Graph Theory
(with Computer)

圖論
(可使用電腦)

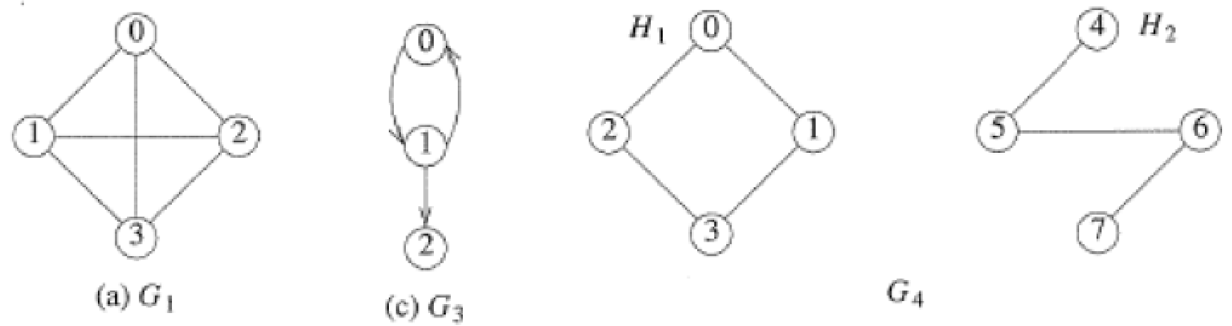
林劭原老師

Graph Representations 如何在電腦中表示圖

- adjacency matrix
- adjacency lists
- adjacency multilists

Adjacency Matrix 相鄰矩陣

- Adjacency matrix is an $n \times n$ matrix, say a, such that $a[i][j] = 1$ iff $(i,j) \in E(G)$($< i,j > \in E(G)$ for a digraph).



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{array}$$

(a) G_1

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{array}$$

(b) G_3

$$\begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

(c) G_4

Figure 6.7: Adjacency matrices

Adjacency Matrix 相鄰矩陣

- 對 undirected graph 而言，adjacency matrix 是對稱矩陣，因此也可以只存 upper triangle 或 lower triangle of the matrix.
- adjacency matrix representation 的優點：很容易得知
  是否 $(i,j) \in E(G)$
  degree of a vertex
  in-degree of a vertex = column sum
  out-degree of a vertex = row sum
  the total number of edges
- adjacency matrix representation 的缺點：
  因為要存an $n \times n$ matrix, 凡是用這種表示法的 algorithms, 都要花 $\Omega(n^2)$ time.
  如果 $e \ll n^2/2$, 就不太合適了，浪費memory, 浪費time.

