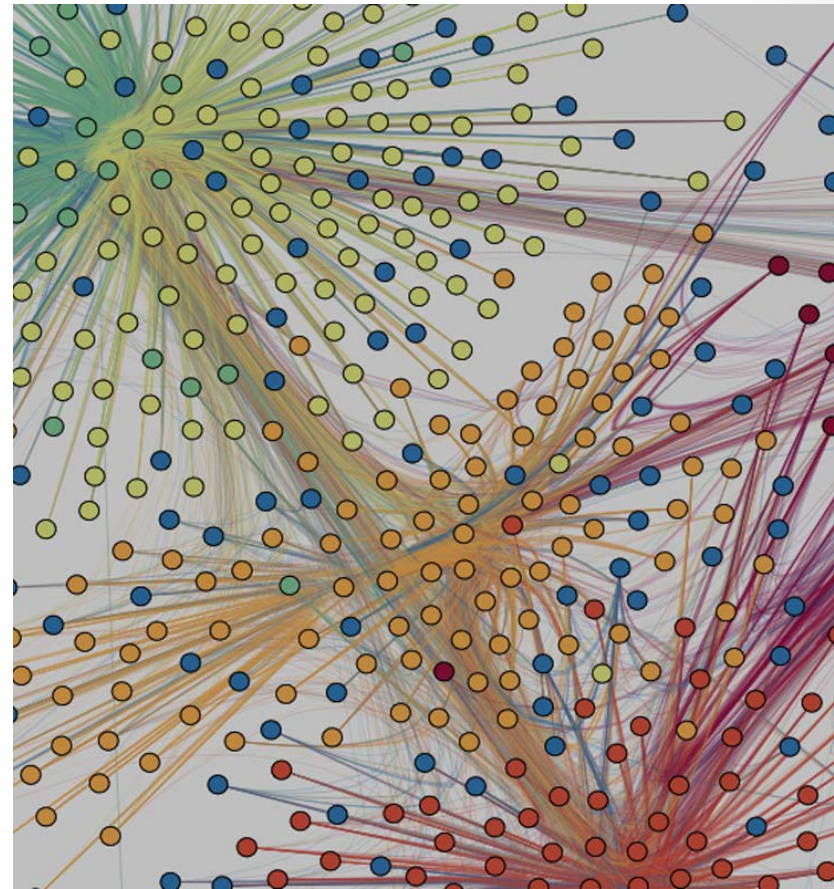


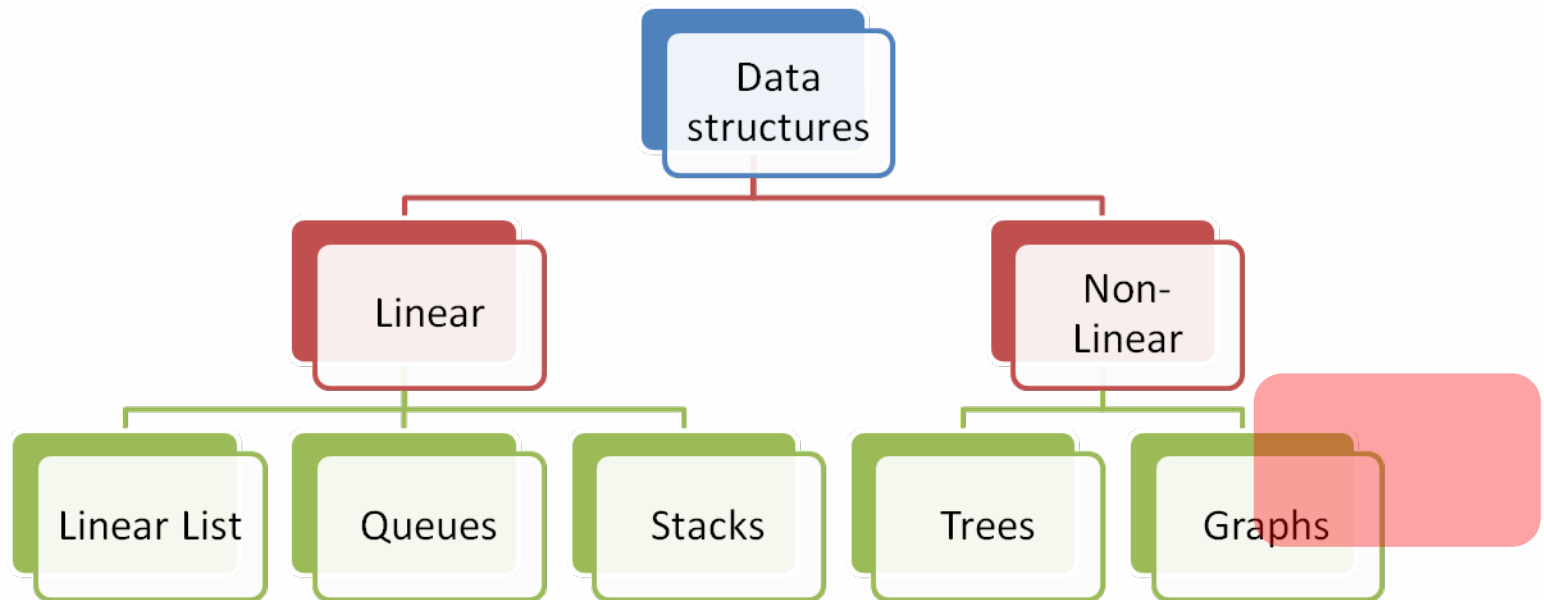
GRAPH (圖形)

許志仲(Chih-Chung Hsu)

Assistant Professor
Department of Management Information Systems,
National Pingtung University of Science and Technology



資料結構示意圖

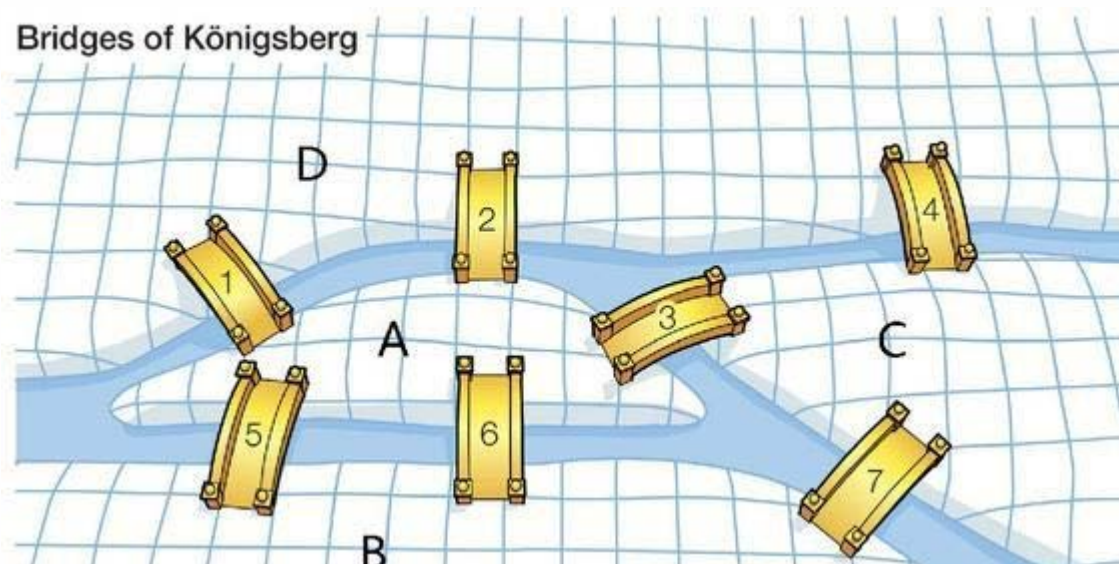


同時也是最有(恐)用(怖)的一個章節!!

圖論

■ Why?

- 七橋問題。Eular (數學家) 為了求解肯尼茲堡橋問題
 - 是什麼? 隨意一地出發，一次經過所有的橋再回到原點，每座橋只走一次

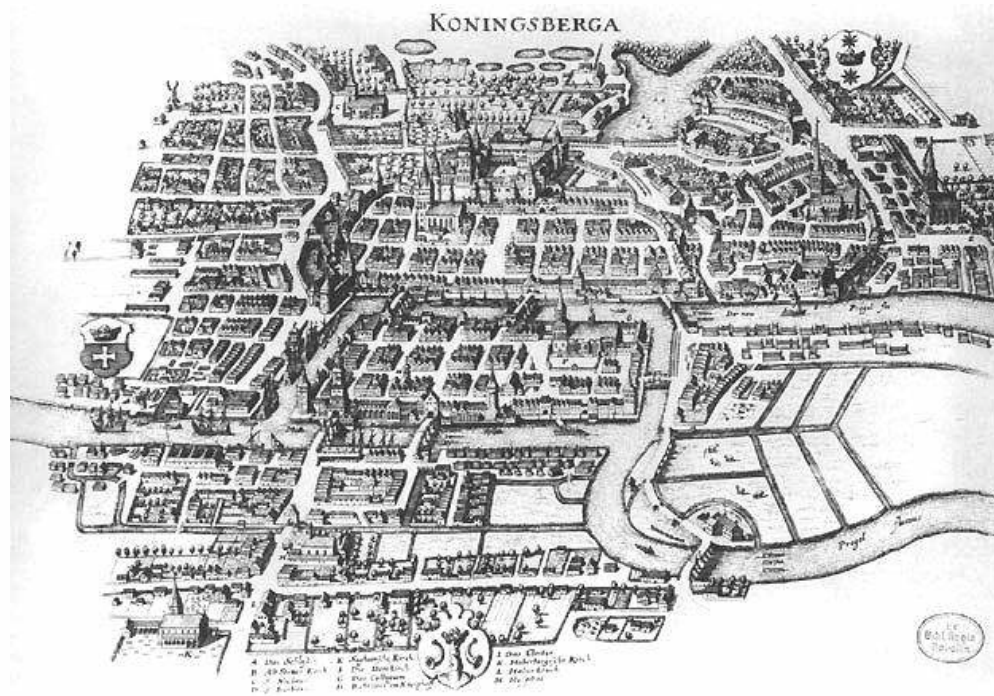


怎麼解決這個問題？

- 給定一張地圖：
 - 窮舉？
- 給定一千張地圖：
 - 逐一窮舉？
 - 找出規則？每張地圖都不一樣，怎麼去找規則？
- 要找規則（rule），必先將實際問題抽象化（abstraction）
 - 抽象化：將實際問題用模型（model）表示；通常是數學模型
 - 略去跟問題本質無關的細節，只留下關鍵元素
 - 抽象化是為了一般化（generalization）

抽象化

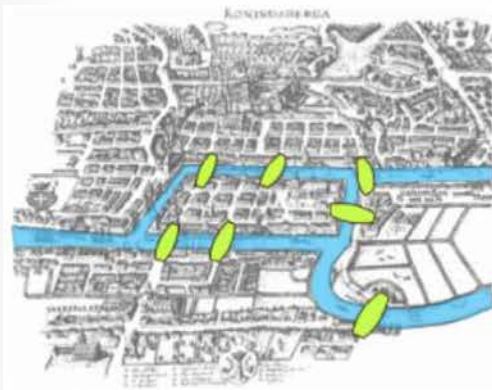
- 什麼是與問題本質無關的細節？



"Königsberg 1651" by Merian-Erbe is under Public Domain

抽象化

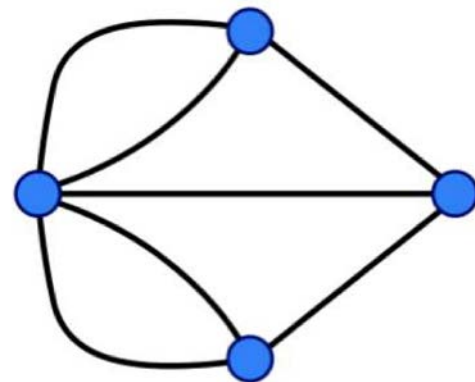
■ 七橋問題的抽象化：



- 尤拉把實際的抽象問題簡化為平面上的點與線組合，每一座橋視為一條線，橋所連接的地區視為點
- 最終的模型是一個圖（graph）或網路（network）
 - 由點（vertex、node）和線（edge、link、arc）組成

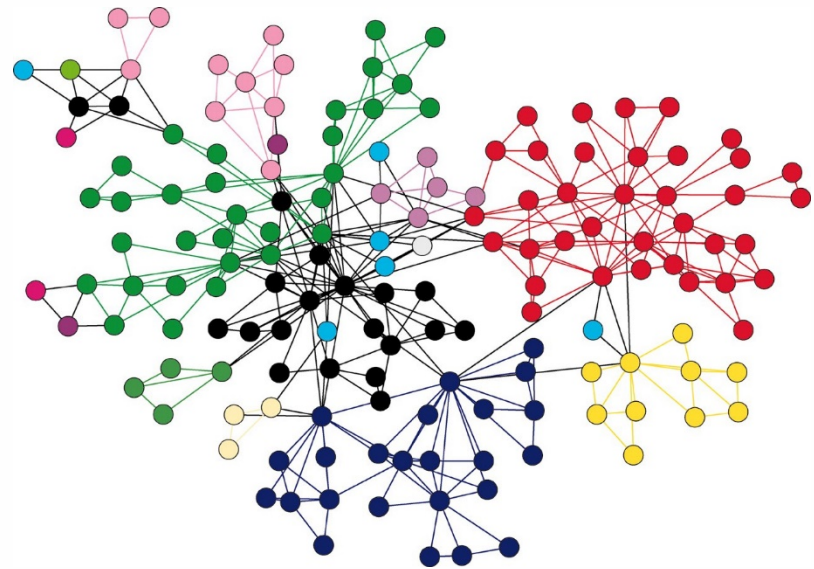
抽象化與一般規則

- 要解七橋問題，只要對著抽象化後的模型做探討即可
- 尤拉 (Euler) 在1735 年論述，這個圖不存一筆劃且不重複地走完所有的邊的解
 - 若從某點出發後最後再回到這點，則這一點上的連結數 (稱為「degree」) 必須是偶數，這樣的點稱為偶頂點。
 - 相對的，連有奇數條線的點稱為奇頂點。
 - 由於這個圖中存在四個奇頂點，所以必然無解



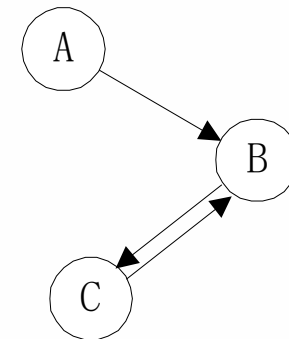
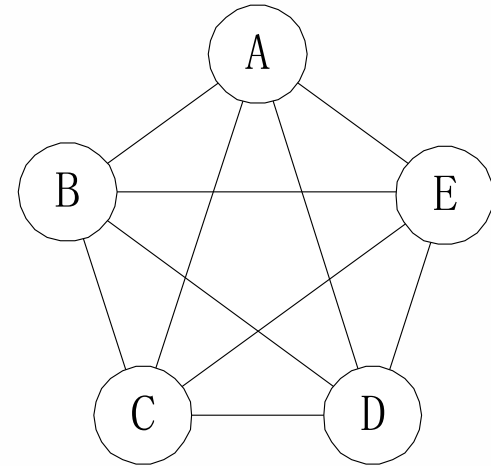
圖形結構能解的問題

- 圖論很廣，通常是屬於離散數學的一種
- 以程式來講，能解決的問題通常是下列這些
 - 最短路徑
 - 路徑規劃
 - Spanning tree
 - 找到最小可以連通的路徑的技巧
 - 子圖搜尋
 - 找到最小的相似結構組合，例如在社群媒體中找到特定的小群



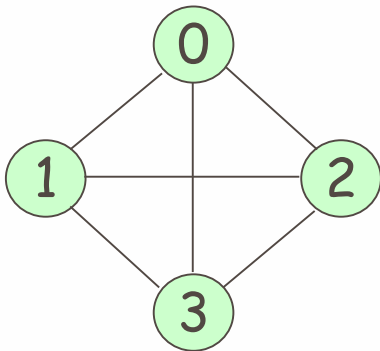
圖形結構定義

- 圖形(Graph)
 - $G = (V \cdot E)$
- 頂點(Vertexes，或稱Nodes)
- 邊(Edges)
- 無向圖(Undirected Graph)
 - $(V1, V2)$
- 有向圖(Directed Graph)
 - $\langle V1, V2 \rangle$



