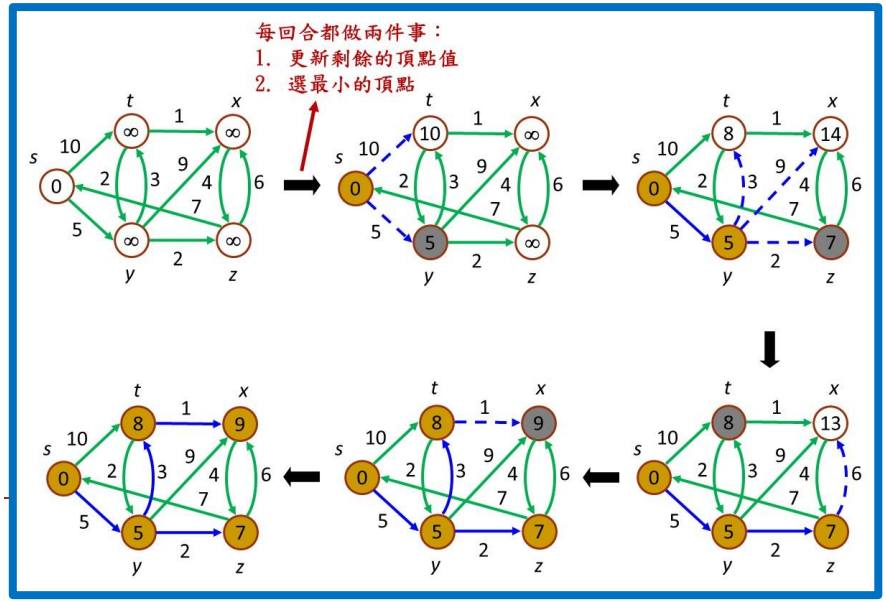
  endwhile



Program 6.9 & 10:
Single Source Shortest Paths – Dijkstra Algorithm 1/2

- **void shortestpath (int v, int cost[][MAX_VERTICES], int distance [], int n, short int found [])**

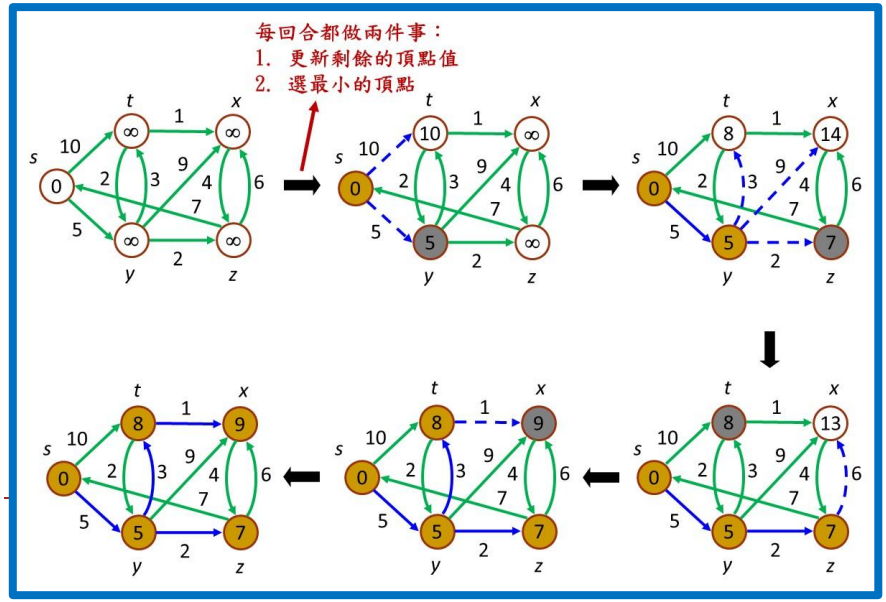
```
{
    /* distance [i] represents the shortest path from vertex v
     to l, found[i] holds a 0 if the shortest path from vertex l
     has not been found and a 1 if it has, cost is the adjacency
     matrix */
   int i, u, w;
   for (i=0; i<n; i++) {
        found [i] =FALSE;
        distance [i] =cost[v][i];
   }
   found[v] =TRUE;
   distance[v] = 0;
   for (i=0; i<n-1; i++) {
        u= choose (distance, n, found);
        found[u] =TRUE;
```