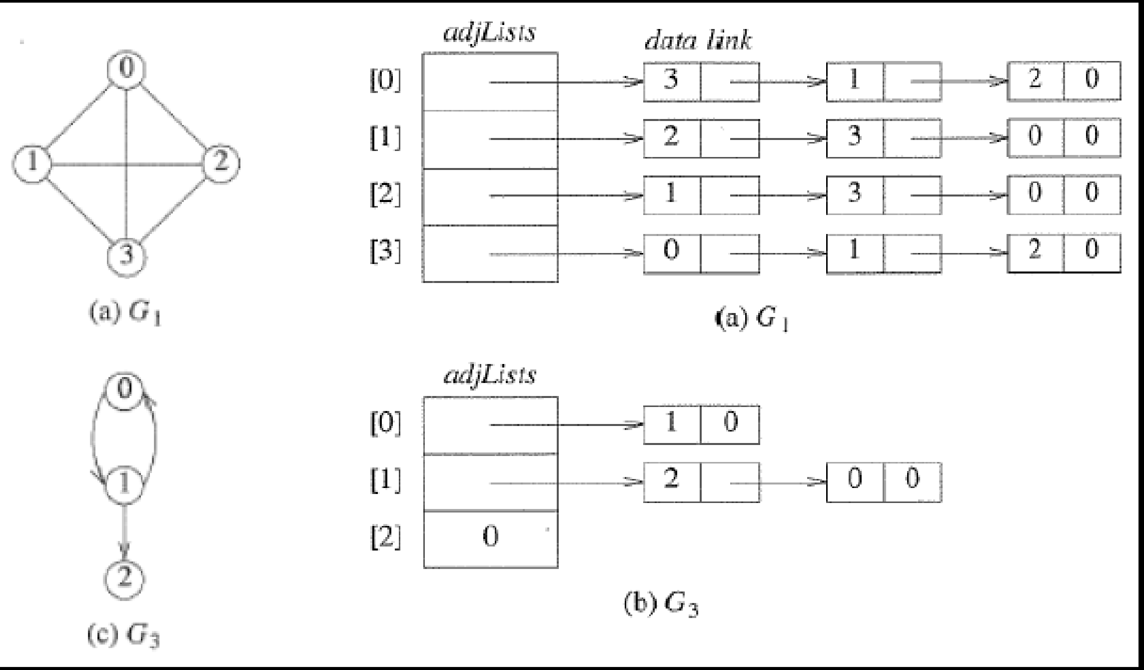
Adjacency Lists 相鄰串列

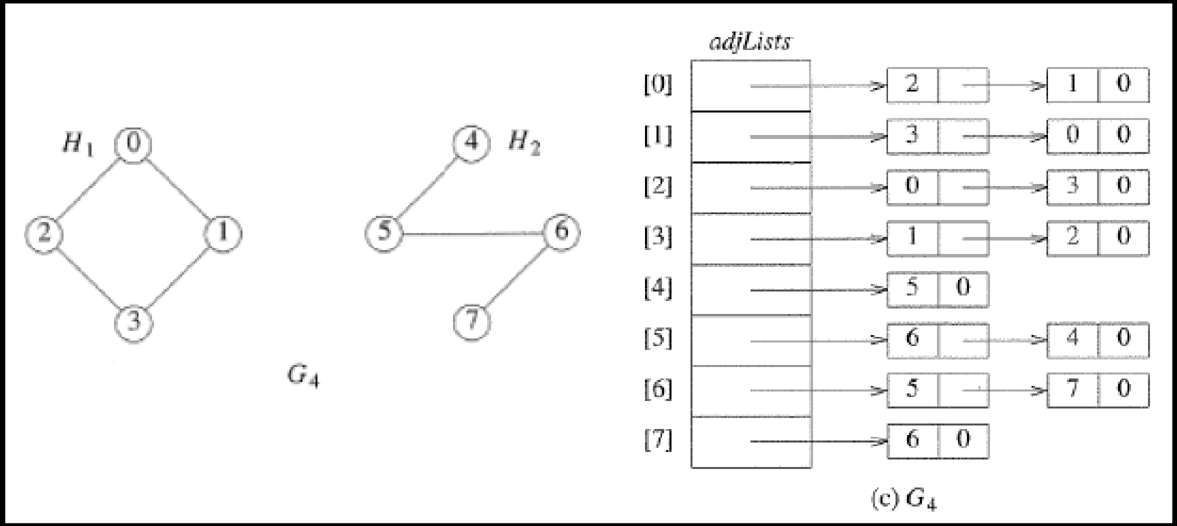
- 每一 vertex 用 a linked list 表示.Node structure of adjacency lists:

data	link

- List i 中存著 adjacent from i 的 vertices(次序可任意).
  每一 list 均有 header node.
- adjacency lists representation 的優點:
  求degree of a vertex, out-degree of a vertex, total number of edges 可以快一點
- adjacency lists representation 的缺點:
  要得知是否 $(i,j) \in E(G)$(或$<$ i, j $>\in E(G)$), in-degree of a vertex 都變麻煩.
  當 $e = \Omega(n^2)$, 沒有比 adjacency matrix 快，也不省 memory.

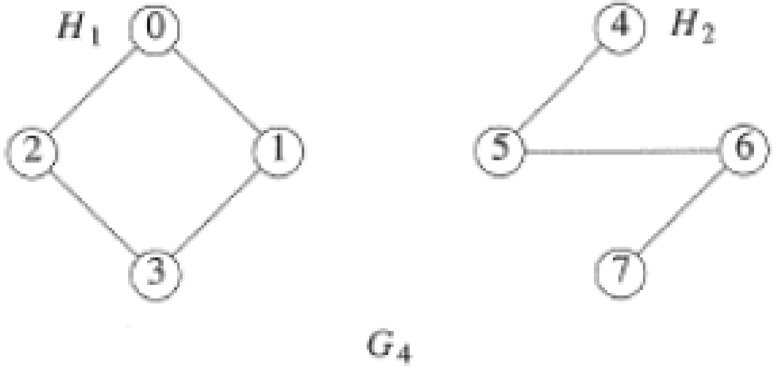
Adjacency Lists 相鄰串列





Adjacency Lists 相鄰串列

- adjacency lists的變化：如果 graph 本身沒有變動(沒有insert/delete)，而我們也不想浪費memory存link fields(指標欄位)，則可用an array of size $n + 2e + 1$ 來存，書上假設array的名字是node。
- 方法是:令$node[n] = n + 2e + 1$,node[n]用來做「擋土牆」
  vertex i 的鄰居資訊，儲存在array中的$node[i]$~$node[i+1] - 1$



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

Figure 6.9: Sequential representation of graph G_4

Adjacency Multilists

- multilists: Lists in which nodes may be shared among several lists.
- 對undirected graph而言，每一edge(u,v)在adjacency lists表示法中，都出現2次，1次在list u中，1次在list v中。
  在某些情況下，一旦某edge被examined檢查，我們就要將此edge所有出現處都mark，因此，最好只讓每一edge只出現一次，但使它屬於two lists。
- Node structure for adjacency multilists:

m	vertex 1	vertex 2	link 1	link 2

- m: a Boolean mark field, indicating whether the edge has been examined.