基本範例

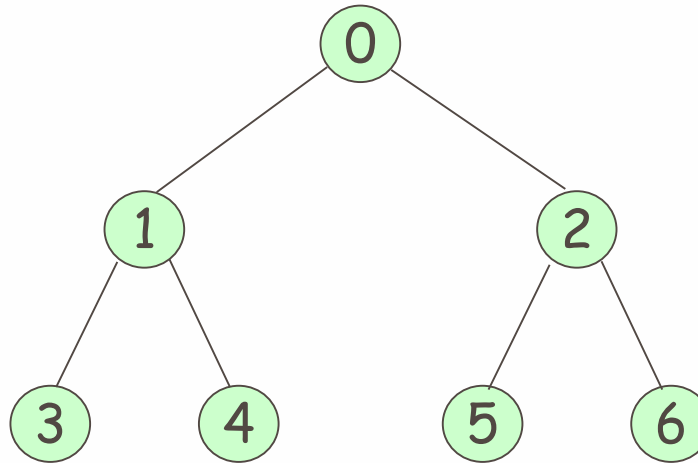
(a) G_1



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

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

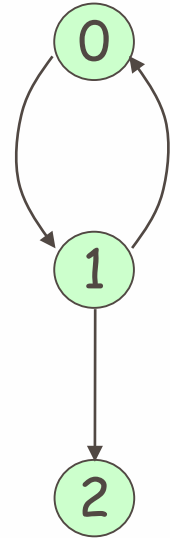
(b) G_2



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

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

(c) G_3



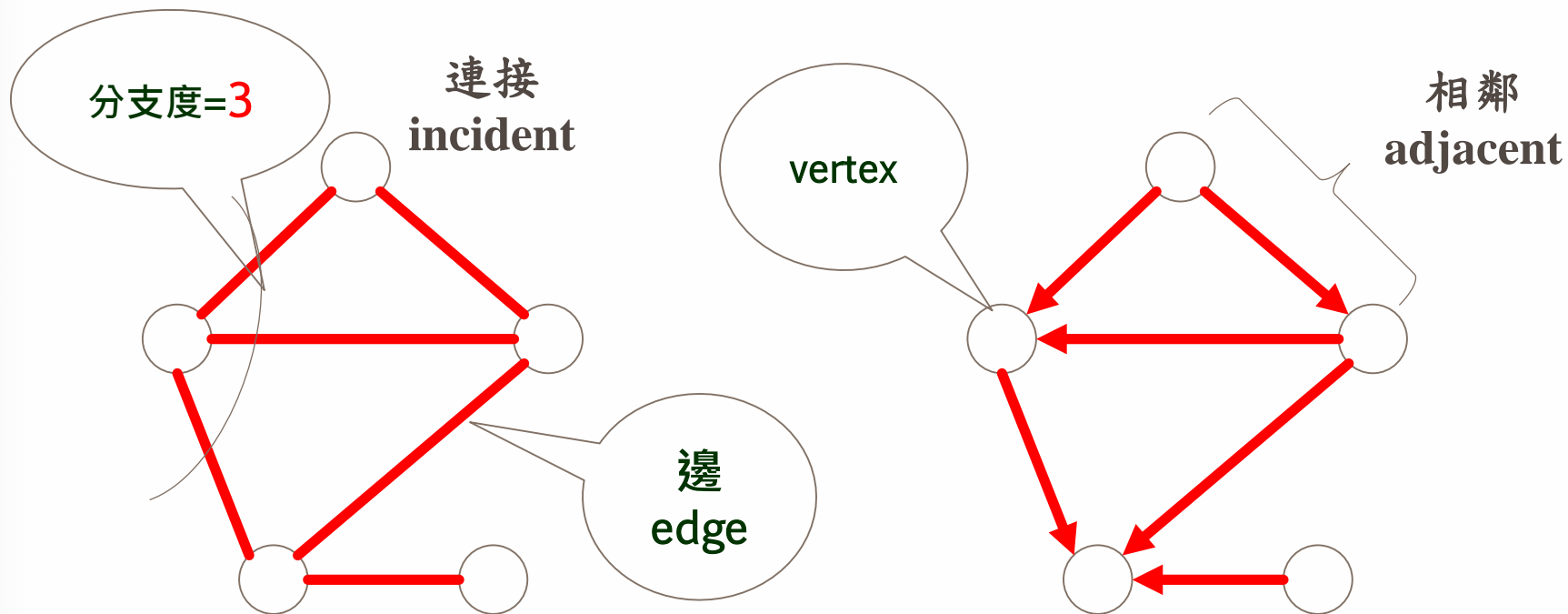
$$V(G_3) = \{0, 1, 2\}$$

$$E(G_3) = \{\langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 2 \rangle\}$$

Undirected Graph

Directed Graph

基本範例

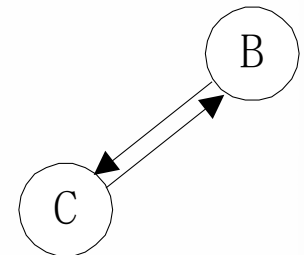
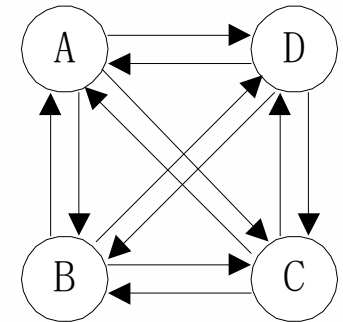
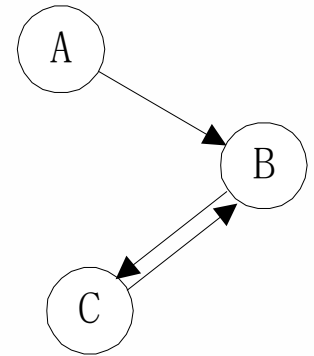
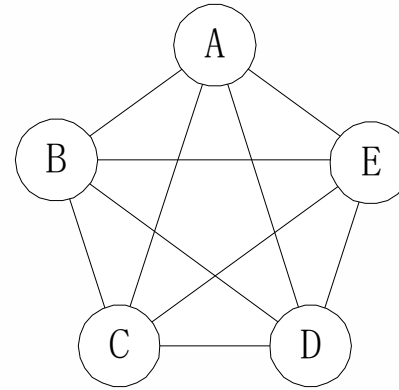


無向圖
undirected graph

有向圖
directed graph

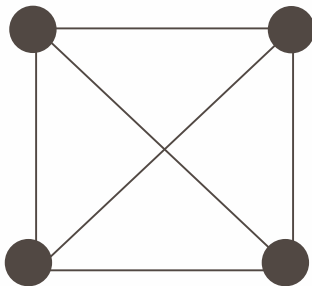
圖形的基本術語

- 完全圖(Complete Graph)
- 路徑(Path)
- 路徑之長度(Path Length)
- 簡單路徑(Simple Path)
- 迴路(Cycle)
- 相連的(Connected)
- 相連單元(Connected Component)
- 子圖(Subgraph)
- 緊密相連(Strongly Connected)
- 緊密相連單元(Strongly Connected Component)
- 出分支度(Out Degree)
- 入分支度(In Degree)

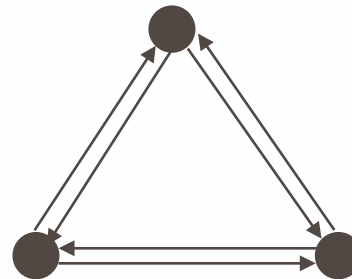


Complete Graph

- 整個圖每一個點，都有跟任意另一個點連線稱之
 - Vertex的數量可計算
- 依據圖形分類
 - Undirected graph
 - an n -vertex graph with exactly $n(n-1)/2$ edges.
 - Directed graph
 - an n -vertex graph with exactly $n(n-1)$ edges



Undirected



Directed

路徑與其長度(Path and Length)

■ 路徑(Path)

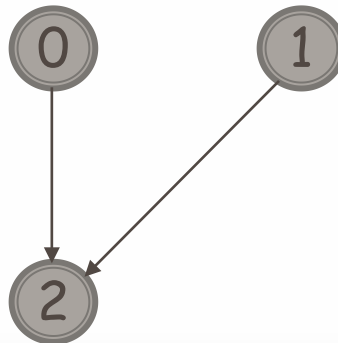
- 定義：在圖形G中，從Vertex v 到 u 所經過的edge，就叫做路徑。例如： $u, i_1, i_2, \dots, i_k, v \rightarrow (u, i_1), (i_1, i_2), \dots, (i_k, v)$.

■ 路徑長度(Length)

- 簡單把上面的路徑數量統計一下就是了...

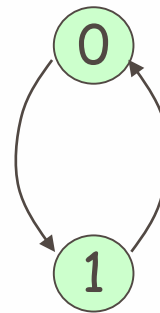
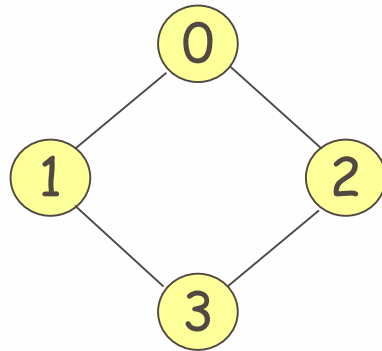
■ Simple Path

- 在一路徑中，除了起點與終點可以相同之外(不同亦可)，其餘頂點不可以重複。

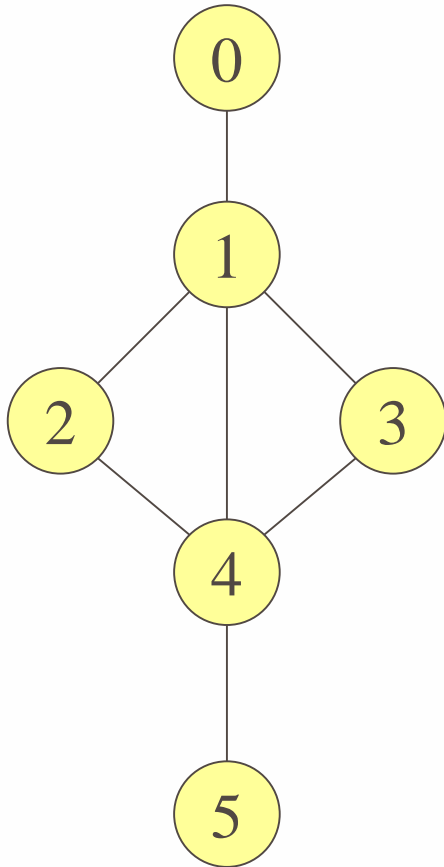


循環(Cycle)

- 圖形中的迴路是一條路徑，且必須符合下列兩要求
 - 是一條簡單路徑(Simple path)
 - 起點與終點是同個頂點。



表示一個路徑的基本範例



Path 1 $(0,1) (1,3) (3,4) (4,2) (2,1) (1,4) (4,5)$

Path 2 $(0,1) (1,3) (3,4) (4,5)$

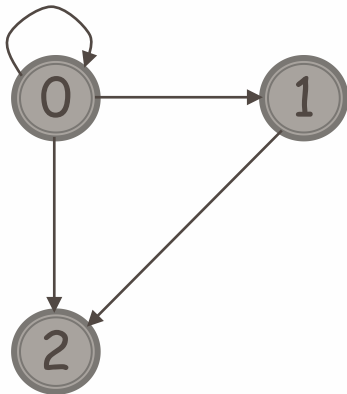
Path 3 $(1,3) (3,4) (4,2) (2,1)$

Simple Path $Path\ 2 \cup Path\ 3$

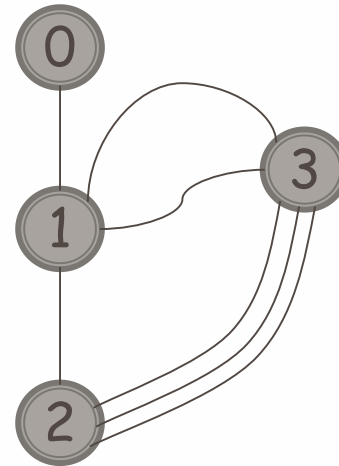
Circle $Path\ 3$

Simple Graph

- 沒有自我迴圈
 - 不能有一個Edge 是自己連到自己!!
- 沒有多重邊或平行邊
 - 不能兩個Vertices 之間有多個路徑!!
- 簡單路徑構成的圖，就是Simple graph



(a) Graph with a self edge



(b) Multigraph

連接與相鄰定義(Adjacent and Incident)

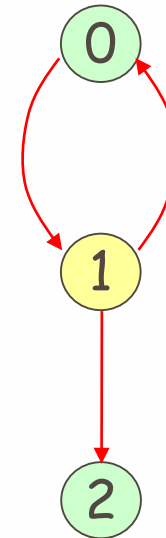
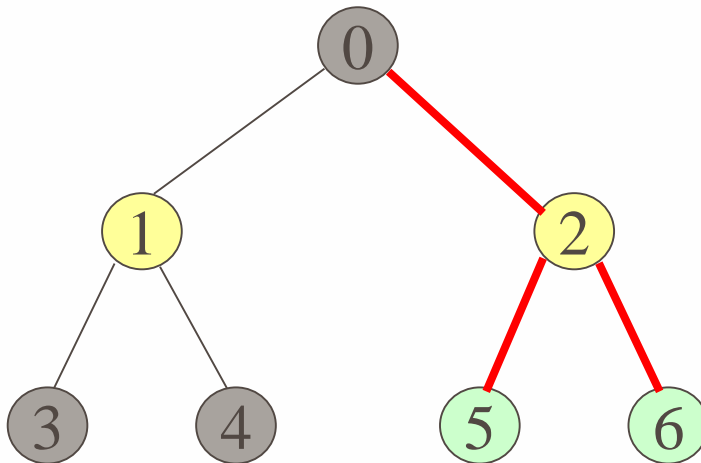
■ 不同圖形類別有不同的定義

■ Undirected graph

- If (u, v) is an edge in $E(G)$, vertices u and v are adjacent and the edge (u, v) is the incident on vertices u and v .

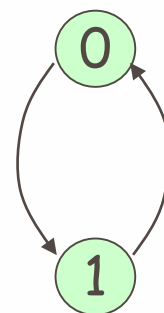
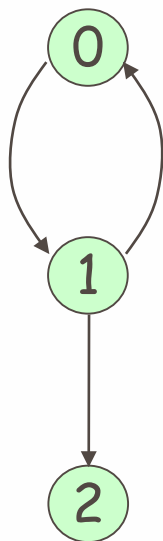
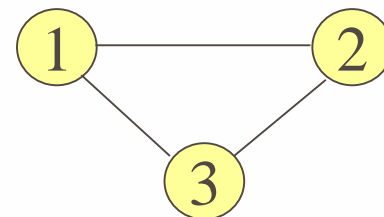
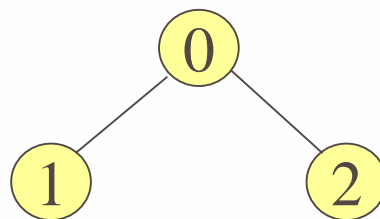
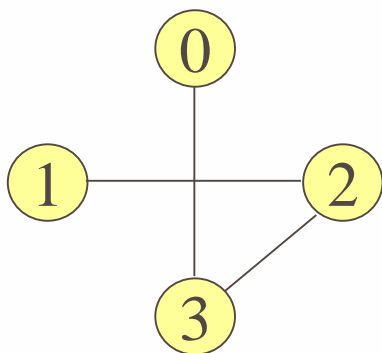
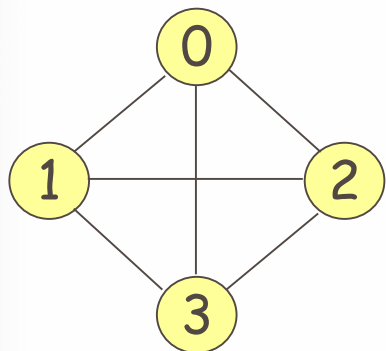
■ Directed graph

- $\langle u, v \rangle$ indicates u is adjacent to v and v is adjacent from u .
- $\langle u, v \rangle$ is incident to u and v .



