

Introduction to Competitive Algorithms
競賽演算法簡介

許胖

2015 年 10 月 19 日

目 錄

第三章	排序 (Sorting)	5
第一節	比較排序法 (Comparing Sorting Algorithms)	5
一、	泡沫排序法 (Bubble Sort)	6
二、	選擇排序法 (Selection Sort)	7
三、	插入排序法 (Insertion Sort)	7
四、	合併排序法 (Merge Sort)	7
五、	快速排序法 (Quick Sort)	7
第二節	非比較排序法 (Non-comparing Sorting Algorithms)	7
一、	計數排序法 (Counting Sort)	7
二、	基數排序法 (Radix Sort)	7
三、	桶排序法 (Bucket Sort)	7
四、	穩定性 (Stability)	7
五、	綜合比較	7
第四章	線性結構	8
第一節	陣列 (Array)	8
第二節	大數 (Big Number)	8
一、	大數加法與減法	9
二、	大數乘法	9
三、	大數除法	10
四、	更有效率的大數	10
第三節	矩陣 (Matrix)	11
一、	定義	11
二、	矩陣加法與減法	11
三、	矩陣乘法	12

目 錄	2
四、單位元素與反元素	13
五、行列式	13
六、反矩陣	13
七、高斯消去法	14
八、01 矩陣與運算	14
第四節 鏈結串列 (Linked-List)	14
一、鏈結串列實作 (Implementation)	14
二、更多變形	14
第五節 堆疊 (Stack)	14
一、括號匹配	14
二、前序、中序、後序表達式	14
三、排隊視線問題	14
第六節 佇列 (Queue)	14
一、佇列 (Queue)	14
二、優先佇列 (Priority Queue)	14
第九章 圖論	15
第一節 緒論	15
一、頂點和邊	15
二、圖與分支度	17
三、子圖	20
四、路徑與環	22
五、二分圖與完全圖	24
六、連通	25
七、權重	25
第二節 圖的表示法 (Representation)	26
一、相鄰矩陣 (Adjacency Matrix)	26
二、相鄰串列 (Adjacency Lists)	27
三、邊列表 (Edge List)	28
四、前向星法 (Forward Star)	28
五、鏈式前向星	28
六、綜合比較	28
第三節 遍歷與排序	28
一、圖的遍歷 (Traversal of Graph)	28
二、時間戳記法 (Time Stamp)	28

	三、有向無環圖 (Directed Acyclic Graph, DAG)	28
	四、拓樸排序法 (Topological Sort)	28
第四節	最小生成樹 (Minimum Spanning Trees)	28
	一、問題定義	28
	二、Kruskal's 演算法	29
	三、Prim's 演算法	29
	四、次小生成樹	30
第五節	最短路徑問題	30
	一、問題定義	30
	二、Bellman-Ford 演算法	30
	三、SPFA	31
	四、Dijkstra's 演算法	31
	五、Floyd-Warshall 演算法	32
第六節	圖的連通性 (Connectivity)	33
	一、問題定義	33
	二、關節點 (Articulation Point)	33
	三、橋 (Bridge)	33
	四、強連通分量 (Strong Connected Component, SCC)	33
	五、雙連通分量 (Bi-Connected Component, BCC)	34
第七節	匹配問題 (Matching)	34
	一、問題定義	34
	二、交錯軌道 (Alternating Path) 和增廣路徑 (Augmenting Path)	36
	三、最大二分匹配 (Maximum Bipartite Matching)	36
	四、最大權二分匹配——匈牙利演算法 (Hungary Algorithm)	37
第八節	網路流 (Network Flow)	37
	一、問題定義	37
	二、最大流量問題 (Maximum Flow)	37
	三、Ford-Fulkerson 演算法	37
	四、Edmond-Karp 演算法	37
	五、網路流建模	37
	六、最小切割 (Minimum Cut)	37
	七、最大流最小切割 (Max-flow Min-cut)	37
	八、最小成本最大流 (Min-cost Max-flow)	37