Adjacency Multilists

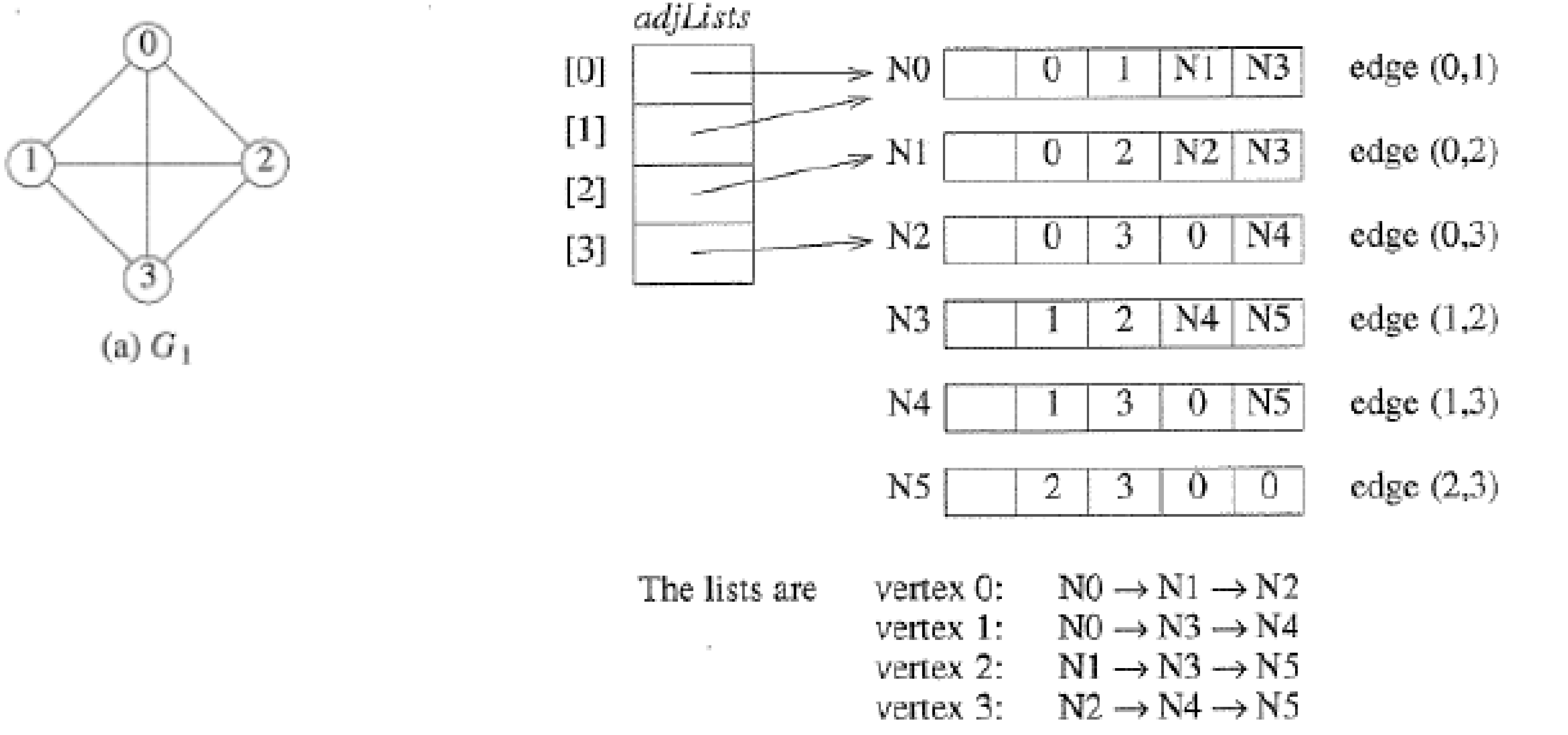


Figure 6.12: Adjacency multilists for G_1 of Figure 6.2(a)