子圖(Subgraph)

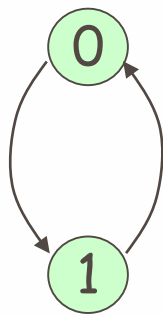
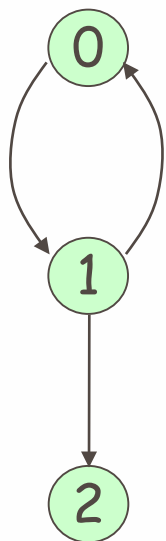
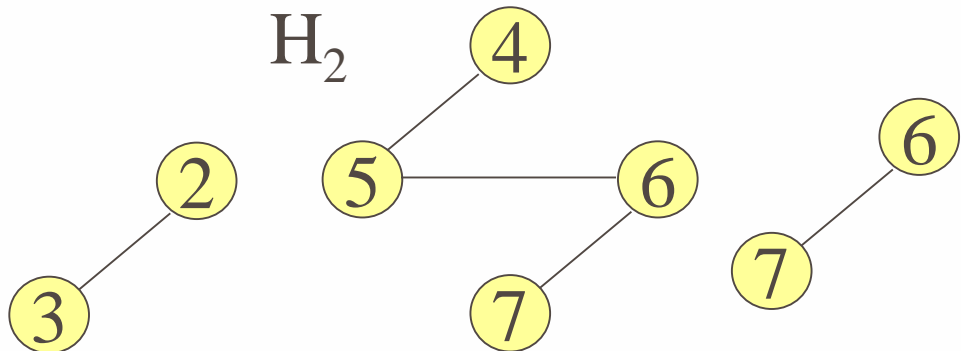
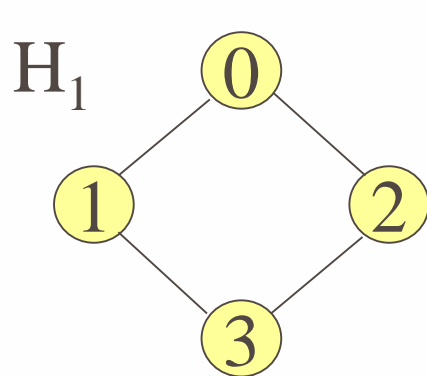
- 一圖形為 G ，其子圖為 G' ，則 $V(G') \subseteq V(G)$ 、 $E(G') \subseteq E(G)$.



連通圖Connected

- 若圖 G ，任2點間有一路徑存在，該圖稱為連通圖。
- 若圖 G 不連通，則圖 G 的最大連通子圖，稱為圖 G 的連通成分(connected components)
- 若圖 G 為有向圖，若且惟若圖 G 中任兩點 u 到 v 有一路徑存在，則 v 到 u 亦存在有一路徑，則圖 G 稱為強連通(strongly connected)
- 有向圖 G 中，最大強連通子圖，稱為圖 G 的強連通成分(strongly connected components)

連通圖的基本範例

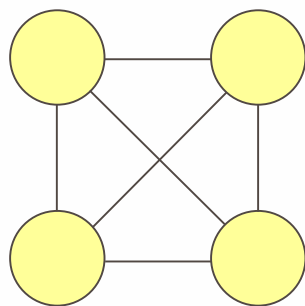


G_3

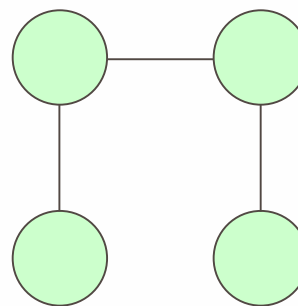
森林和樹(Forest & Tree)

- 森林是沒有迴路(Cycle) 的圖。
- 樹是連通的森林。
- 生成樹(Spanning tree) 是圖G 的形成樹的生成子圖。

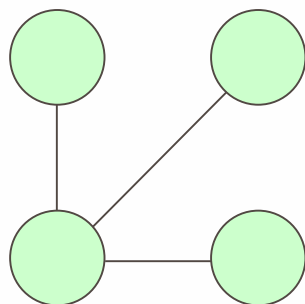
完全圖



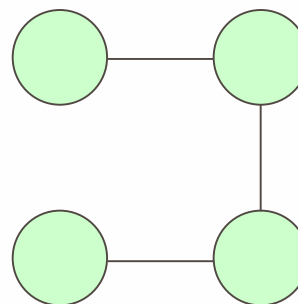
SP-1



SP-2



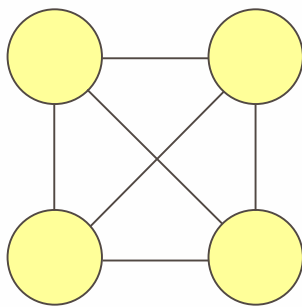
SP-3



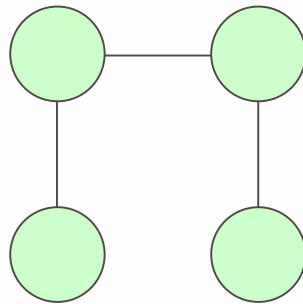
生成樹的特質

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

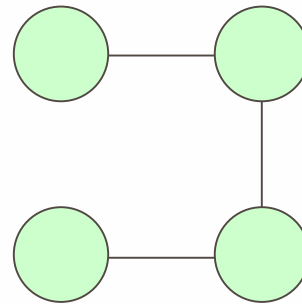
完全圖



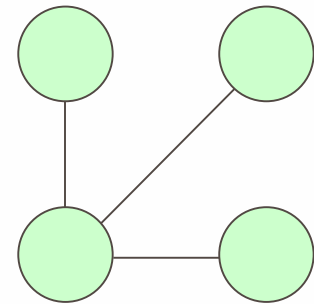
$N=4$



$N=3$



$N=3$



$N=3$

Graph 的一些性質

- 計算Degree 的方法

- 令G為一具有m邊的圖，則

$$\sum_{v \in G} \deg(v) = 2m$$

- 內連/外連Degree 的關係!!

- 令G為一具有m邊的有向圖，則

$$\sum_{v \in G} \text{in deg}(v) = \sum_{v \in G} \text{out deg}(v) = m$$

Graph 的一些性質

- 量化Edge 數量與Vertex 數量之間的關係

- 令G為一具有n 個頂點和m 個邊的簡單圖。

- 若G為無向圖，則

$$m \leq n(n-1)/2$$

- 若G 為有向圖，則

$$m \leq n(n-1)$$

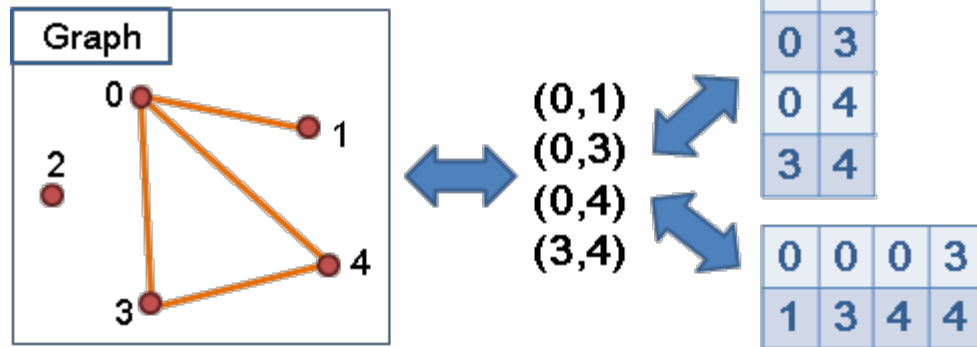
Graph 的一些性質

- 確認 Graph, tree, forest 之間的關係
- 圖 G 為一有 n 個頂點與 m 個邊的無向圖，則
 - 若 G 為連通圖，則 $m \geq n - 1$
 - 若 G 為樹，則 $m = n - 1$
 - 若 G 為森林，則 $m \leq n - 1$

圖形表示法

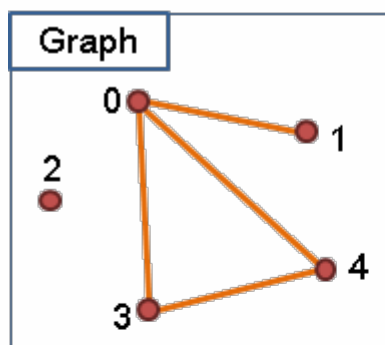
■ 邊的清單(Edge list)

- 兩兩一對的當成儲存空間
- 最省空間!!
 - 不利於計算

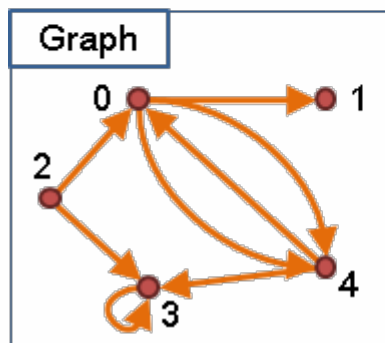


圖形的表示法

- 對於圖形結構，要怎麼儲存？
 - 鄰接矩陣(Adjacency matrix) 表示法是一種



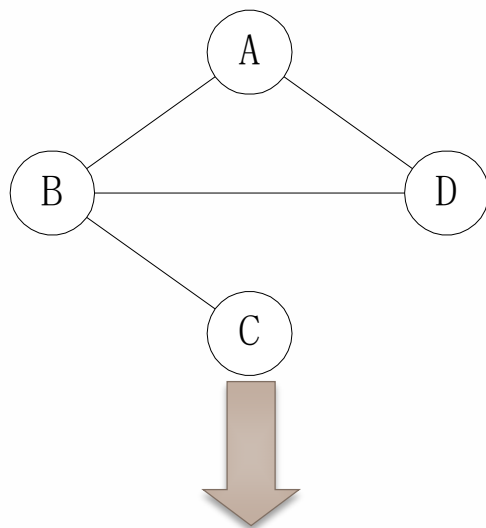
	0	1	2	3	4
0	0	1	0	1	1
1	1	0	0	0	0
2	0	0	0	0	0
3	1	0	0	0	1
4	1	0	0	1	0



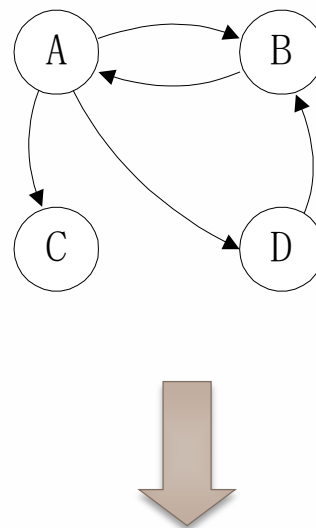
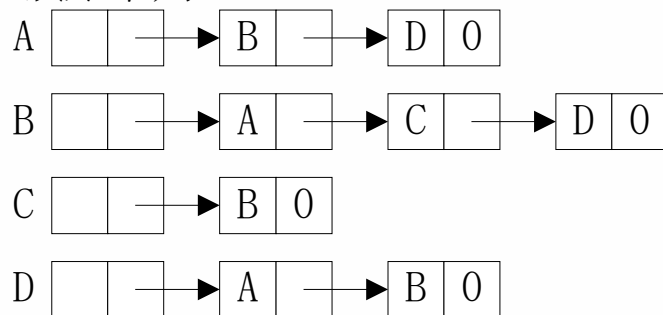
	0	1	2	3	4
0	0	1	0	0	2
1	0	0	0	0	0
2	1	0	0	1	0
3	0	0	0	1	0
4	1	0	0	1	0

圖形的表示法

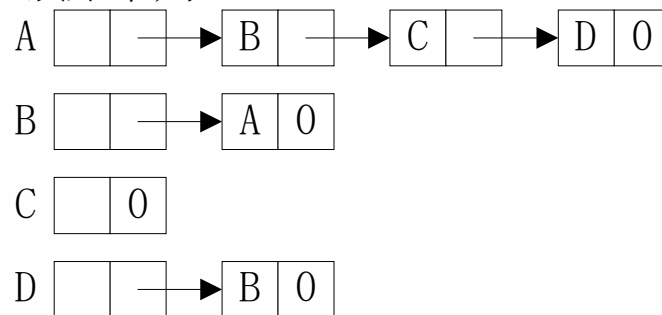
■ 鄰接串列(Adjacency list) 表示法



頂點串列

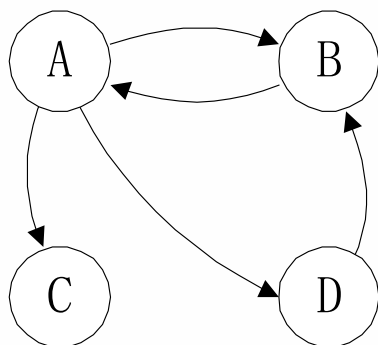


頂點串列



圖形的表示法

■ 鄰接串列與反鄰接串列

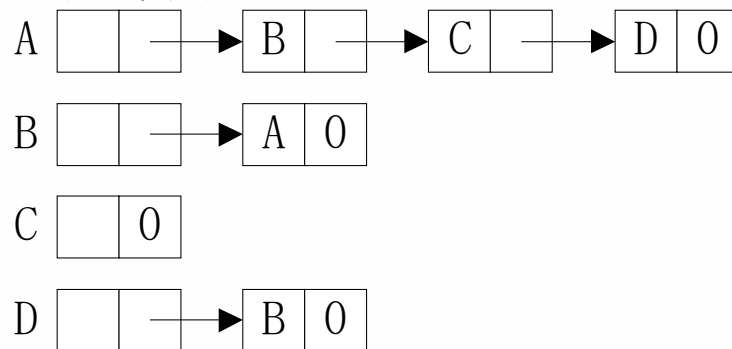


連接串列

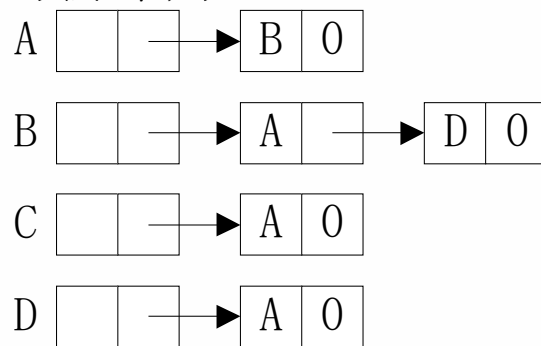
反連接串列

看的箭頭方向相反!!

頂點串列

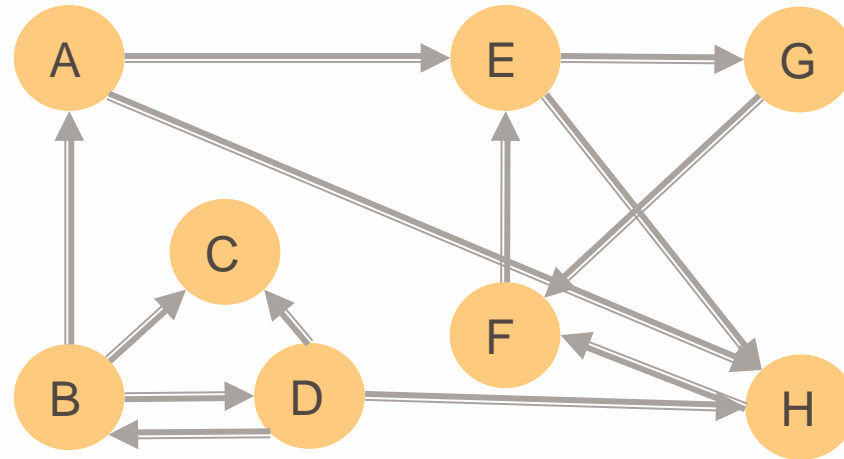


頂點串列



圖形表示法

- 複習一下兩種表示法-- 各有優缺點



相鄰矩陣
表示法

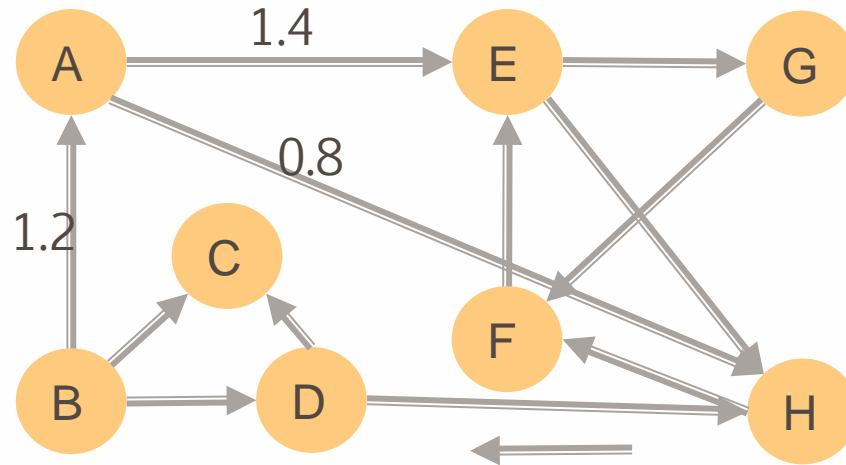
	A	B	C	D	E	F	G	H
A	0	0	0	0	1	0	0	1
B	1	0	1	1	0	0	0	0
C	0	0	0	0	0	0	0	0
D	0	1	1	0	0	0	0	1
E	0	0	0	0	0	0	1	1
F	0	0	0	0	1	0	0	0
G	0	0	0	0	0	1	0	0
H	0	0	0	0	0	1	0	0

邊的清單
表示法

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

相鄰圖形表示法加強板

- 萬一Edge表示Vertices之間的關係呢？

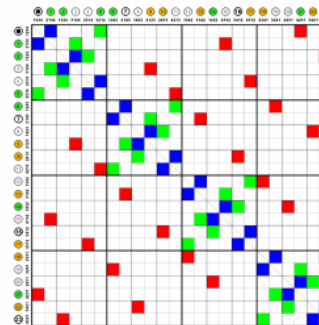


相鄰矩陣
表示法

	A	B	C	D	E	F	G	H
A	0	0	0	0	1.4	0	0	0.8
B	1.2	0	1	1	0	0	0	0
C	0	0	0	0	0	0	0	0
D	0	1	1	0	0	0	0	1
E	0	0	0	0	0	0	1	1
F	0	0	0	0	1	0	0	0
G	0	0	0	0	0	1	0	0
H	0	0	0	0	0	1	0	0

圖形結構的精髓

- 一般圖形結構僅僅只要利用相鄰矩陣就可以儲存
 - 跟一般陣列沒什麼兩樣?!!
 - 沒錯!!
 - 重點在於這樣的結構，有很多漂亮的演算法可以用
 - 搜尋路徑(Graph traversal)
 - 最短路徑搜尋(Shortest path)
 - 找關係最相像的資料(Relationship inferring)
 - 把資料分成幾群(Clustering)
 - 把要找的某個結構從圖中找出來(Sub-graph problem)
 - 太多了...