目 錄	4
-----	---

第十章 計算幾何	38
第一節 向量	38
一、向量運算	38
二、內積	40
三、外積	40
第二節 計算幾何初步	41
一、基本操作	41

第三章

排序 (Sorting)

這一章中我們會講到排序，在競賽中大多數的排序已被內建函式取代 (例如 C 語言的 `qsort`、C++ 的 `std::sort`)，但是在多數筆試中，題目通常會要求熟悉每個排序的過程。因此本章介紹最經典的幾個排序法，務必在腦中多模擬演練。

我們要探討最經典的**排序問題**：給你 n 個數字，要如何將他從小到大排序好呢？例如，給你 5 個數字 3、7、1、2、5，這個問題最後要得到的答案是 1、2、3、5、7，那麼如何將前面的數字排序好就是本章的重點所在。

本章附教學影片，教學影片可以在稍微了解各個排序法之後再觀看，更能熟悉理解每個排序，以免看不懂。

第一節 比較排序法 (Comparing Sorting Algorithms)

排序中，最基礎的即是比較排序法。比較排序法，意思就是裡面的元素兩兩比較，基本上來說，如果 n 個元素兩兩比較，總共要比較 $\frac{n \times (n-1)}{2}$ 次，時間複雜度會達到 $O(n^2)$ ，但事實上，有些比較是多餘的，例如：我們已知 $a < b$ ， $b < c$ ，這時我們再比較 a 和 c 就是多餘的 (很明顯，因為我們可以從上述兩個關係式找出 $a < c$)，因此發展出特殊的排序法來減少無謂的比較，時間複雜度可達到 $O(n \lg n)$ 。

接下來介紹三個基礎的排序法：泡沫排序、選擇排序、插入排序。

一、 泡沫排序法 (Bubble Sort)

教學影片：[泡沫排序法](#)

泡沫排序法正如其名，排序時最大的數字會「浮出來」，大略的過程如下：首先我們先讓最大的數字浮上來 (到最右邊)，接著在讓第二大的數字浮上來，以此類推。那要如何讓他「浮上來」呢？答案是「交換」，別忘了我們是比較排序法，因此我們要將數字兩兩比較，如果發現左邊的數字比右邊來得大，那麼就把較大的數字交換到右邊去。

詳細過程如圖 3.1 所示，第一次我們比較第 0 個和第 1 個數字 (第一行)—— 3 和 7，發現右邊的 7 比較大，所以不需要做交換的動作；接著比較第 1 個和第 2 個數字 (第二行)—— 7 和 1，發現左邊的 7 比右邊的 1 大，因此 7 和 1 交換；第 2 個和第 3 個數字 (第三行)—— 7 和 2，第 2 個數字是 7 是剛剛我們對 7 和 1 比較後交換而來，因此這次 7 大於 2，就將 7 交換到右側；第 3 個和第 4 個數字 (第四行)—— 7 和 5 交換；最後 (第五行)，我們發現最右邊的 7 就是最大的數字，剩下前 4 個數字還未進行排序的動作。

二、 選擇排序法 (Selection Sort)

三、 插入排序法 (Insertion Sort)

四、 合併排序法 (Merge Sort)

五、 快速排序法 (Quick Sort)

第二節 非比較排序法 (Non-comparing Sorting Algorithms)

一、 計數排序法 (Counting Sort)

二、 基數排序法 (Radix Sort)

三、 桶排序法 (Bucket Sort)

四、 穩定性 (Stability)

五、 綜合比較

第四章

線性結構

第一節 陣列 (Array)

大家對陣列都不陌生，抽象意義上，陣列就是一群**有關聯**的元素排成的**序列 (Sequence)**，這個序列可以是一維的，也可以是二維以上，二維以上的陣列看來不像是序列，反而像是矩陣 (二維時) 或是一大堆數字整齊地堆在一起 (二維以上)，因此陣列又有另外的稱呼叫做**數組 (Array)**。