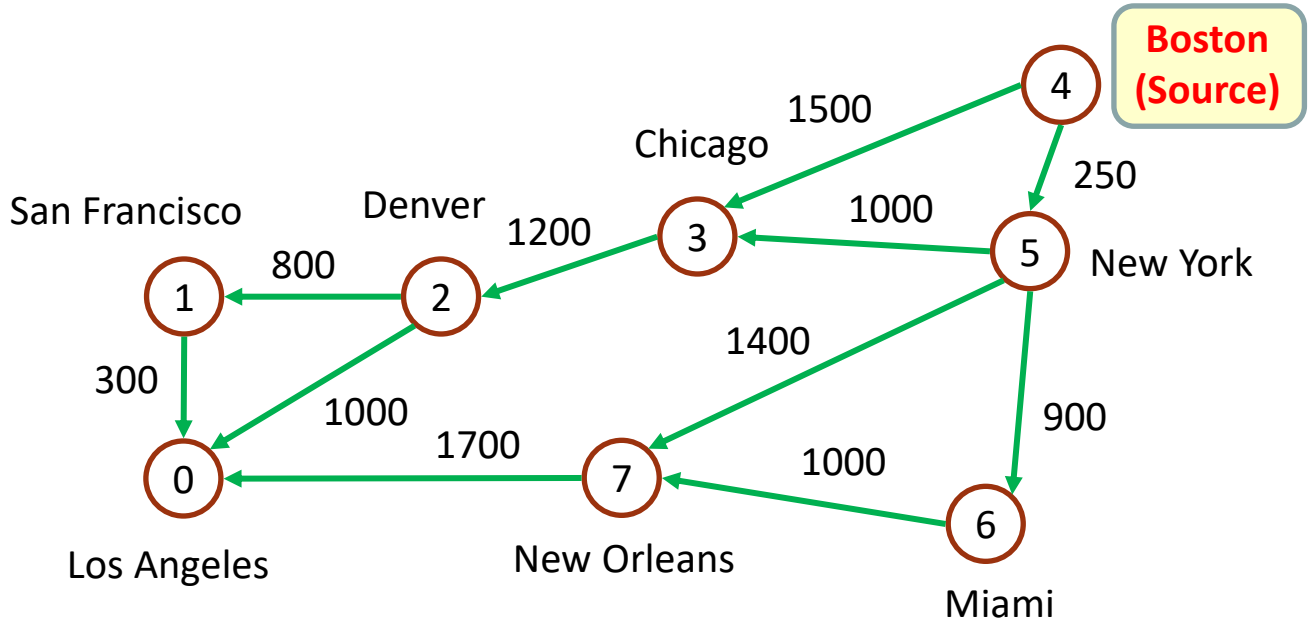


Program 6.9 & 10:
Single Source Shortest Paths – Dijkstra Algorithm 2/2

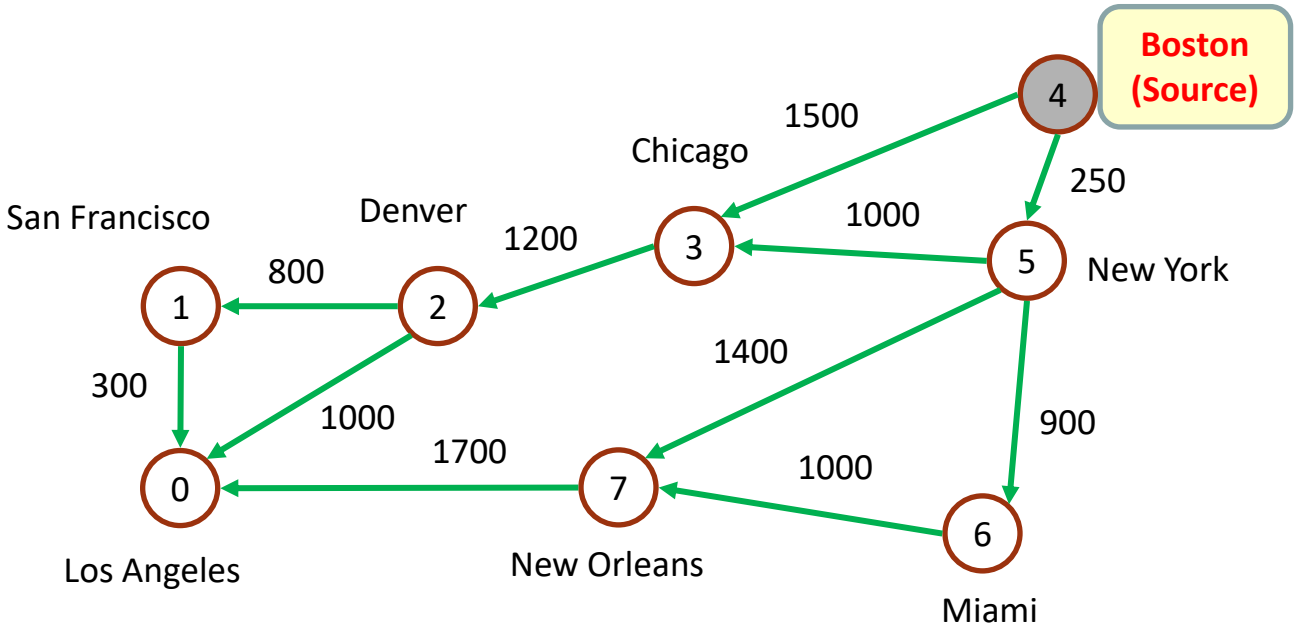
```
        for (w=0; w<n; w++)
                if(!found[w])
                        if (distance[u] + cost[u][w] < distance[w])
                                distance[w] = distance[u]+cost[u][w];
    }
}
```

```
int choose (int distance[], int n, short int found[])
{
    /* find the smallest distance not yet checked */
    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;
}
```