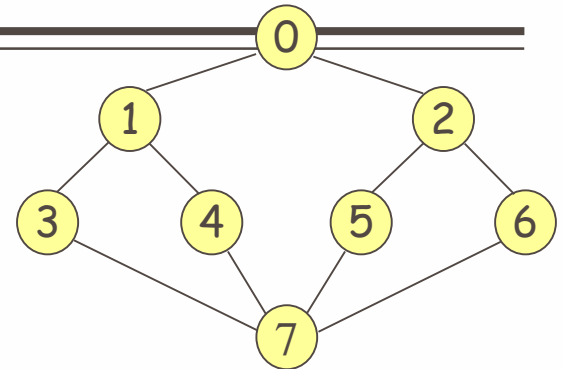


Graph Traversal 問題

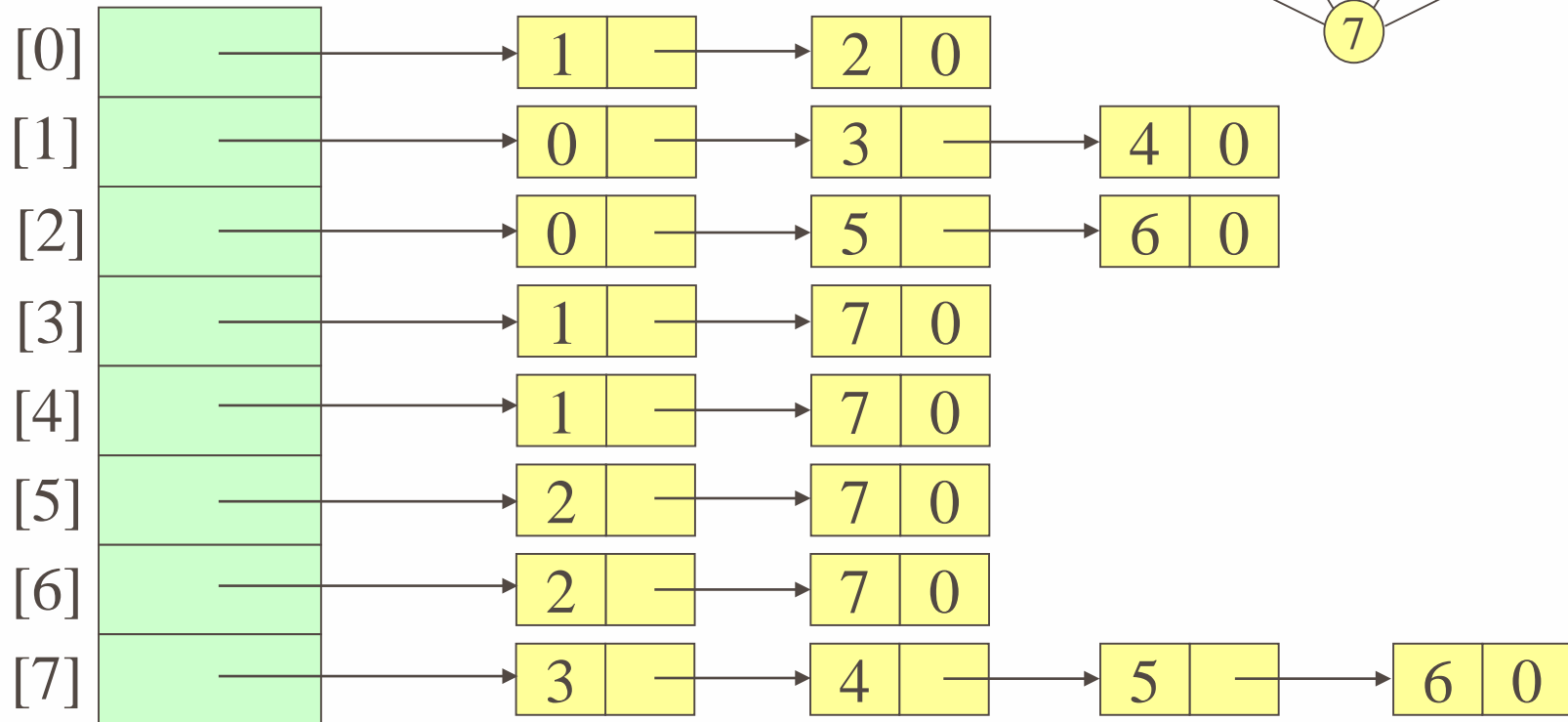
- 要有效率地把所有的Vertices都找過一遍不容易
 - 樹狀結構，很有規則，可有前中後序的找法
 - Graph?
- 普遍來說，共有兩種解法
 - Breadth-first search (BFS)
 - Depth-first search (DFS)

深度優先(Depth-First Search)

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



HeadNodes



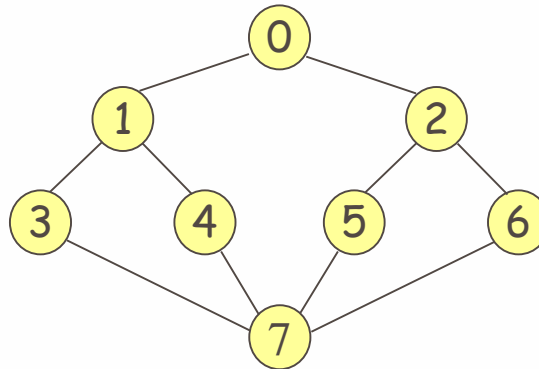
深度優先搜尋

- 由樹的根(或圖的某一點當成根) 開始
- 先到相鄰的(有一edge的意思) Vertex上，並且未搜尋過的節點上
- 所謂深度；即讓此節點所有附近的點都跑過了，再回去上一層...
- 有沒有很熟悉的感覺?!
 - 遞迴 + Stack / Queue...!!!

深度優先分解動作

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

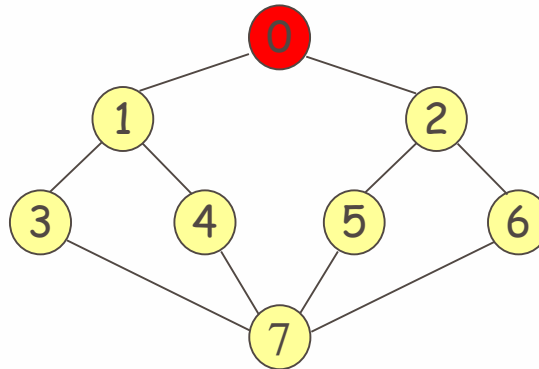
- 隨機找一個當root，先找0



深度優先分解動作

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

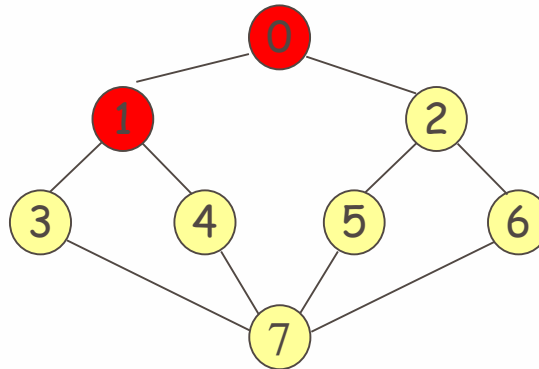
- 找相鄰的沒找過的node當下一個node! 就是1



深度優先分解動作

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

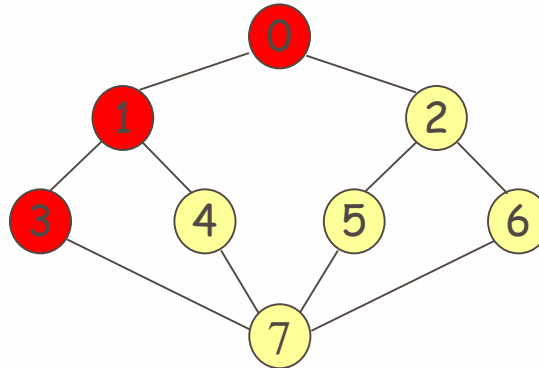
- (目前在1): 找相鄰的沒找過的node當下一個node! 就是3



深度優先分解動作

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

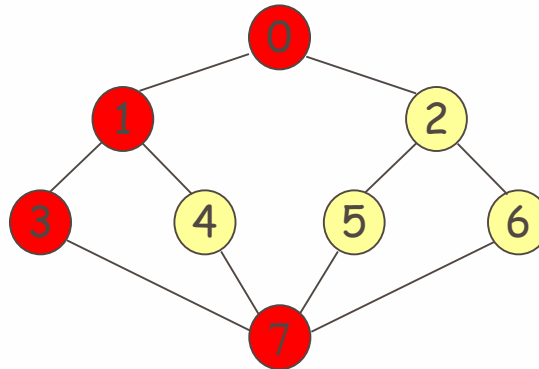
- (目前在3): 找相鄰的沒找過的node當下一個node! 就是7



深度優先分解動作

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

- (目前在7): 找相鄰的沒找過的node當下一個node! 就是4

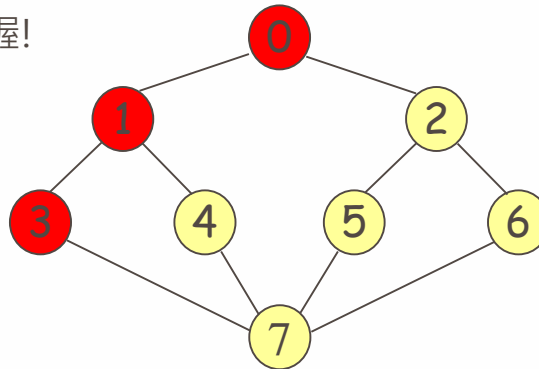


深度優先分解動作

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

■(目前在4): 找相鄰的沒找過的node當下一個node!

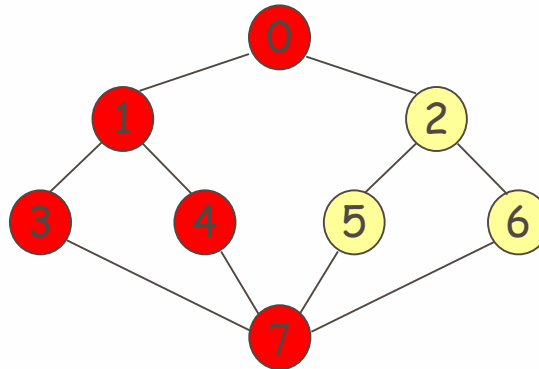
- 沒有這回事!! 都看過了喔!
- 回上一層，所以回到7



深度優先分解動作

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

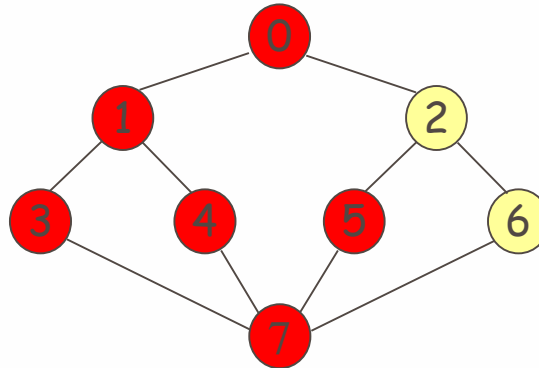
- (目前在7): 找相鄰的沒找過的node當下一個node! 就是5
 - 4已經走訪過了



深度優先分解動作

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

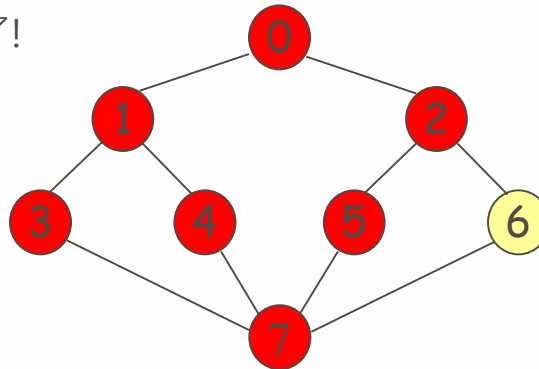
- (目前在5): 找相鄰的沒找過的node當下一個node! 就是2



深度優先分解動作

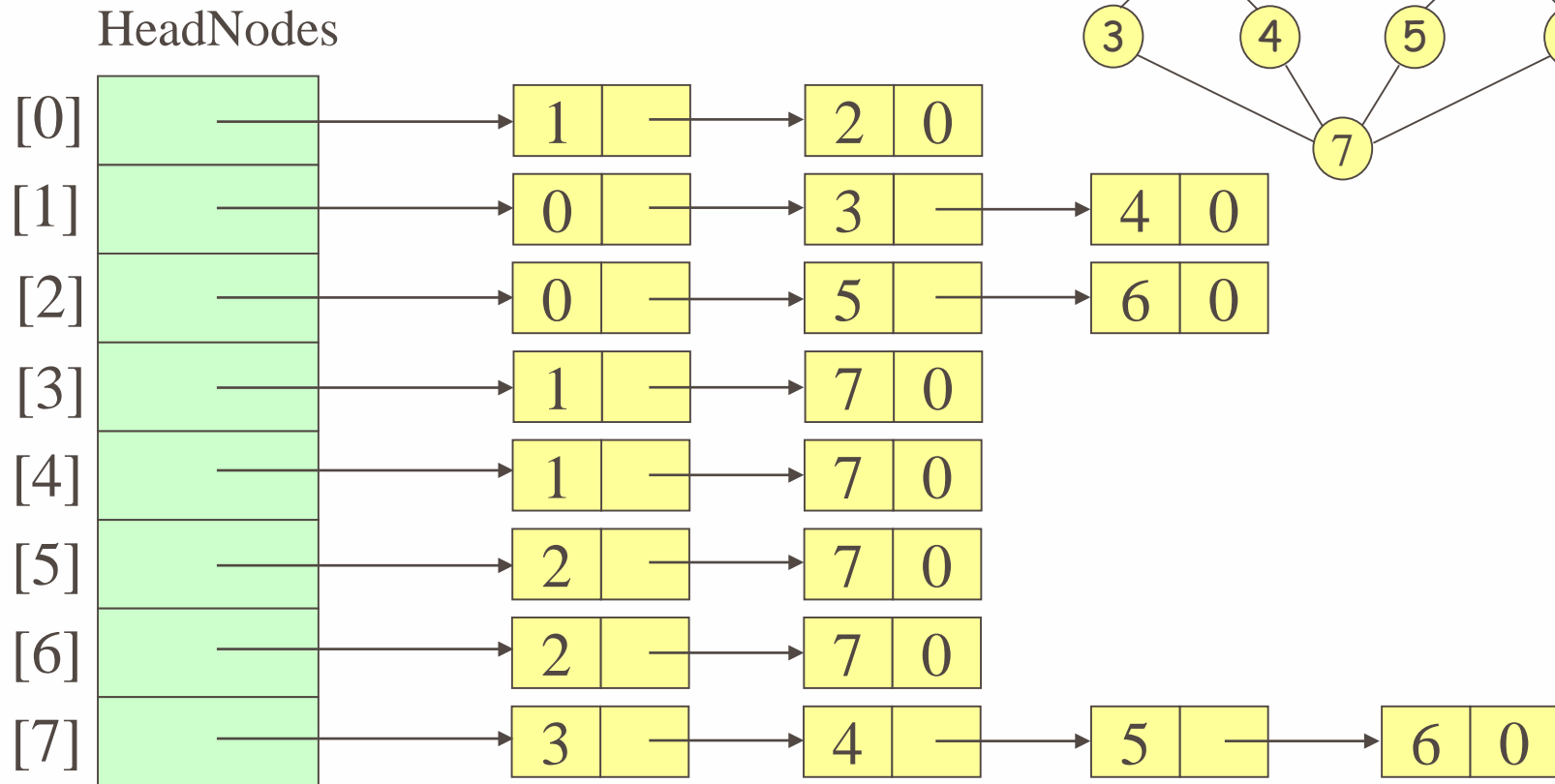
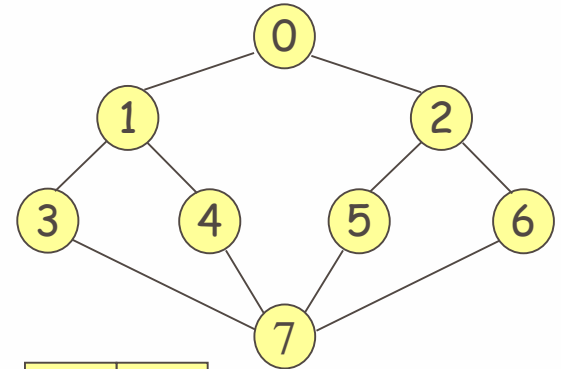
0,1,3,7,4,5,2,6

- (目前在2): 找相鄰的沒找過的node當下一個node! 就是6
 - 因為0一開始就被找過了!