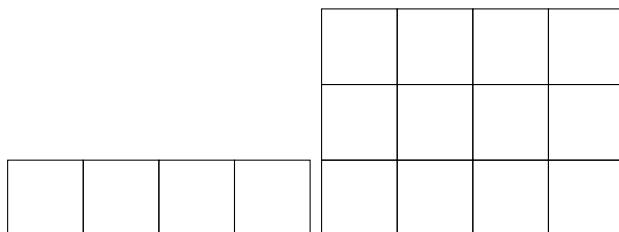


圖 4.1: 一維陣列(左)和二維陣列(右)，或稱數組、矩陣

因為陣列是「**關係上連續**」的元素所形成的序列，所以實際上，我們有兩種作法：

1. 第一種是讓他真的在實際的記憶體上連續，這就是我們常用的陣列；
2. 第二種，我們不管他是不是在記憶體上連續，模擬連續的情形，這種又有一個名稱叫做**鏈結串列 (Linked list)**，如下圖 4.2：

第二節 大數 (Big Number)

有些時候，我們可能會存不下數字，這些存不下的數字我們稱為**大數 (Big num-**

ber)。例如整數 (integer) 最多只能存到 $2^{32} - 1$ (無號)，長整數 (long integer) 最多只能存到 $2^{64} - 1$ (無號)，不管是任何的程式語言，都有他的限制，那麼們要怎樣儲存一個連最強大的資料型態都不能儲存的數呢？

我們用陣列來模擬。如果現在給你一個數字 1234567890，我們用陣列將數字一位一位儲存，如下圖 4.3。

一、大數加法與減法



我們知道怎麼儲存後，接著我們要知道怎麼運算，很簡單，就像我們小學的加減法一般，將個位數對齊、十位數對齊、百位數對齊、... 以此類推，全部加起來就可以囉。當然，回憶一下我們小學的時候，算加減法一定會注意進位和借位 (carry)，因此我們的程式也要自己去模擬進位和借位，不同的是，我們做加減法時可以先不用處理進位，等到後來再一口氣進位。時間複雜度為 $O(n)$ 。

以下為大數加法的虛擬碼：

練習題

- ✓ UVa 424 / ZJ c034: *Integer Inquiry*
求很多大數之和。

更多練習題


-  UVa 495 / ZJ c121: *Fibonacci Freeze*
-  UVa 10579: *Fibonacci Numbers*


二、大數乘法

練習題

- ✓ UVa 10106: *Product*
給你 X, Y ($0 \leq X, Y \leq 10^{250}$)，輸出 $X \times Y$ 。
- ✓ UVa 10220 / ZJ c119: *I Love Big Numbers*
求 $n!$ 每個位數之和。

更多練習題

 UVa 623 / ZJ c120: 500!

 UVa 324: Factorial Frequencies

三、大數除法

練習題

✓ ZJ a021: 大數運算
計算 a, b 的加減乘除。

✓ ZJ b115: TOI2008 2. 大數運算

四、更有效率的大數

練習題

✓ UVa 619 / ZJ c132: Numerically Speaking

26 進制數字轉 10 進制數字。

✓ UVa 288: Arithmetic Operations With Large Integers

給定一運算式，包含加法、減法、乘法、指數次方，無括號，但是運算先後順序依然存在，請問運算結果？

✓ UVa 10023: Square root

給你一個數字 Y ($1 \leq Y \leq 10^{1000}$)，問 Y 的開根號為多少？

✓ UVa 748: Exponentiation


給一浮點數 R ($0.0 < R < 99.999$) 和整數 n ($0 < n \leq 25$)，求 R^n 。

更多練習題

 ZJ d283: 大數加法


 UVa 10013 / ZJ d056: Super long sums

 ZJ d202: $a+b??$

 UVa 485: Pascal's Triangle of Death

 UVa 10083: Division

 UVa 465 / ZJ d873: Overflow

 ZJ d218: 加法與次方的奧妙

🔗 UVa 10523: Very Easy !!!