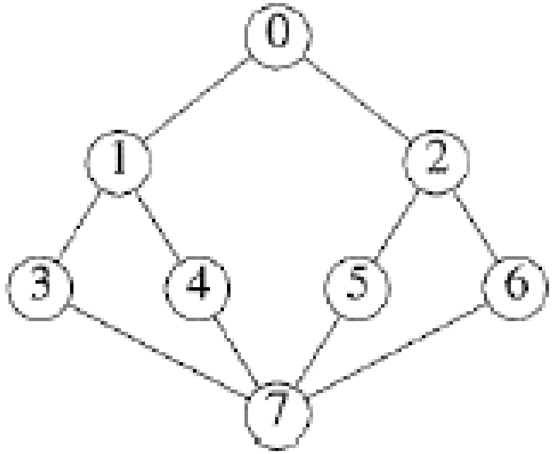
Weighted Edges 邊上有權重

- 對 adjacency matrix 而言，weight≠0 可直接存於 a[i][j].
- 對 adjacency lists 而言，可增加一個 field weight 來存。
- A graph with weighted edges is called a network.

Elementary Graph Operations

- graph 的 traversals 遍歷
- depth-first-search (深先搜尋) (DFS)
- breadth-first-search (廣先搜尋) (BFS)

例如:



若由 0 開始,
BFS: 0 1 2 3 4 5 6 7 (一層一層地),
DFS: 0 1 3 7 4 5 2 6 (先走沒走過的)

Depth-First Search(DFS)

- We begin by visiting the start vertex v.
- Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated發起.
- When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w.
- The search terminates終止 when no unvisited vertex can be reached from any of the visited vertices.

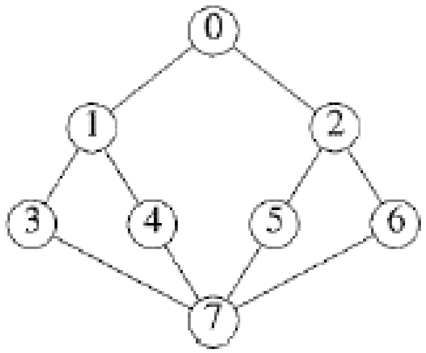
Depth-First Search(DFS)

- void Graph::DFS(){ //driver 啟動做事者
```
        visited = new bool [n];
        fill(visited, visited + n, false);
        DFS(0);   //suppose that the search starts at vertex 0
        delete[] visited;
}
```
- void Graph::DFS(int v){ //workhorse 苦力，真正做事者
```
        visited[v] = true;
        for(each vertex w adjacent to v)   //這行需要自己寫
                if(!visited[w]) DFS(w);
}
```

- 因為不想重複visit相同的點，需要一個array來紀錄是否被visited了。
- 當 v 有多個連接到的點都是unvisited時，誰被選中成為w，取決於adjacency lists 中排列的次序。
- DFS 通常寫成recursive形式。(因為recursice, 所以用到stack)

Depth-First Search(DFS)

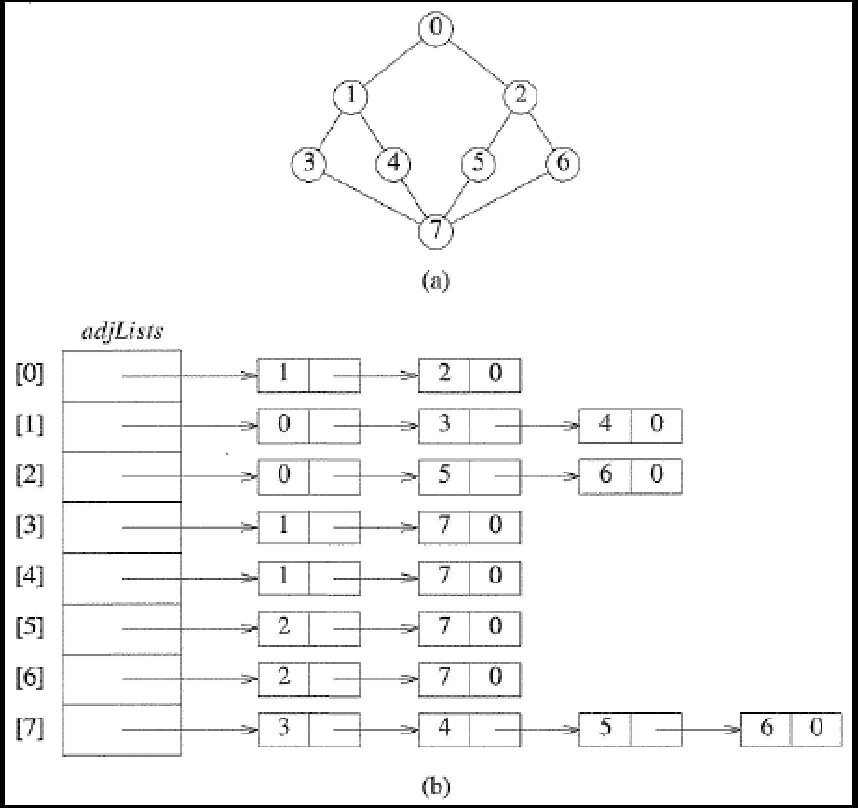
例如:



若由 0 開始,

BFS: 0 1 2 3 4 5 6 7 (一層一層地),

DFS: 0 1 3 7 4 5 2 6 (先走沒走過的)



- DFS呼叫DFS(0)，得 0 1 3 7 4 5 2 6
- DFS可以檢查一個 graph 是否 connected, 列出所有 connected components.
- 花的時間:用 adjacency lists 花 $O(e)$ time, 用 adjacency matrix 花$O(n^2)$ time.

Breath-First Search(BFS)

- In a breadth-first search, we begin by visiting the start vertex v.
- Next, all unvisited vertices adjacent to v are visited.
- Unvisited vertices adjacent to these newly visited vertices are then, visited, and so on.

Breath-First Search(BFS)

- 也需要一個 array 來紀錄是否被 visited了，被 visit 到的次序與 adjacency lists 中排列的次序有關。
- BFS通常寫成 iterative 形式，而且用到 queue。

- 上面例子: 0 1 2 3 4 5 6 7
- BFS 也可以檢查一個 graph 是否 connected, 列出所有 connected components.
- BFS花的時間:同DFS

Breath-First Search(BFS)

```
void Graph::BFS(int v){
        visited = new bool[n];
        fill(visited, visited + n, false);
        visited[v] = true;
        Queue<int> q;
        q.Push(v);
        while(!q.IsEmpty()){
                v = q.Front();
                q.Pop();
                for(all vertices w adjacent to v)  //這行需自己寫
                        if(!visited[w]){
                                q.Push(w);
                                visited[w] = true;
                        }
        }
        delete [] visited;
}
```

Spanning Trees 生成樹

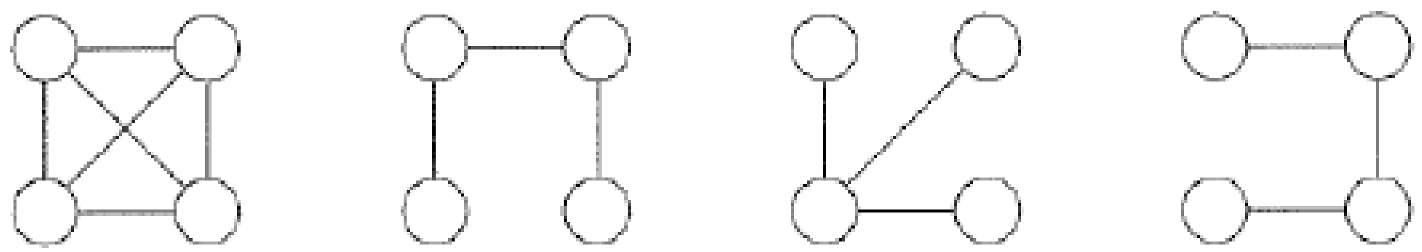


Figure 6.18: A complete graph and three of its spanning trees

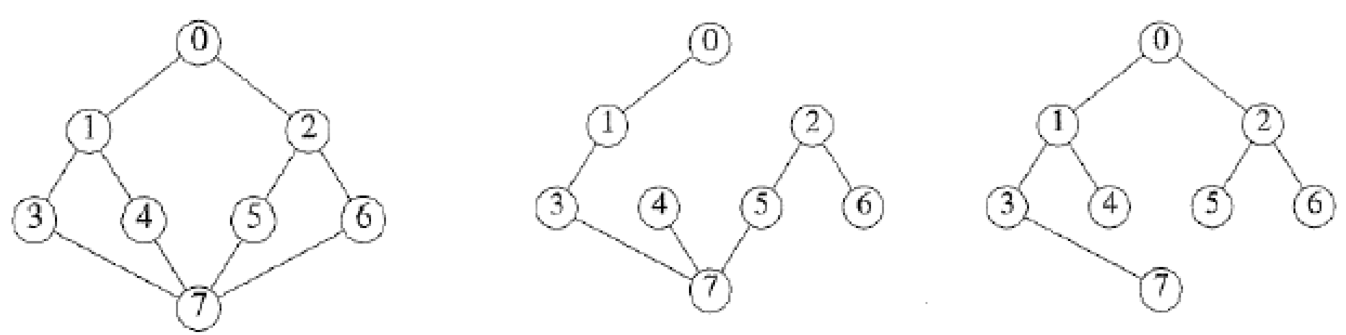
Spanning Trees 生成樹

- A tree is a connected acyclic graph. 連通的、沒有cycle的圖
- A spanning tree of G is a tree that includes all vertices of G. 包含G所有點的tree

- 只要 G 是a connected graph, G 就有 spanning tree(s).
- 當 G 沒有 cycle 時，就只有一個 spanning tree.