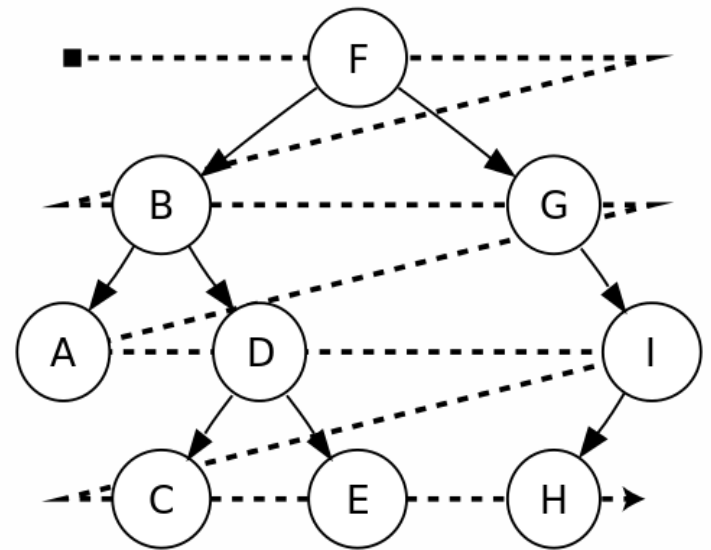
廣度優先(Breadth-First Search)

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



BFS 的方法

- 建立一個具有Vertices個數的Queue
- 照順序讀取資料進來，並根據下列規則處理
- 不斷找出尚未遍歷的點當作起點：
 - 把起點放入Queue。
 - While (queue != empty)
 - 從Queue 當中取出一點。
 - 找出跟此點相鄰且還沒看過的點，照順序存入Queue。

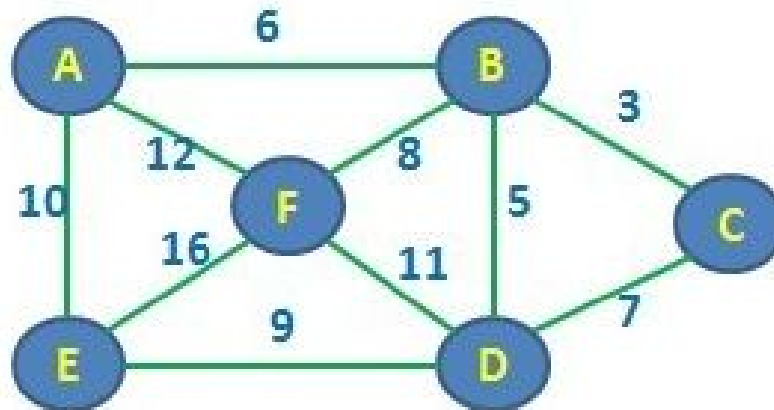


BFS 的方法(Pseudo code)

- 1. 首先準備一個佇列為 **Qu**
- 2. 再將起始頂點 v 加入到 **Qu** 之中 (亦即進行 Enqueue 動作)
- 3. 如果 **Qu** 不為空 ' 則執行下列步驟 . 否則跳到步驟 4:
 - 3.1 從 **Qu** 中取出一頂點 w (亦即進行 Dequeue 動作)
 - 3.2 並將 w 標示為『已拜訪過』
 - 3.3 將所有與 w 相鄰且尚未標示『已拜訪過』的頂點加入到 **Qu** 之中 (亦即進行 Enqueue 動作)
 - 3.4 回到步驟 3
- 4. 結束

Let's Practice

- 1. 現在Graph結構都沒有weight，請改成可以存weight的形式，並表示成陣列形式
- 2. 給定下圖，分別利用DFS與BFS計算A走到D的成本。
 - 從請改程式讓程式可以顯示出“成本”

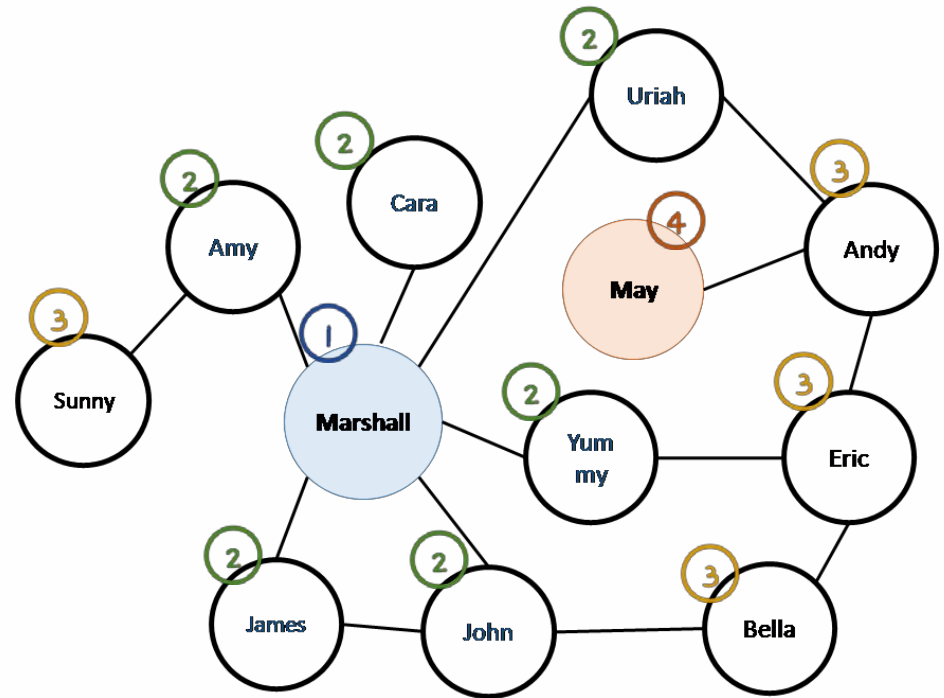




SHORTEST PATH

透過幾個人介紹可以認識？

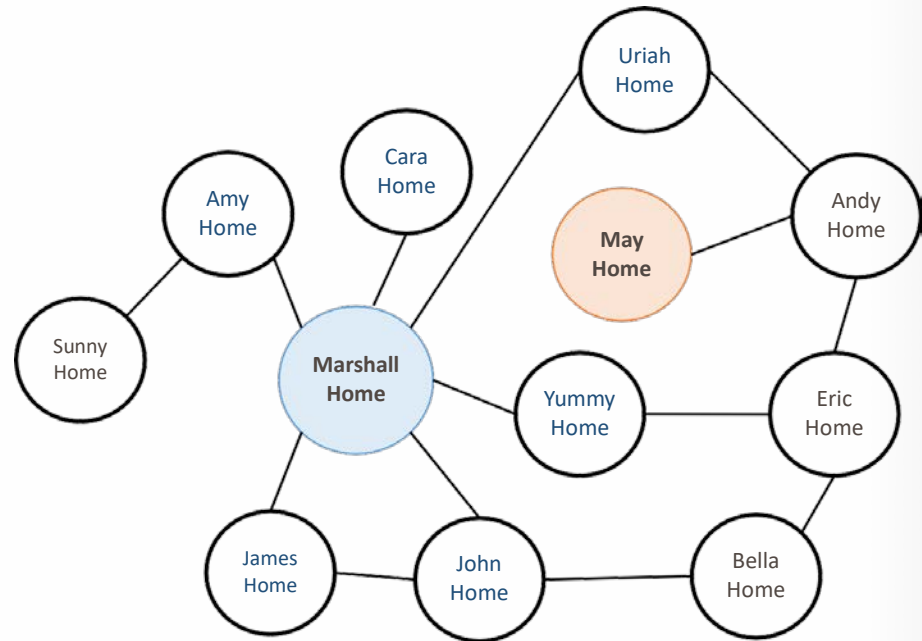
- 如果 Marshall 想要認識 May，最少要透過幾個人介紹才可以認識？
- 將 Marshall 視為起始點，執行廣度優先搜尋BFS，就可以知道這兩點的**最短路徑**！



最短路徑：Marshall -> Uriah -> Andy -> May

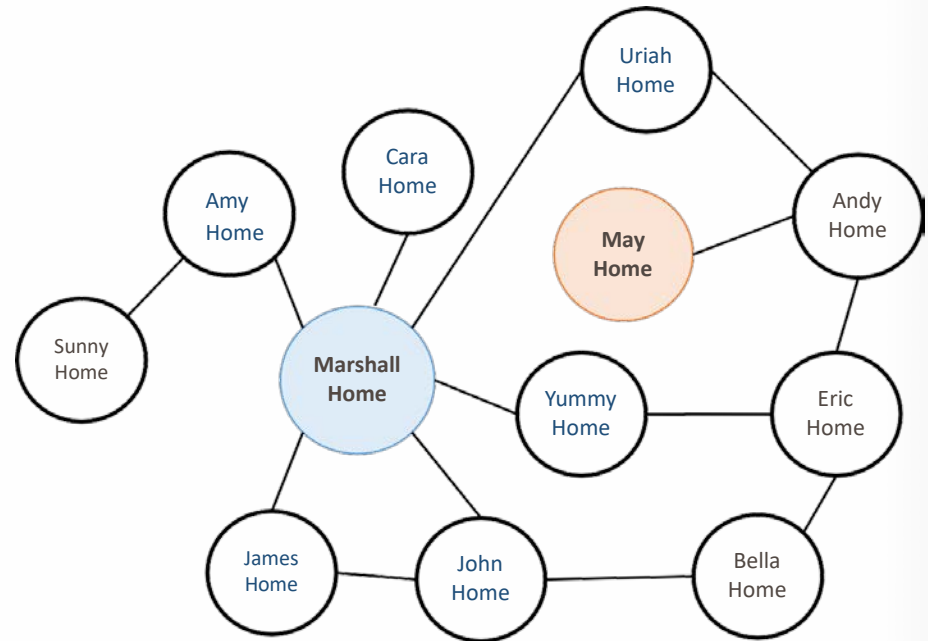
如果將人際關係圖改成交通路線圖呢？

- Marshall Home -> Uriah Home-> Andy Home -> May Home 仍然會最快嗎？
- 當圖代表的是交通路線時，我們會想知道的是任兩個點之間如何最快到達，至於中間是否能經過最少點就不是那麼重要了！



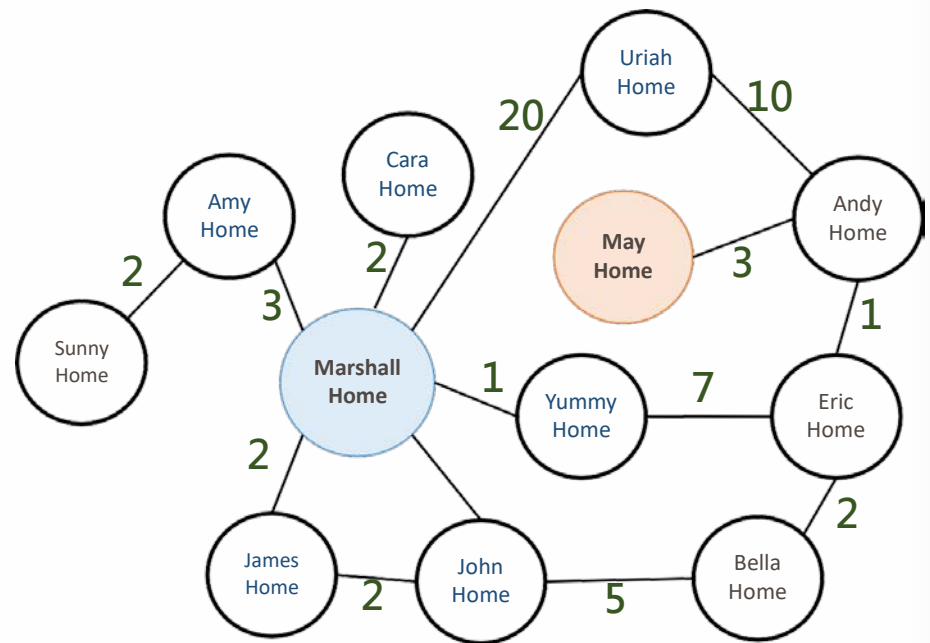
加權圖形Weighted Graph

- 使用**權重** weight 代表相鄰兩點之間的距離感。
- 有權重的圖稱為**加權圖形** **Weighted Graph**
- 依照圖的種類，權重可以代表：
 - 兩點之間相距幾公里。
 - 兩點之間相距幾分鐘。
 - 兩點之間相差多少錢。
 -



權重weight

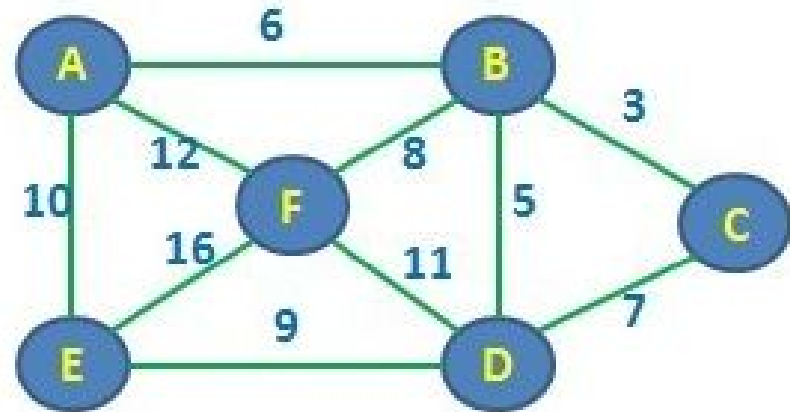
- 當邊上沒有任何權重值時，預設每邊的權重值就是 1。
- 當每邊權重值是 1 時，經過最少點的路徑就會是最短路徑。
- 如果邊上有權重值，那最短路徑又該如何取得呢？