第三節 矩陣 (Matrix)

一、定義

矩陣就是二維陣列，若說一維陣列是一個由 n 個元素排成一列的結構，那麼矩陣可以看做是有 m 個一維陣列並排，每個一維陣列有 n 個元素排成一列。

通常我們用英文字母大寫來稱呼矩陣，如下圖是矩陣 A ，裡面所有元素形成一個二維陣列，其中下圖的 $a[1, 1]$ 、 $a[1, 2]$ 、 $a[1, 3]$ 、 \dots 、 $a[1, n]$ ， n 個元素形成一個列 (row)， $a[1, 1]$ 、 $a[1, 2]$ 、 \dots 等等叫做第一列， $a[2, 1]$ 、 $a[2, 2]$ 、 \dots 等等叫做第二列，以此類推；同樣地， $a[1, 1]$ 、 $a[2, 1]$ 、 \dots 、 $a[m, 1]$ ， m 個元素形成一個行 (column)， $a[1, 1]$ 、 $a[2, 1]$ 、 \dots 、 $a[m, 1]$ 就是第一行。二維陣列有兩個註標，通常來說，第一個註標表示列，第二個註標表示行，例如第 i 列第 j 行的元素就是 $a[i, j]$ 。

如果要描述一個矩陣的大小，我們就看他有幾列幾行，一個有 m 列 n 行的矩陣被稱為一個 $m \times n$ 矩陣，記為 $A_{m \times n}$ 。如果 $m = n$ 的話，我們稱為方陣，或是 n 階矩陣，例如一個 3×3 的矩陣我們可以稱他為 3 階矩陣。

矩陣其他的性質眾多，以下只介紹高中才會接觸到的範圍，其他的東西，有興趣的讀者可以參考大學的一門課程「線性代數」。

練習題

✓ ZJ a015: 矩陣的翻轉

將元素 $a[i, j]$ 翻轉至 $a[j, i]$ 。

二、矩陣加法與減法

矩陣加法的規則很簡單，假設現在有兩個矩陣 $A_{m \times n}$ 和 $B_{m \times n}$ ，兩個矩陣的大小必須一樣，則

$$A + B = a[i, j] + b[i, j]$$

，意即 A 與 B 每個對應元素相加；矩陣減法和加法類似， $A - B = a[i, j] - b[i, j]$ ，代表 A 和 B 矩陣相減。舉例，甲國與乙國生產汽車和稻米，將去年的產量和今年的產量做成下面兩表：

假設我們要計算兩年間，兩國所有汽車和稻米的總產量，那麼就會應用到矩陣加法：

我們可以很清楚地從圖 4.13 看到矩陣加法的運算方式，如果矩陣的大小不同，也就直接代表兩個矩陣的資料沒有對應，因此不同大小的矩陣無法相加似乎挺自然的(?)，如果兩個矩陣大小一樣，但是裡面的資料屬性不同呢？以圖 4.14 來說：

丙國不生產汽車，取而代之的是生產麵粉和稻米，那麼這個矩陣可以和前面兩個甲國和乙國的矩陣相加嘛？

就矩陣加法而言，它是可以相加的 (因為矩陣大小相同)，矩陣加法只是提供一個抽象的運算概念，並無法表示兩個矩陣相加的實際意涵，就如同我們知道 $2 + 3 = 5$ ，但要是 2 顆蘋果加上 3 台汽車呢？讀者可以思考一下。

三、矩陣乘法

接著我們看看矩陣乘法，不少讀者可能會認為矩陣乘法的運算複雜，但只要抓住它核心的思想，那麼矩陣乘法應是很好理解的。

假設現在有兩個矩陣 $A_{m \times r}$ 和 $B_{r \times n}$ (注意矩陣大小)，做矩陣乘法後的矩陣為 $C_{m \times n}$ (注意矩陣大小)，運算規則為：

$$A \times B = C = c[i, j] = \sum_{k=1}^r a[i, k] \times b[k, j]$$

看起來非常複雜？我們一步一步來分析矩陣乘法。首先，我們再三要注意的是，矩陣乘法的 A 、 B 唯一的限制—— A 的行數要和 B 的列數相同，圖 4.15 表示了這個條件。

從圖 4.15 可以看到，矩陣乘法的方向，當我們的矩陣相乘時，前面的矩陣橫著走，後面的矩陣順著往下走，假設是 A 矩陣的第 i 列、 B 矩陣的第 j 行，運算公式告訴