Spanning Trees 生成樹

- Spanning tree 可能有很多個，利用 DFS(或BFS) 可幫我們找出一個 spanning tree, 稱之為 DFS(或BFS) spanning tree.
- 方法是:能 visit 到 a new vertex 的 edges 就 ∈ tree. 例如:



(a) DFS(0) spanning tree (b) BFS(0) spanning tree

- 令T表spanning tree的edges所成之集合，則DFS(或BFS)的格式，只要在 if-指令中增加 $T = T \cup \{(u, v)\}$ 就可以找出DFS(或BFS) spanning tree.

Spanning Trees 生成樹

- 一個有 n vertices 的 connected graph 有 $\geq n-1$ edges.
- Spanning tree of G (say G’), 是一個有最少的邊數，而且滿足 V(G‘) = V(G) 的 connected graph, 其他滿足 V(G’) = V(G) 的 connected graphs 邊數都比 spanning tree 多。

Minimum-cost Spanning Trees

- 當 graph 的 edges 有 weights 時，一個 spanning tree 的 cost = 它的 edges 的 weights(costs)之和。
- 所有 spanning trees 中，cost 最小的那個(有可能同時有多個 spanning trees 的 cost 的最小)稱為是 a minimum spanning tree.(我們簡寫成MCST)
- 我們會介紹3個求MCST的方法，他們全都是 greedy methods.
- G的MCST的建構過程必須注意:
  (1)只能用G中的 edges,
  (2)恰好用 n-1 edges,
  (3) 不可有 cycles.

Minimum-cost Spanning Trees

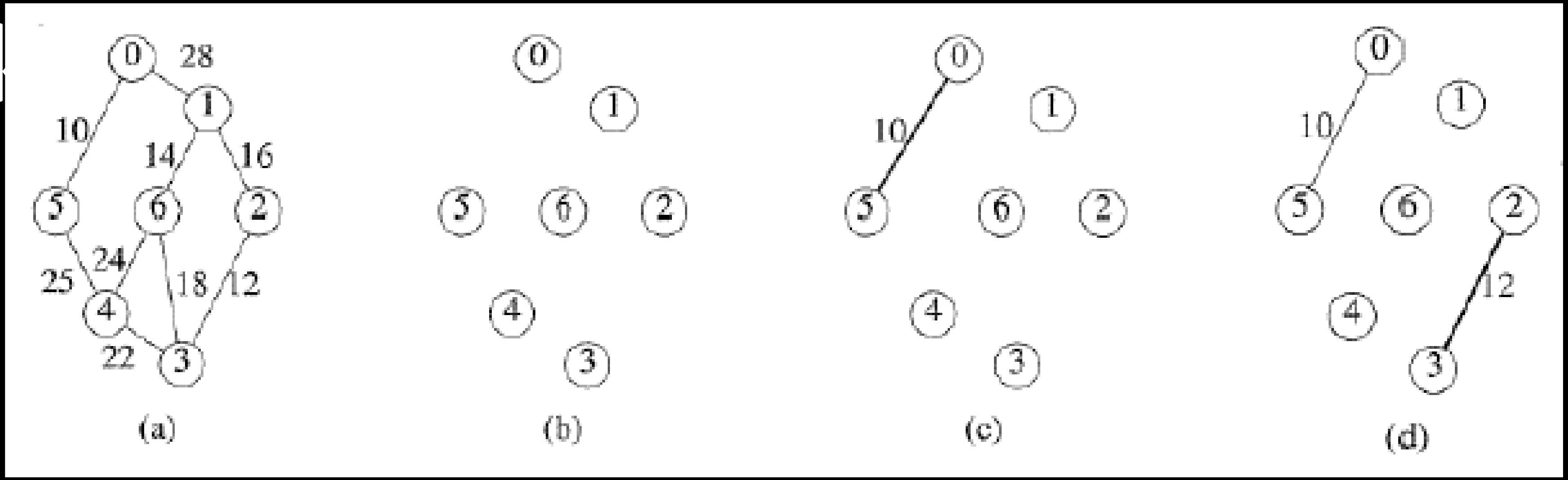
- 以下是3個求MCST的方法：
- Kruskal’s algorithm
- Prim’s algorithm
- Sollin’s algorithm