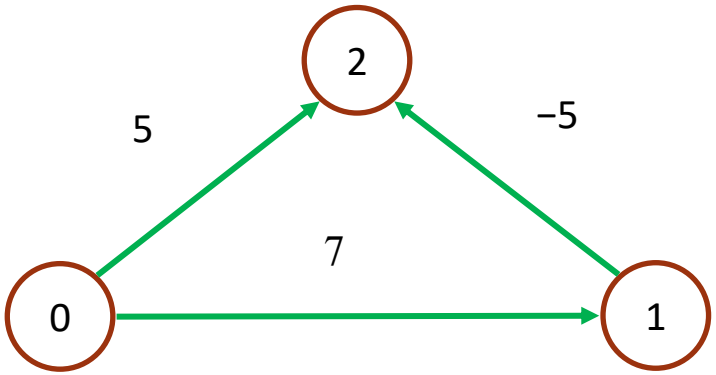
	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4				1500	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0

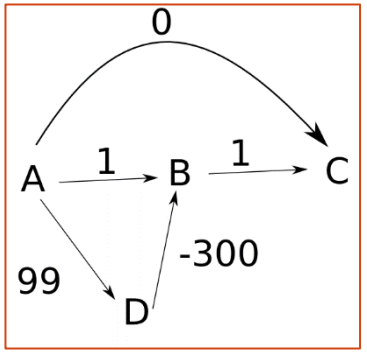


Iteration	Vertex Selected	Distance							
		LA	SF	DEN	CHI	BOST	NY	MIA	NO
		[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	---	∞	∞	∞	**1500**	**0**	**250**	∞	∞
1	5	∞	∞	∞	1250	0	250	**1150**	**1650**
2	6	∞	∞	∞	1250	0	250	1150	**1650**
3	3	∞	∞	**2450**	1250	0	250	1150	1650
4	7	**3350**	∞	2450	1250	0	250	1150	1650
5	2	3350	**3250**	2450	1250	0	250	1150	1650
6	1	**3350**	3250	2450	1250	0	250	1150	1650
7	0	3350	3250	2450	1250	0	250	1150	1650

Single Source/All Destinations: General Weights

- We now consider the general case when some or all of the edges of the directed graph *G* may have negative length.
  （我們現在考慮邊上的權重有可能是負的情形）
- The function shortestpath does not necessarily work.
  （**shortestpath 有可能會出錯!**）
  - According to the function, we have *dist*[1]=7, *dist*[2]=5.
  - But the shortest path from 0 to 2 is 0, 1, 2.
  - This path has length 2 !

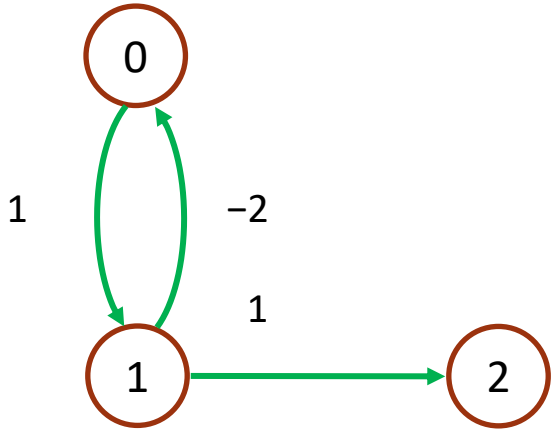




Iteration	Vertex Selected	Distance			
		A	B	C	D
Initial	---	**0**	1	0	99
1	C	0	1	0	99
2	B	0	1	0	99
3	D	0	-201	0	99

No negative cycle is permitted !