我們這兩個相乘後會產生

$$c[i, j] = a[i, 1] \times b[1, j] + a[i, 2] \times b[2, j] + \cdots + a[i, k] \times b[k, j] + \cdots + a[i, r] \times b[r, j]$$

，每個 $a[i, k]$ 一定會和 $b[k, j]$ 相乘，因此 A 的行數和 B 的列數一定要一樣才能做乘法，否則會沒有對應。

如果照著規則走的話，那麼最後產生的矩陣就會是一個 $m \times n$ 的矩陣：因為 A 是 $m \times r$ 的矩陣，而 B 是 $r \times n$ 的矩陣，其中我們在做乘法的過程中，我們會把 A 矩陣第 i 列和 B 矩陣第 j 行的 r 個數相乘並相加，最後變成 C 矩陣的一個數—— $c[i, j]$ 。

因為 A 矩陣有 m 列， C 矩陣也會有 m 列；同理，因為 B 矩陣有 n 行，所以 C 矩陣也依然會有 n 行，總而言之， C 矩陣就是個 $m \times n$ 的矩陣。

練習題

✓ ZJ d481: 矩陣乘法

給兩個矩陣 A 、 B ，若能相乘，請輸出 $AimesB$ 的結果。

四、單位元素與反元素

練習題

✓ UVa 11149 / ZJ d766: undefined

給一個 $nimesn$ 矩陣 A ，計算 $A + A^2 + A^3 + \cdots + A^k$ 。

五、行列式

六、反矩陣

練習題

✓ ZJ d623: 反方陣

求二階反矩陣。

七、 高斯消去法

練習題

✓ ZJ *b016*: D. Mitlab

矩陣加減乘除運算。

八、 01 矩陣與運算

練習題

✓ ZJ *b002*: 關燈

求最少按多少次開關，燈會全暗。

✓ ZJ *a243*: 第四題：點燈遊戲

求是否能夠經由有限次開關操作，燈會全暗。

第四節 鏈結串列 (Linked-List)

一、 鏈結串列實作 (Implementation)

二、 更多變形

第五節 堆疊 (Stack)

一、 括號匹配

二、 前序、中序、後序表達式

三、 排隊視線問題

第六節 佇列 (Queue)

一、 佇列 (Queue)

二、 優先佇列 (Priority Queue)

第九章

圖論

第一節 緒論

這裡要先介紹一些圖論的基本觀念，對初學者會稍微複雜、冗長，因為圖論要討論的東西太多了，在這一章節中筆者已儘量挑出最重要的觀念來解說，讀者也可以配合後面的章節交互翻閱，或許能增加理解度。

一、頂點和邊

圖論研究的重點就是「圖」，它是一種很特別的數學模型，而非平常我們所見到的「圖形」。一張圖會有數個**頂點 (Vertex)**，頂點跟頂點之間可能會有**邊 (Edge)** 連通，只要符合這個條件，我們就稱爲一張圖，記爲 G 。若用數學來描述的話，圖 G 有兩個集合：一個集合 V 包含頂點、另一個集合 E 包含圖 G 的所有邊，則圖 G 記爲 $G = (V, E)$ 。

對於一張圖 G ，我們用 $V(G)$ 來表示 G 的頂點集合，而用 $E(G)$ 表示 G 的邊集合。我們用 $|E(G)|$ 或是 $|G|$ 表示邊數、或者稱爲圖 G 的大小 (Size)；用 $|V(G)|$ 或者 $\|G\|$ 表示頂點數、又稱爲圖 G 的階數 (Order)，在本章的各節中，爲了方便表示，我們一律假設圖 G 的頂點數爲 n ，邊數爲 m 。