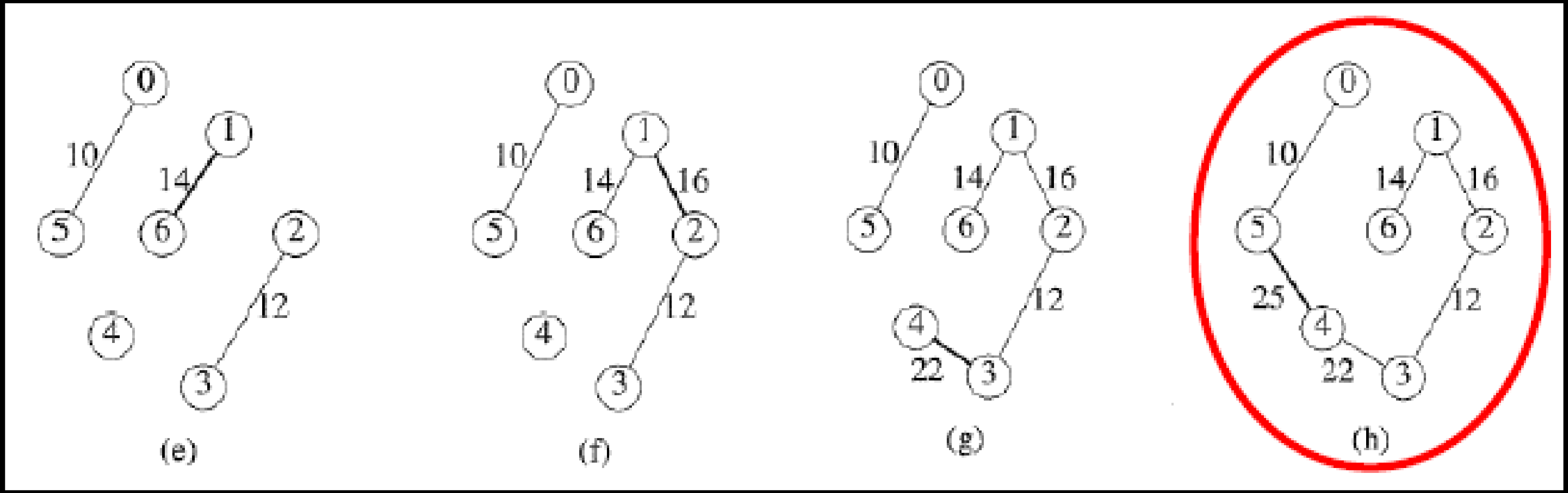
Kruskal’s Algorithm

- 先將G中的 edges 依 weights(costs) 由小到大 sort 好。
- Initially, 每個 G 中的點都各自在一個 tree 中。
- Greedy method:每次選擇 cost 最小且不會與已選出的 edges 形成 cycle 的 edge, 一直到有 n-1 個 edges 被選出為止。(每次只加一個邊)
- pseudo code:

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

Krusk





Kruskal's Algorithm

- Kruskal's algorithm 可在 O(eloge) time 完成, 其中 e = |E(G)|.
- 方法是:
  Sort all edges, 花 O(eloge) time.
  利用 union-and-find 的方法可查出是否造成 cycle.
- 例如:
  做完(f)之後，下一個 cost 最小的 edge 是(3,6)
  因為{{0,5},{1,2,3,6},{4}}而且3和6屬於 the same set, 所以知道用 (3,6) 會有 cycle.