以上周範例來看

- 先計算出Adjacency matrix

	A	B	C	D	E	F
A		6	0	0	10	12
B	6		3	5	0	8
C	0	3		7	0	0
D	0	5	7		9	11
E	10	0	0	9		16
F	12	8	0	11	16	



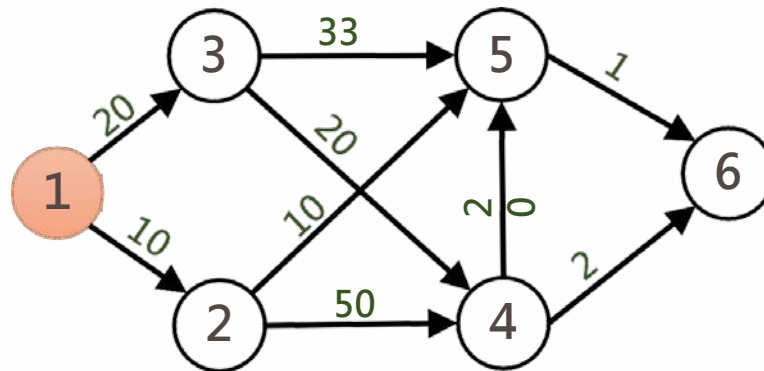
- 令A-F為 Gnode 的1-6，改為
N*3 陣列



SEARCH ALGORITHM: DIJKSTRA

Dijkstra 演算法

- Dijkstra 演算法是用來處理單源最短路徑問題：計算圖上某一點到其他所有點的最短路徑。



Dijkstra 演算法— 概念

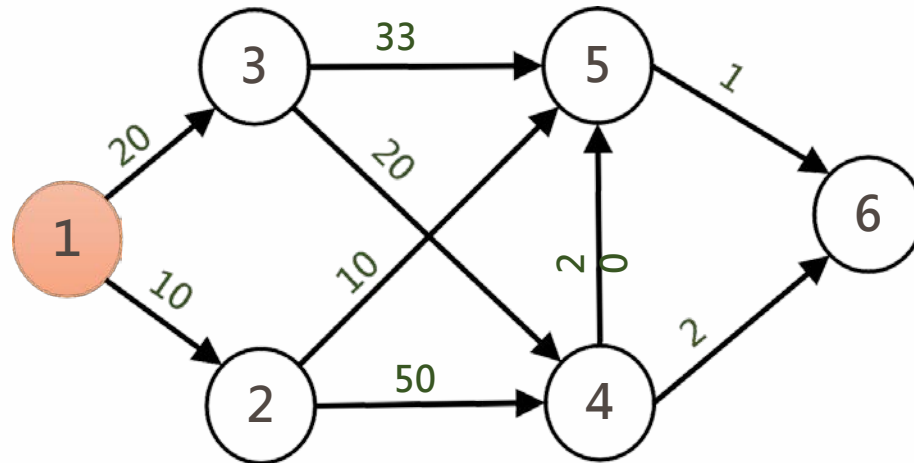
- 任兩點之間的最短路徑有兩種情況：
 - 此兩點之間有邊相連，是相鄰的兩點
 - 此兩點會經過其它點，產生最短路徑
- 若是相鄰兩點產生的最短路徑，只要列出所有的邊就可得到答案
- 若是兩點會經過其它點，產生最短路徑，就需要先找出指定點與另一點的最短路徑，畢竟只有最短路徑才能讓另一個路徑也是最短的！

Dijkstra 演算法— 流程

- 步驟 1：列出初始距離
- 步驟 2：設定兩個節點集合：
 - 不確定節點集合: 除了指定節點外的所有節點
 - 確定節點集合: 空集合
- 步驟 3：從不確定節點集合裡面找出有最短路徑的節點移到確定節點集合
- 步驟 4：反覆執行步驟 3，直到所有節點都放到確定節點集合。

Dijkstra 演算法-範例

- 請找出節點 1 到其它各節點的最短路徑



Dijkstra 演算法-步驟1

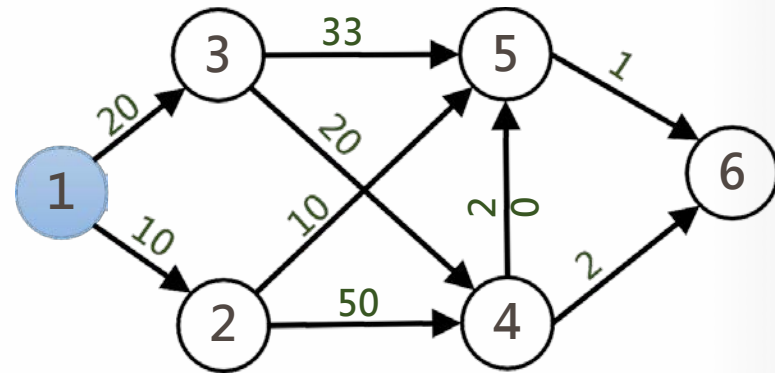
- 步驟 1：先列出指定點(節點1)到其它點的初始距離

節點 1 只有跟節點2與節點3相鄰，
因此只有這兩點有初始路程，
其它都先將路程設為無限大

	1	2	3	4	5	6
距離	0	10	20	∞	∞	∞

自己到自己的
距離設為 0

與指定點沒有相鄰的點，
都先將距離設為無限大。



Dijkstra 演算法- 步驟2 <第1個確定點>

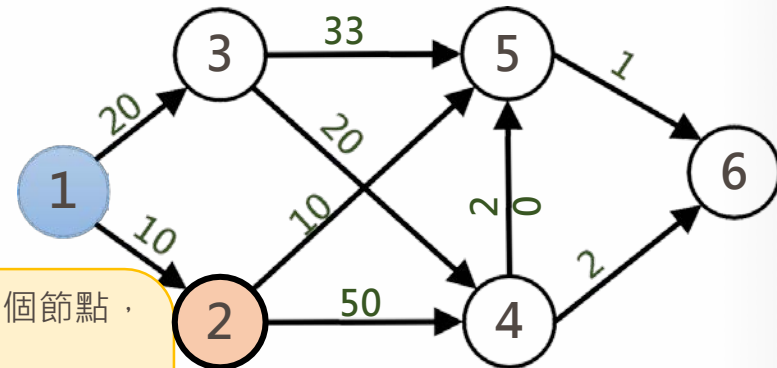
- 步驟 2：在初始距離中**找出距離是最短的點做為第 1 個確定點**，之後此點的最短路徑就不會再被更動。
 - 如果經過的距離不是最短距離，那後續一定無法產生最短距離，因此必須先從初使距離中決定一個點的最短路徑。

	1	2	3	4	5	6
距離	0	10	20	∞	∞	∞

•節點 1 只有跟節點 2 與節點 3 相鄰，所以之後不論節點 1 是要到哪個節點，一定是先經過這兩點的其中一點。

•節點 1 到節點 2 的距離是 10，節點 1 到節點 3 的距離是 20， $10 < 20$ ，此時就可以得到節點 1 到節點 2 的最短路徑必定是 10。

•若是先通過節點 3，那距離至少會大於 20，所以是不可能的！

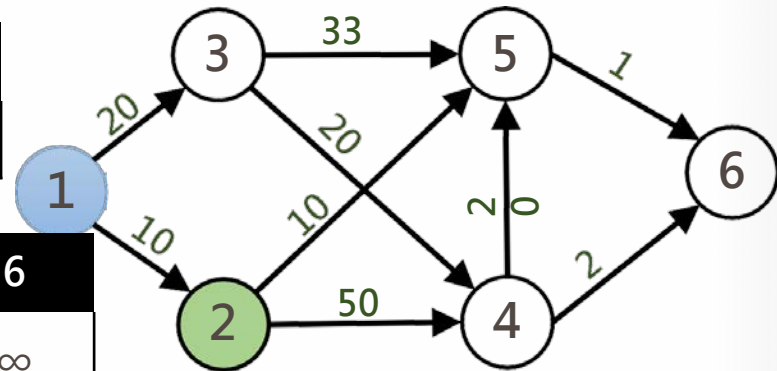


Dijkstra 演算法- 步驟3 <第2個確定點>

- 步驟 3：在剩下尚未決定最短距離的節點中，判斷若是經過已經確定最短路徑的節點 2，是否可降低各點的最短路徑，若有降低，就更改該點的最短距離值，若沒有，就維持原距離值。並在這些點中找出距離最短的頂點最為**第 2 個確定點**。
 - 同樣的，第 2 個確定點產生的原因也是因為如果經過的距離不是最短距離，那後續一定無法產生最短距離，因此必須先從已知的距離中找出一個最短距離。

	1	2	3	4	5	6
原距離	0	10	20	∞	∞	∞

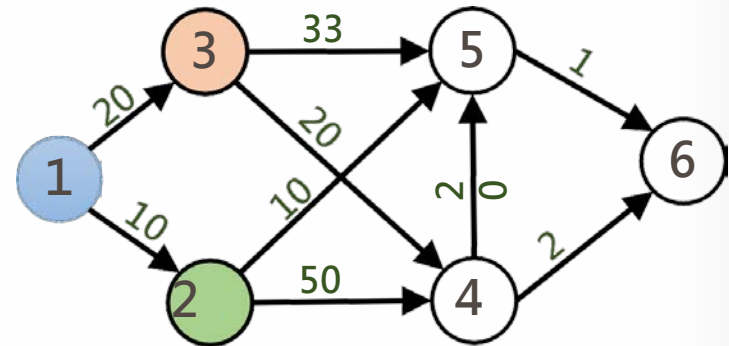
	1	2	3	4	5	6
經過節點 2 的新距離	0	10	20	60	20	∞



Dijkstra 演算法– 步驟3 <第2個確定點>

- 從節點 1 經過節點 2 可以讓節點 4 與節點 5 的距離降低，所以就先這兩個節點的距離值更新。
- 節點 3、節點 4、節點 5、節點 6 中距離最短的是節點 3 與節點 5，選擇節點 3 最為第 2 個確定點。
 - 當有兩個節點的距離同時都是最低的值時，可任意選一點。

	1	2	3	4	5	6
距離	0	10	20	60	20	∞

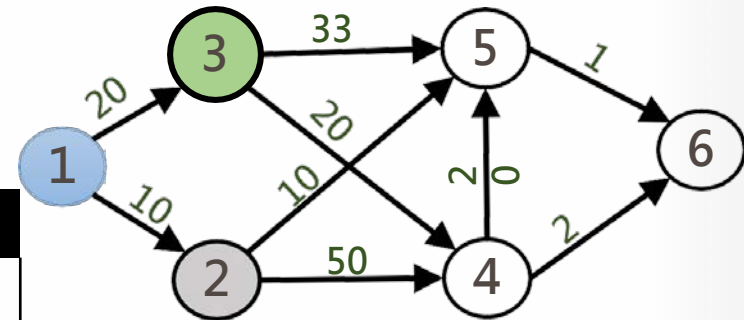


Dijkstra 演算法- 步驟4<第3個確定點>

- 步驟 4：在剩下尚未決定最短距離的節點中，判斷若是經過前一個步驟產生的確定點(節點3)，是否可降低各點的最短路徑，若有降低，就更該點的最短距離值，若沒有，就維持原距離值。並在這些點中找出距離最短的頂點最為**第 3 個確定點**。
 - 同樣的，第 3 個確定點產生的原因也是因為如果經過的距離不是最短距離，那後續一定無法產生最短距離，因此必須先從已知的距離中找出一個最短距離。

	1	2	3	4	5	6
距離	0	10	20	60	20	∞

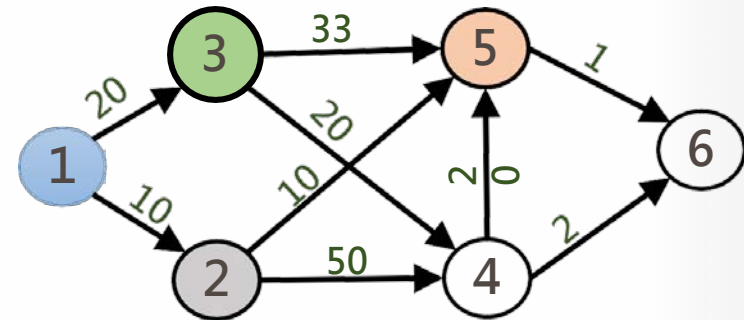
	1	2	3	4	5	6
經過節點 3 的新 距離	0	10	20	40	20	∞



Dijkstra 演算法- 步驟4 <第3個確定點>

- 從節點 1 經過節點 3 可以讓節點 4 的距離降低，所以要將節點4 的距離值更新。
- 節點 4、節點 5、節點 6中距離最短的是節點5，選擇**節點 5** 最為**第 3 個確定點**。

	1	2	3	4	5	6
距離	0	10	20	40	20	∞

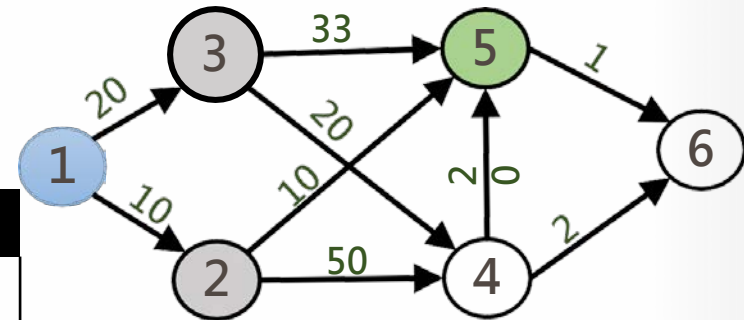


Dijkstra 演算法- 步驟5 <第4個確定點>

- 步驟 5：在剩下尚未決定最短距離的節點中，判斷若是經過前一個步驟產生的確定點(節點5)，是否可降低各點的最短路徑，若有降低，就更改該點的最短距離值，若沒有，就維持原距離值。並在這些點中找出距離最短的頂點最為**第 4 個確定點**。
 - 同樣的，第 4 個確定點產生的原因也是因為如果經過的距離不是最短距離，那後續一定無法產生最短距離，因此必須先從已知的距離中找出一個最短距離。

	1	2	3	4	5	6
距離	0	10	20	40	20	∞

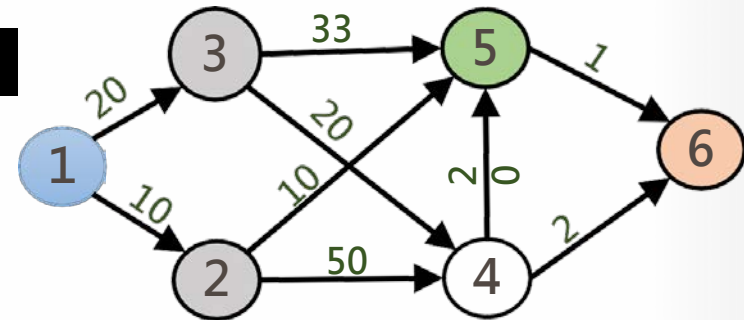
	1	2	3	4	5	6
經過節點 5 的新距離	0	10	20	40	20	21



Dijkstra 演算法- 步驟5 <第4個確定點>

- 從節點 1 經過節點 5 可以讓節點 6 的距離降低，所以要將節點6的距離值更新。
- 節點 4、節點 6中距離最短的是節點6，選擇節點 6 最為**第 4 個確定點**。

	1	2	3	4	5	6
距離	0	10	20	40	20	21

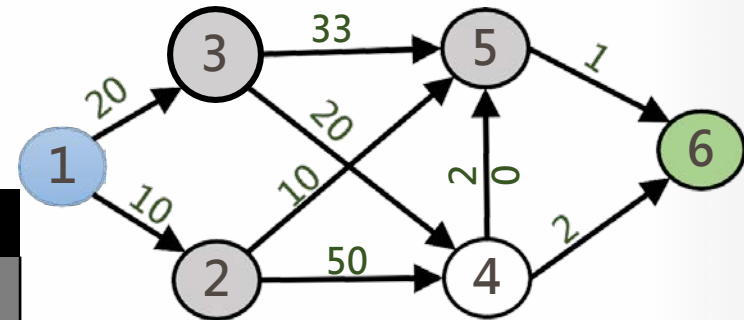


Dijkstra 演算法- 步驟6 <第5個確定點>

- 步驟 6：最後只剩下節點 4 尚未確定最短路徑，因此只要計算節點4如果經過節點6是否可以降低距離，有降低就更新記錄值。不論是否有降低，節點 4 一定會是**第 5 個確定點**。

	1	2	3	4	5	6
距離	0	10	20	40	20	21

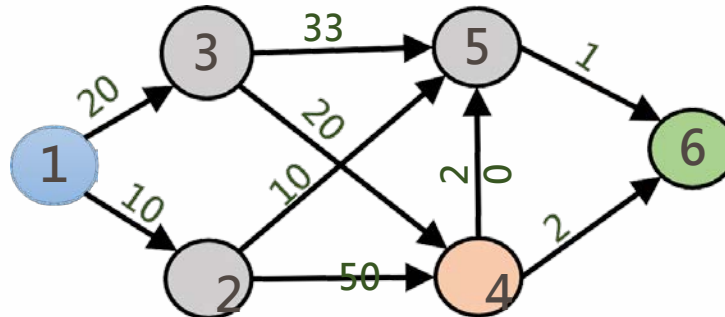
	1	2	3	4	5	6
經過節點 6 的新距離	0	10	20	40	20	21