圖 9.1 中有兩張圖，注意到在圖論中，我們只討論頂點和頂點間的連接關係，不討論邊的長度、形狀，因此和圖 9.1 中兩張圖是相同的。

我們將頂點編號，用 (u, v) 來表示一條連結 u 和 v 的**無向邊 (Edge)**，無向邊可以

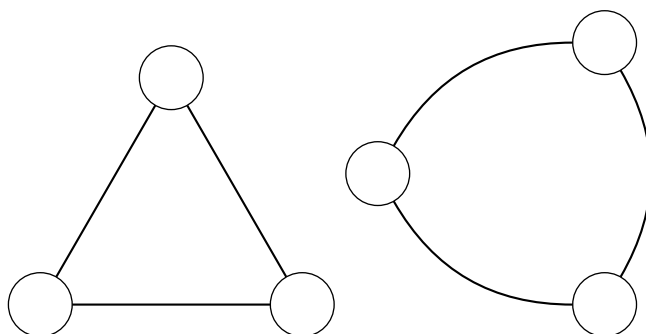


圖 9.1: 圖不考慮大小、長度、形狀

想像成：如果你在頂點 u 的話，妳可以藉由這條邊走到頂點 v ，反之也可以從頂點 v 走到頂點 u ，因此 $(u, v) = (v, u)$ 。如果是一條有向邊 (Arc) 的話，我們記為 $\langle u, v \rangle$ ，代表這條邊只能從 u 走到 v ，但不能從 v 走回 u ，如圖 9.2。

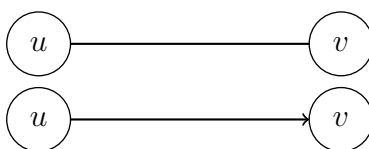


圖 9.2: 無向邊和有向邊

一條無向邊 (u, v) 或是有向邊 $\langle u, v \rangle$ ， u 、 v 兩點都是這條邊的端點 (Endpoint, Endvertex)。如果有一條邊 e 連接 u 、 v 兩個頂點，我們稱 u 和 v 相鄰 (Adjacent)，或是互相稱為鄰點 (Adjacent Node)；而 e 和頂點 u 、 v 相接 (Incidence)。若兩條邊 e_1 、 e_2 至少有一個相同的端點，我們也稱 e_1 、 e_2 相鄰，或是互相稱為鄰邊 (Adjacent Edge)。

我們可以把圖 9.1 的頂點編號 (Vertex-labeled) 為 $V = \{v_1, v_2, v_3\}$ ，我們會得到三條邊： $E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$ ，因此圖 9.1 用數學來表示即為

$$\begin{aligned} G &= (V, E) \\ &= (\{v_1, v_2, v_3\}, \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}) \end{aligned}$$

結果如圖 9.3。

有時候，我們會對邊進行編號 (Edge-labeled)，圖 9.3 中，我們依序將邊編號為 $e_1 = (v_1, v_2)$ 、 $e_2 = (v_2, v_3)$ 、 $e_3 = (v_3, v_1)$ 。

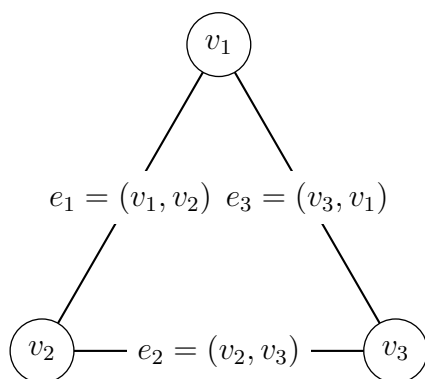


圖 9.3: 編號

有了編號的觀念後，我們再看有向邊的性質。對於有向邊 $\langle u, v \rangle$ ，頂點 u 稱為頂點 v 的前驅 (Successor)， v 稱為 u 的後繼 (Predecessor)，如圖 9.4，頂點 v_1 的前驅有 v_4 、 v_6 、 v_7 、 v_8 這 4 個頂點，而後繼有 v_2 、 v_3 、 v_5 這 3 個節點。