林劭原 老師

Prim's Algorithm

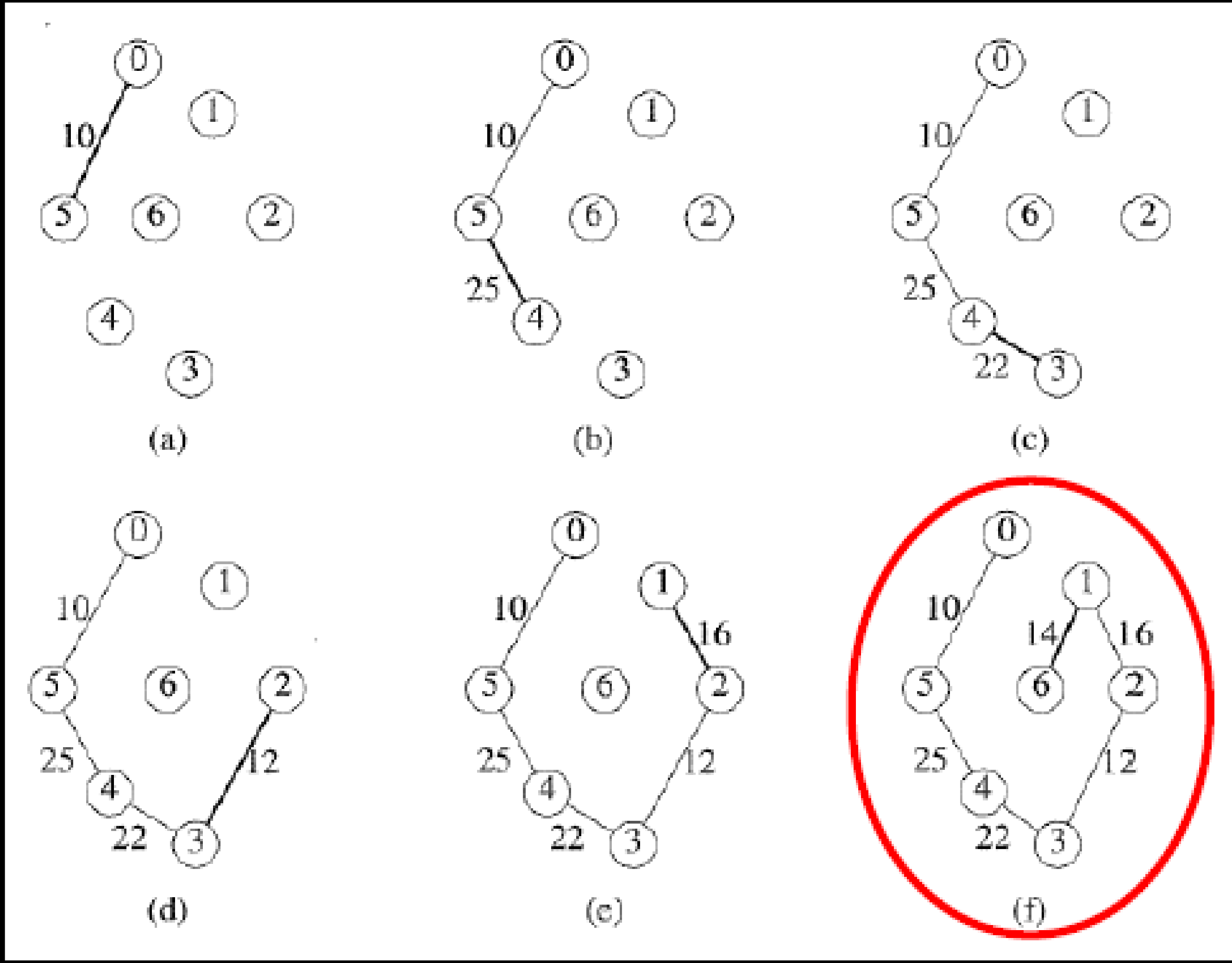
- Kruskal's algorithm 一開始是一個有 n trees 的 forest, 一直加邊直到形成 a tree, 必須做 sorting.
- Prim's algorithm 則是自始至終都只有 one tree, 但 tree 中的點數一直增加，直到 n vertices 全在裡面。
- Prim's algorithm: 由任一點開始，令T = {此點}.
- Prim's algorithm 用到的 greedy method 是:
  每次由連接著 ∈T 與 ∉T 之點的邊中，選出 cost 最小者，加到 T 中，(不必擔心 cycle ，因為不可能 why?) 直到 T 有 n-1 個 edges 為止。(每次只加一個邊)

Prim's Algorithm

- // Assume that G has at least one vertex

```
TV = {0};   //start with vertex 0 and no edges
for(T = ∅;T contains fewer than n-1 edges;add (u,v) to T){
        Let (u,v) be a least-cost edge such that u∈TV and v∉TV;
        if (there is no such edge) break;
        add v to TV;
}
if (T contains fewer than n-1 edges) cout << "no spanning tree" << endl;
```

Prim's Algorithm



Prim's Algorithm

- Prim's algorithm 花 $O(n^2)$ time.

- 比較:
- Kruskal's algorithm $O(eloge)$ time 適合 e 小
- Prim's algorithm $O(n^2)$ time 適合 e 大

Sollin’s Algorithm

- 一開始每點都在 a tree 中，因此是一個有 n trees 的 forest.
- 每個 tree 找一個 cost 最小的、且連向外面的 edge.(Sollin’s algorithm 的 greedy method)
- 同一個邊可能同時被兩個 tree 選中，而且，當有數個邊的 cost 相同時，兩個 trees 可能選出不同的 edges 來連接彼此，這些邊當中，只能留下一個。