Dijkstra 演算法- 步驟6 <第5個確定點>

- 節點 4 成為第 5 個確定點。

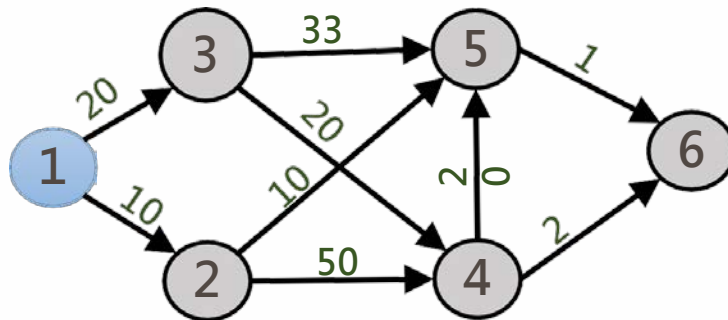
	1	2	3	4	5	6
距離	0	10	20	40	20	21



Dijkstra 演算法- 步驟7 <完成>

- 步驟 7：已經完成節點 1 到各節點的最短路徑，最後只要根據需求回傳所需要的路徑長度或是路徑資料就完成了！

	1	2	3	4	5	6
距離	0	10	20	40	20	21



Dijkstra

Dijkstra 實作

若是使用相鄰矩陣記錄圖型結構，
那矩陣內指定節點對應的那行就會是初始距離。

若仍有不確定節點，就會持續執行

找出不確定集合裡面最短的距離

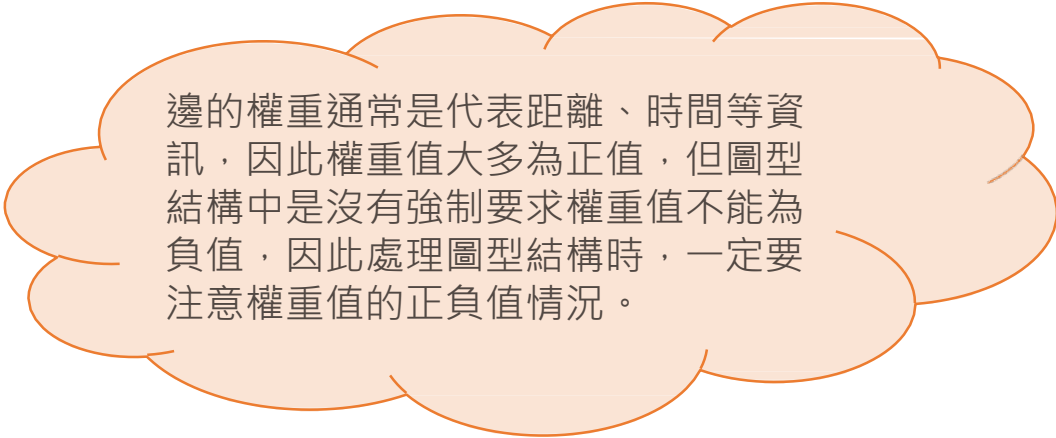
找出下一個要從不確定節點集合移到
確定節點集合的結點。也就是找出可以
經過剛剛確定的最短
距離點而縮點距離的點

```
void Dijkstra(int vertex, int verticesCount, int graph[][MAX_VERTICES])
{
    int distance[MAX_VERTICES], done[MAX_VERTICES];
    int i, lastVertex, doneVerticesCount, shortestLength;
    for (i = 0; i < verticesCount; i++) {
        distance[i] = graph[vertex][i];
        done[i] = 0;
    }
    done[vertex] = 1;
    doneVerticesCount = 1;
    while(doneVerticesCount < verticesCount) {
        shortestLength = 99999;
        for (i = 0; i < verticesCount; i++) {
            if (done[i] == 1) continue;
            if (distance[i] < shortestLength) {
                shortestLength = distance[i];
                lastVertex = i;
            }
        }
        done[lastVertex] = 1;
        doneVerticesCount++;
        for (i = 0; i < verticesCount; i++) {
            if (done[i] == 1) continue;
            if (distance[i] > distance[lastVertex] + graph[lastVertex][i])
                distance[i] = distance[lastVertex] + graph[lastVertex][i];
        }
    }
    displayDistance(verticesCount, distance);
}
```

done 陣列代表的是確定節點集合
doneVerticesCount則是記錄確定節點數量

Dijkstra 演算法

- Dijkstra 演算法的主軸思想就是只有前面是最短路徑，後面才會是最短路徑，因此處理的圖絕對**不能有負權重邊**！



邊的權重通常是代表距離、時間等資訊，因此權重值大多為正值，但圖型結構中是沒有強制要求權重值不能為負值，因此處理圖型結構時，一定要注意權重值的正負值情況。

問題來了...

如果邊有負權重，
要怎麼找最短路徑呢？

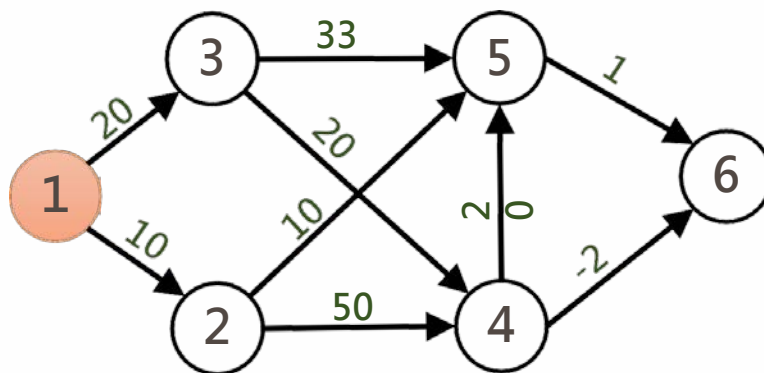




BELLMAN-FORD

Bellman-Ford 演算法

- Bellman-Ford 演算法是用來處理單源最短路徑問題：計算圖上某一點到其他所有點的最短路徑。



Bellman-Ford 演算法— 概念

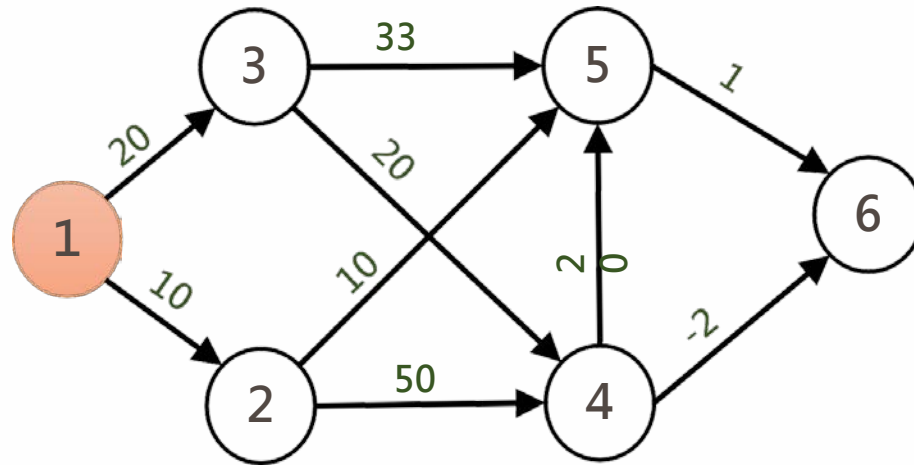
- 若在任兩節點之間的可能路徑中加入一個邊，可以讓此兩點的路徑距離降低，那就將這個邊加到此兩點的距離中，反之，就忽略這個邊。
- 只要每個頂點都各別考慮過加入各個邊後是否會造成影響，就可以產生最短路徑。
 - 若有 N 個頂點， M 條邊，最多執行 $N * M$ 的路徑距離大小判斷就可以 得到結果。
 - 但是，若 M 條邊的任一條邊加入後都沒有路徑會更改距離，那就可以提 早結束判斷。

Bellman-Ford 演算法— 流程

- 步驟 1：將指定節點到其他任一個節點的距離長度設為無限大。
- 步驟 2：依序取出每一條邊，找出因為這一條邊而可以降低最短距離的兩節點。
- 步驟 3：為了避免步驟 2 中有任兩節點因為加入新邊而降低距離，會間接對
其它點的最短路徑造成影響，因此需要重複執行步驟 2，直到所有邊都有對 每一個節點檢查。
- 步驟 4：除了完成所有邊對每一個節點檢查以外，如果加入任何一條邊都不會更改任一點的最短路徑時，就可以結束。

Bellman-Ford 演算法— 範例

- 請找出節點 1 到其它各節點的最短路徑

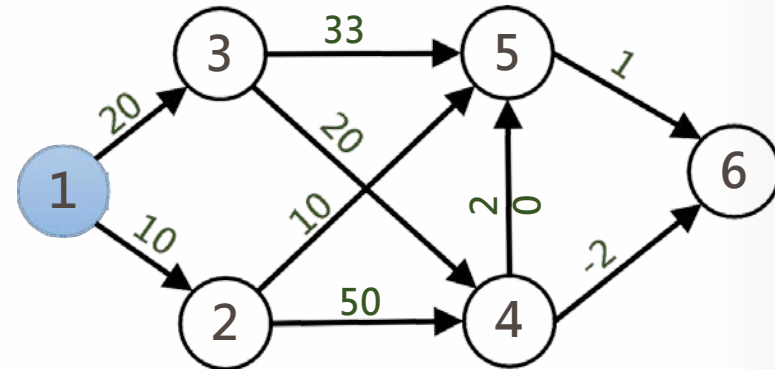


Bellman-Ford 演算法- 步驟1

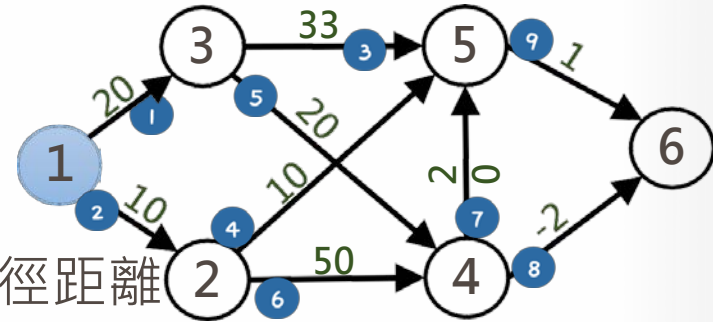
- 步驟 1：先將指定點(節點1)到其它點的初始距離都設為無限大

	1	2	3	4	5	6
距離	0	∞	∞	∞	∞	∞

自己到自己的
距離設為 0



Bellman-Ford 演算法- 步驟2(1)



- 加入第 1 條邊：降低節點 1 到節點 3 的路徑距離

	1	2	3	4	5	6
距離	0	∞	20	∞	∞	∞

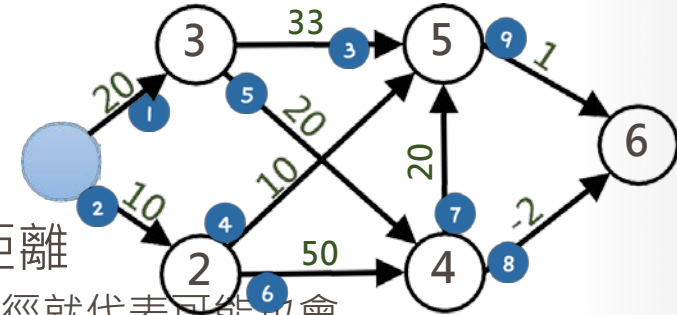


- 加入第 2 條邊：降低節點 1 到節點 2 的路徑距離

	1	2	3	4	5	6
距離	0	10	20	∞	∞	∞



Bellman-Ford 演算法– 步驟 2



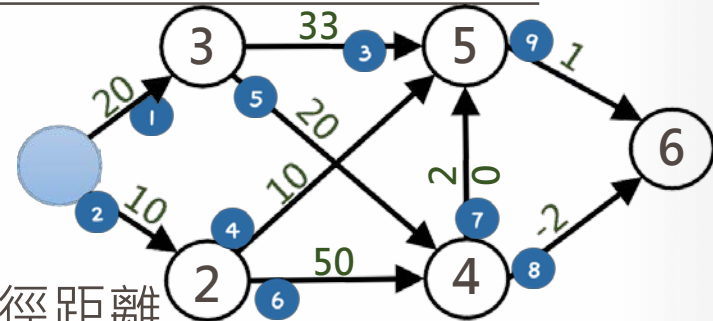
- 加入第 3 條邊：降低節點 3 到節點 5 的路徑距離
 - 節點 1 可到節點 3，因此更新節點 3 到節點 5 的最短路徑就代表可能也會更新節點 1 到節點 5 的最短路徑

	1	2	3	4	5	6
距離	0	10	20	∞	53	∞

- 加入第 4 條邊：降低節點 2 到節點 5 的路徑距離
 - 節點 1 可到節點 2，因此更新節點 2 到節點 5 的最短路徑就代表可能也會更新節點 1 到節點 5 的最短路徑

	1	2	3	4	5	6
距離	0	10	20	∞	20	∞

Bellman-Ford 演算法- 步驟2(3)



- 加入第 5 條邊：降低節點 3 到節點 4 的路徑距離

	1	2	3	4	5	6
距離	0	10	20	40	20	∞

- 加入第 6 條邊：降低節點 2 到節點 2 的路徑距離

	1	2	3	4	5	6
距離	0	10	20	40	20	60

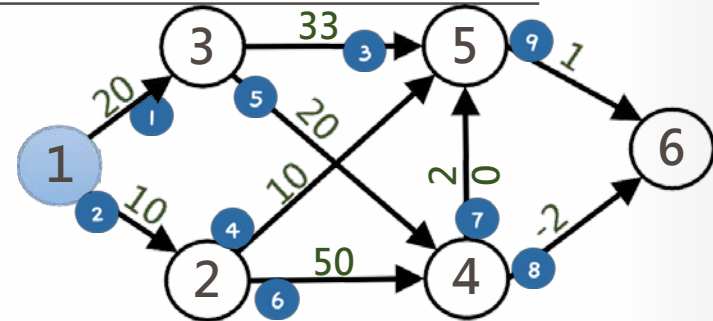
Bellman-Ford 演算法- 步驟2(4)

- 加入第 7 條邊：沒有影響任兩點的距離，不用更改記錄值

	1	2	3	4	5	6
距離	0	10	20	40	20	60

- 加入第 8 條邊：降低節點 4 到節點 6 的路徑距離

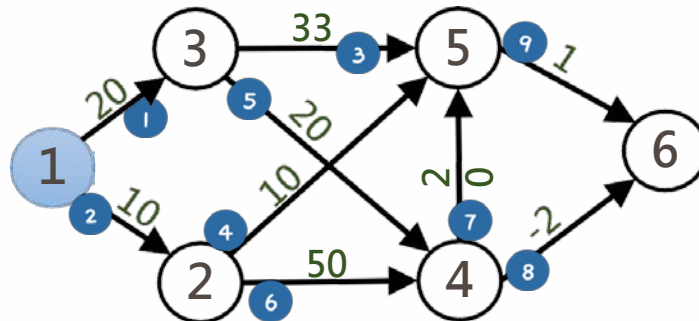
	1	2	3	4	5	6
距離	0	10	20	40	20	38



Bellman-Ford 演算法- 步驟2(5)

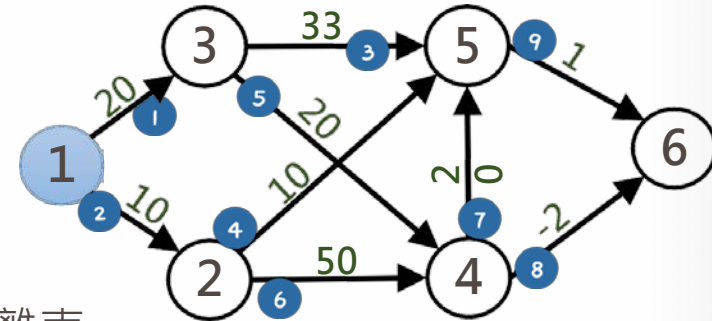
- 加入第 9 條邊：降低節點 5 到節點 6 的路徑距離

	1	2	3	4	5	6
距離	0	10	20	40	20	21



Bellman-Ford 演算法- 步驟3

- 重複步驟2，再度更新距離表
- 反覆執行，直到：
 - 全部 9 條邊不論加入哪一條都不會更改到距離表。
 - 最多執行 6 (頂點數目)次



	1	2	3	4	5	6
距離	0	10	20	40	20	21

Bellman-Ford 演算法- 實作

```
void BellmanFord(int vertex, int verticesCount, int edgesCount, int edgesList[][3])
{
    int i, j, isChange;
    int distance[MAX_VERTICES], predecessor[MAX_VERTICES];

    for (i = 0; i < verticesCount; i++) {
        distance[i] = 99999;
        predecessor[i] = i;
    }
    distance[vertex] = 0;
    for (i = 0; i < verticesCount; i++) {
        isChange = 0;
        for (j = 0; j < edgesCount; j++) {
            if (distance[edgesList[j][0]] + edgesList[j][2] < distance[edgesList[j][1]]) {
                distance[edgesList[j][1]] = distance[edgesList[j][0]] + edgesList[j][2];
                predecessor[edgesList[j][1]] = edgesList[j][0];
                isChange = 1;
            }
        }
        if (isChange == 0) break;
    }
}
```

將指定點到任一點的距離初始值設為無限大

每次檢查一個邊，看加入此邊後會不會降低現有的最短距離

最多執行N (頂點數) 次，但如果加入任一邊都不會有變化的話，就不需要再反覆檢查了。

Bellman-Ford 演算法- 檢查是否有負迴圈

Bellman-Ford

```
isChange = 0;
for (j = 0; j < edgesCount; j++) {
    if (distance[edgesList[j][0]] + edgesList[j][2] < distance[edgesList[j][1]]) {
        distance[edgesList[j][1]] = distance[edgesList[j][0]] + edgesList[j][2];
        predecessor[edgesList[j][1]] = edgesList[j][0];
        isChange = 1;
    }
}
```

如果已經執行N (頂點數) 次的各邊判斷後，再多加一次執行各邊判斷時，仍然有最短距離被更改，就代表圖內有負迴圈，才會一直造成距離變動

還是有問題...