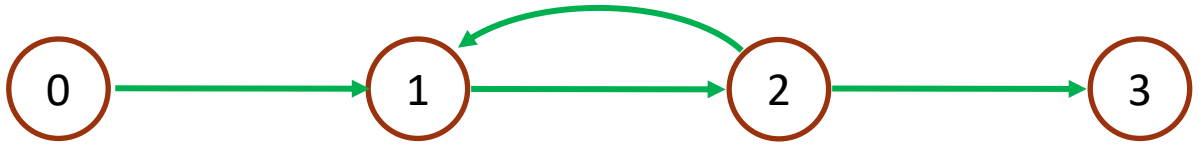
- When negative edge lengths are permitted, we require that the graph have no cycles of negative length.
  (當我們考慮邊上的權重有可能是負的情形, **我們不允許迴路長度為負的**)
- This is necessary so as the ensure that shortest paths consist of a finite number of edges.
  (為了確保最短路徑的邊數為有限個, 我們必須作這個限制)



Directed graph with a cycle of negative length.

Observation

- When there are no cycles of negative length, there is a shortest path between any two vertices of an *n*-vertex graph that has at most *n*-1 edges on it.
  （當迴路長度不為負的, 任何兩點的最短路徑至多有*n*-1 邊）
  - Otherwise, the path must repeat at least one vertex and hence must contain a cycle.
    （否則路徑至少會在某一點重覆, 因此路徑至少會包含一個迴路）
- So, elimination the cycles from the path results in another path with the same source and destination. The length of the new path is no more than that of the original.
  （因此，消除這個迴路會產生另一個新的路徑. 新的路徑長度只有可能更短）



Definition

- Let ***dist^k[u]*** be the length of a shortest path from the source vertex *v* **to vertex *u*** under the constraint that the shortest path contains at most *k* edges.
  （令 *dist*^{k}[*u*] 表示, **在最多使用 *k* 個邊的條件下, 從 *v* 到 *u* 的最短路徑**）
- Then, *dist*^{1}[*u*] = length[*v*][*u*], $0 \le u < n$.
  （因此, *dist*^{1}[*u*] = length[*v*][*u*], $0 \le u < n$.）
- Our goal then is to compute *dist*$^{n-1}$[*u*] for all *u*.
  （我們的目標是計算 *dist*$^{n-1}$[*u*] ）
- This can be done using the **dynamic programming** methodology.
  （我們將使用**動態規劃**來完成）

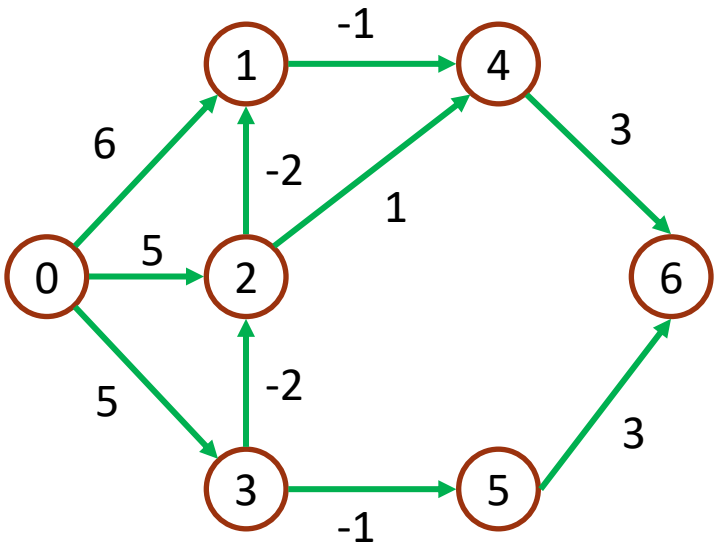
The Main Spirit of Bellman-Ford Algorithm

- If the shortest path from *v* to *u* with at most *k*, *k* > 1, edges has no more than *k*-1 edges, then $dist^k[u] = dist^{k-1}[u]$.
  (如果在最多使用 *k* 個邊的條件下, 從 *v* 到 *u* 的最短路徑只用到最多*k*-1個邊, 則 $dist^k[u] = dist^{k-1}[u]$)
- If the shortest path from *v* to *u* with at most *k*, *k* > 1, edges has exactly *k* edges, there exists a vertex *i* such that $dist^{k-1}[i]$ + length[*i*][*u*] is minimum.
  (如果在最多使用 *k* 個邊的條件下, 從 *v* 到 *u* 的最短路徑恰好用了*k* 個邊, **則存在一個點 *i* 使得 $dist^{k-1}[i]$ + length[*i*][*u*] 會最小**)
- These observations result in the following recurrence for dist:
  (以上的觀察產生以下的遞迴關係)

$$dist^k[u] = \min\{dist^{k-1}[u], \min_i\{dist^{k-1}[i] + length[i][u]\}\}$$

Figure 6.31:
Shortest paths with negative edge lengths

$$dist^k[u] = \min\{dist^{k-1}[u], \min\{dist^{k-1}[i] + length[i][u]\}\}$$



(a) A directed graph

k	$dist^k[u]$						
	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) $dist^k$