一個圖如果有 10 個頂點，
就要執行 10 次的 Dijkstra
或 Bellman-Ford 才可以得
到圖上任兩點的最短路徑？





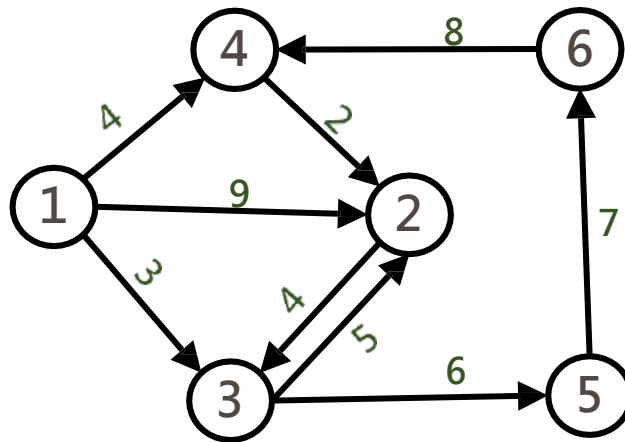
FLOYD-WARSHALL

多源最短路徑

- 指定點到其他任一點的最短路徑問題稱為**單源最短路徑問題**
 - Dijkstra, Bellman-Ford
- 任兩點的最短路徑問題稱為**多源最短路徑問題**
 - Floyd-Warshall

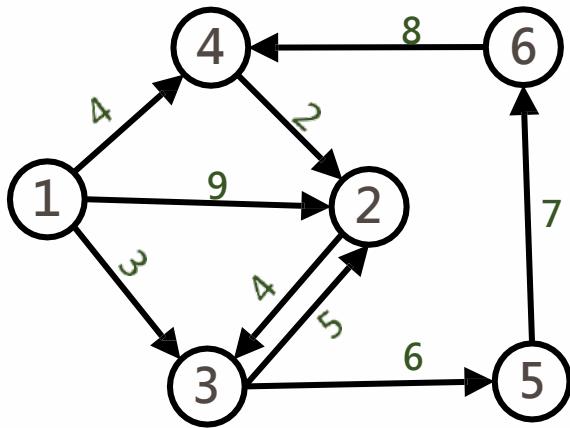
Floyd-Warshall 演算法

- Floyd-Warshall 是用來處理多源最短路徑問題：計算圖上任兩點的最短路徑。
- Floyd-Warshall 與 Dijkstra 一樣，不處理負權重的圖型問題



Floyd-Warshall 演算法– 概念

- Floyd-Warshall 與 Dijkstra 類似，都認為任兩點之間的最短路徑有兩種情況：
 - 此兩點之間有邊相連，是相鄰的兩點
 - 此兩點會經過其它點，產生最短路徑



	1	2	3	4	5	6
1	0	9	3	4	∞	∞
2	∞	0	4	∞	∞	∞
3	∞	5	0	∞	6	∞
4	∞	2	∞	0	∞	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	∞	8	∞	0

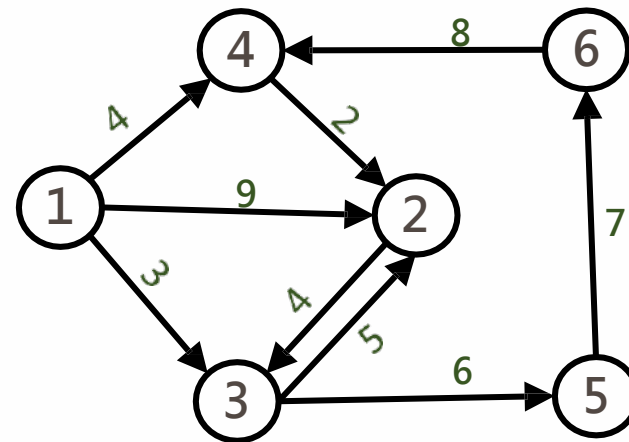
Floyd-Warshall 演算法— 流程

- 步驟 1: 複製一份圖型的相鄰矩陣資料，將這份矩陣資料做為最短路徑的初始值。
- 步驟 2: 在任兩點之間加入其它節點，確認距離是否有縮短，若有，就更新矩陣內對應的元素值，重複此步驟，直到所有節點都有嘗試放入任兩點節點之間。
- 步驟 3: 全部確認完成後，經由步驟 2 調整後的矩陣資料就是圖中任兩點的最短路徑。

Floyd-Warshall 演算法- 步驟1

- 步驟 1: 複製一份圖型的相鄰矩陣資料，將這份矩陣資料做為最短路徑的初始值。

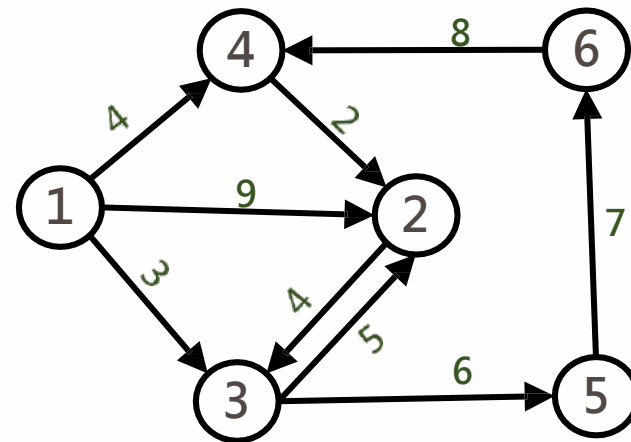
	1	2	3	4	5	6
1	0	9	3	4	∞	∞
2	∞	0	4	∞	∞	∞
3	∞	5	0	∞	6	∞
4	∞	2	∞	0	∞	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	∞	8	∞	0



Floyd-Warshall 演算法- 步驟2

- 步驟 2: 在任兩點之間加入節點 1，確認距離是否有縮短

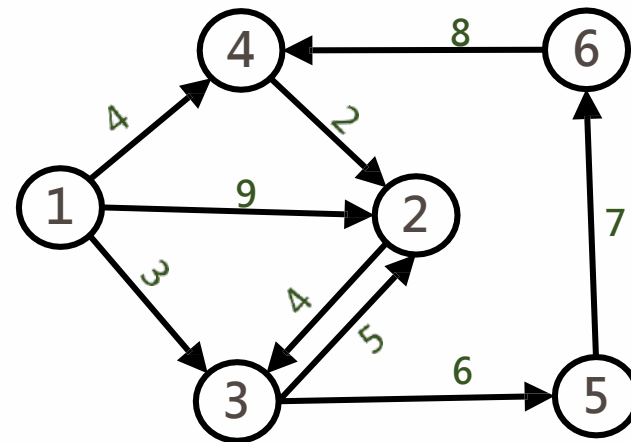
	1	2	3	4	5	6
1	0	9	3	4	∞	∞
2	∞	0	4	∞	∞	∞
3	∞	5	0	∞	6	∞
4	∞	2	∞	0	∞	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	∞	8	∞	0



Floyd-Warshall 演算法- 步驟3

- 步驟 3: 在任兩點之間加入節點 2，確認距離是否有縮短

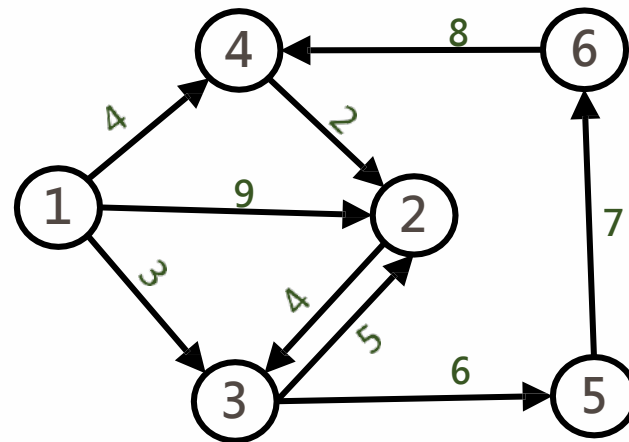
	1	2	3	4	5	6
1	0	9	3	4	∞	∞
2	∞	0	4	∞	∞	∞
3	∞	5	0	∞	6	∞
4	∞	2	6	0	∞	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	∞	8	∞	0



Floyd-Warshall 演算法- 步驟4

- 步驟 4: 在任兩點之間加入節點 3，確認距離是否有縮短

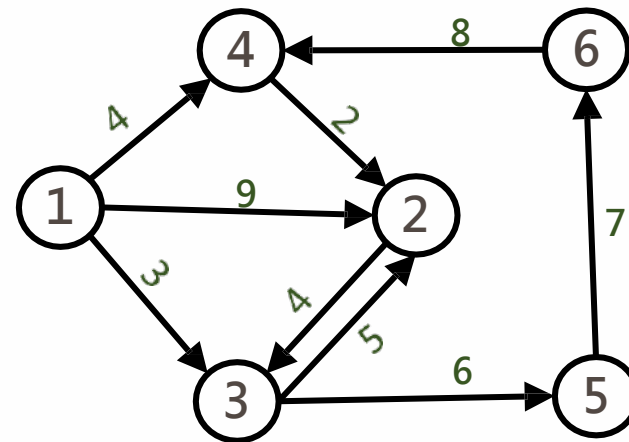
	1	2	3	4	5	6
1	0	8	3	4	9	∞
2	∞	0	4	∞	10	∞
3	∞	5	0	∞	6	∞
4	∞	2	6	0	12	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	∞	8	∞	0



Floyd-Warshall 演算法- 步驟5

- 步驟 5: 在任兩點之間加入節點 4，確認距離是否有縮短

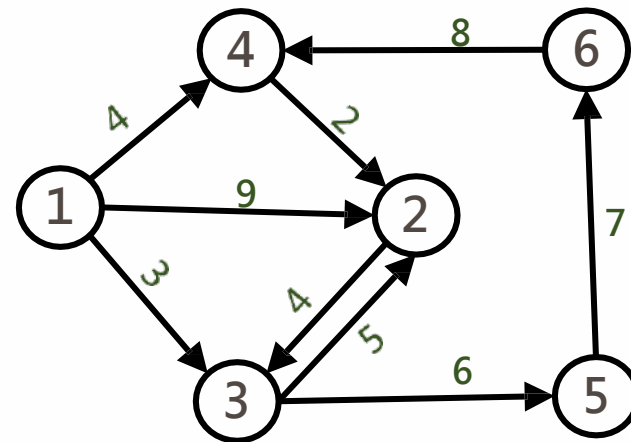
	1	2	3	4	5	6
1	0	6	3	4	9	∞
2	∞	0	4	∞	10	∞
3	∞	5	0	∞	6	∞
4	∞	2	6	0	12	∞
5	∞	∞	∞	∞	0	7
6	∞	10	14	8	20	0



Floyd-Warshall 演算法- 步驟6

- 步驟 6: 在任兩點之間加入節點 5，確認距離是否有縮短

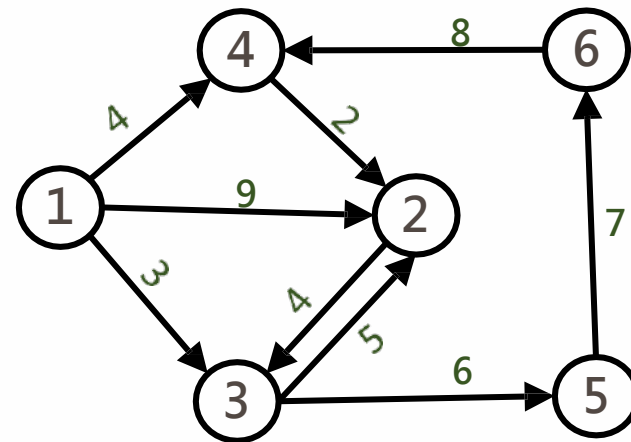
	1	2	3	4	5	6
1	0	6	3	4	9	16
2	∞	0	4	∞	10	17
3	∞	5	0	∞	6	13
4	∞	2	6	0	12	19
5	∞	∞	∞	∞	0	7
6	∞	10	14	8	20	0



Floyd-Warshall 演算法- 步驟7

- 步驟 7: 在任兩點之間加入節點 6，確認距離是否有縮短

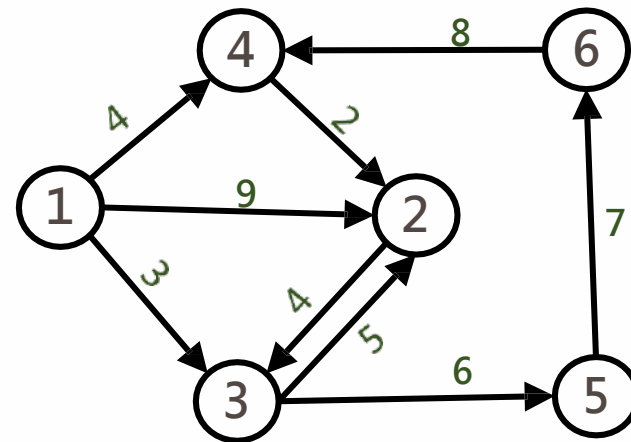
	1	2	3	4	5	6
1	0	6	3	4	9	16
2	∞	0	4	25	10	17
3	∞	5	0	21	6	13
4	∞	2	6	0	12	19
5	∞	17	21	15	0	7
6	∞	10	14	8	20	0



Floyd-Warshall 演算法— 步驟8

- 步驟 8: 全部確認完成後，最後的矩陣資料就是圖中任兩點的最短路徑。

	1	2	3	4	5	6
1	0	6	3	4	9	16
2	∞	0	4	25	10	17
3	∞	5	0	21	6	13
4	∞	2	6	0	12	19
5	∞	17	21	15	0	7
6	∞	10	14	8	20	0



Floyd-Warshall實作

使用相鄰矩陣的表示
方式記錄圖讓任兩點
的最短距離

比較任兩點之間的最短
路徑，是否會因為多經
過某節點就減少距離。

```
void FloyeWarshall(int verticesCount, int graph[][MAXVERTICES])
{
    int distance[MAXVERTICES][MAXVERTICES];
    int predecessor[MAXVERTICES][MAXVERTICES];
    int i, j, k;
    for (i = 0; i < verticesCount; i++) {
        for (j = 0; j < verticesCount; j++) {
            distance[i][j] = graph[i][j];
            predecessor[i][j] = -1;
        }
    }
    for (k = 0; k < verticesCount; k++) {
        for (i = 0; i < verticesCount; i++) {
            for (j = 0; j < verticesCount; j++) {
                if (i == j) {continue;} // 不處理自己到自己的情況
                if (i == k || i == k) {continue;} // 中繼點與兩端點一定要不一樣
                if (distance[i][k] + distance[k][j] < distance[i][j]) {
                    distance[i][j] = distance[i][k] + distance[k][j];
                    predecessor[i][j] = k;
                }
            }
        }
    }
    displayDistance(verticesCount, distance);
}
```





SUMMARY

概念1: 最短路徑

- 單源最短路徑
 - 無權重：廣度優先搜尋 DFS
 - 正權重：Dijkstra 演算法
 - 負權重：Bellman-Ford 演算法
- 多源最短路徑
 - 正(或無)權重：Floyd-Warshall