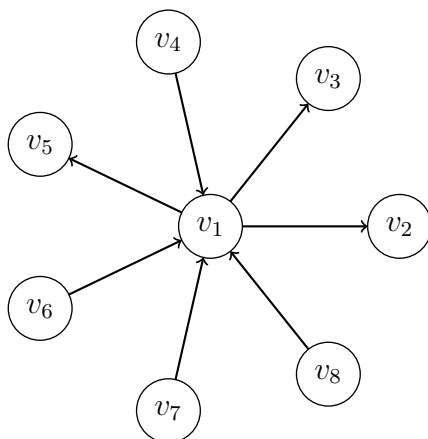


圖 9.4: 前驅和後繼

對於頂點 u ，所有連向後繼 v 的邊，我們稱為前向星 (Forward Star)；若是對於頂點 v ，所有連向前驅 u 的邊，我們稱為後向星 (Backward Star)，所有的前向星和後向星合稱為星 (Star)。回到圖 9.4，我們可以找出頂點 v_1 的前向星為 $\{\langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle, \langle v_1, v_5 \rangle\}$ 、後向星為 $\{\langle v_4, v_1 \rangle, \langle v_6, v_1 \rangle, \langle v_7, v_1 \rangle, \langle v_8, v_1 \rangle\}$ ，而這些邊形成的集合就是頂點 v_1 的星。

二、圖與分支度

如果一張圖 G 皆由無向邊構成，我們稱為無向圖 (Undirected Graph)，例如，圖

9.1 就是一張無向圖；如果都是由有向邊所構成，則稱為有向圖 (Directed Graph, Digraph)，圖 9.5 表示一有向圖；若不全為無向邊和有向邊，則稱為混合圖 (Mixed Graph)。

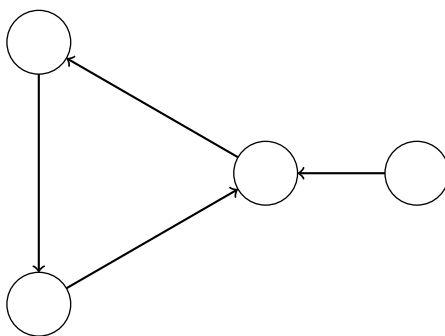


圖 9.5: 有向圖

我們會計算圖 G 中，一個頂點 u 的分支度 (Degree)，代表這個頂點和邊的相接數量，記為 $d_G(u)$ ，例如圖 9.6 中，節點 v_1 的分支度為 3，因為它有相接的 3 條邊： (v_1, v_2) 、 (v_1, v_3) 、 (v_1, v_5) ；節點 v_4 的分支度為 $d_G(v_4) = 2$ ，相接的邊為 (v_2, v_4) 、 (v_4, v_5) ，以此類推。

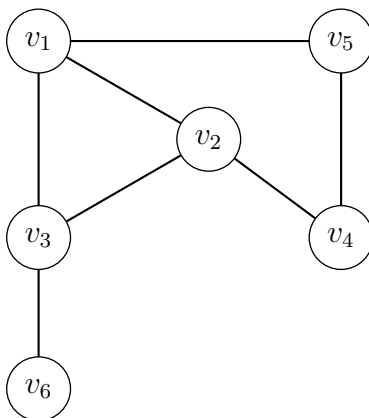


圖 9.6: 計算分支度

我們也會計算整張圖 G 的總分支度 (Total Degree) $d(G)$ ，它代表每個點分支度的總和，我們依然以圖 9.6 為例，我們依序求出各節點的分支度： $d_G(v_1) = 3$ 、 $d_G(v_2) = 3$ 、 $d_G(v_3) = 3$ 、 $d_G(v_4) = 2$ 、 $d_G(v_5) = 2$ 、 $d_G(v_6) = 1$ ，我們可以求出總分支度：

$$d(G) = d_G(v_1) + d_G(v_2) + d_G(v_3) + d_G(v_4) + d_G(v_5) = 14$$

我們可以從總分支度發現一個性質：那就是不管圖 G 長什麼樣子， $d(G)$ 必為偶數。為什麼呢？我們想想，對於節點 u ，當你分支度增加時，代表有新的邊和節點 u 相接，那麼分支度實際上會和邊有關，我們換個角度來觀察，一條邊有兩個端點，每個端點都會貢獻「1 個」分支度，因此不管圖 G 為何，總分支度恰好就是邊數的 2 倍。

$$d(G) = \sum_{u \in V(G)} d_G(u) = 2 \times |E(G)| \quad (1.1)$$

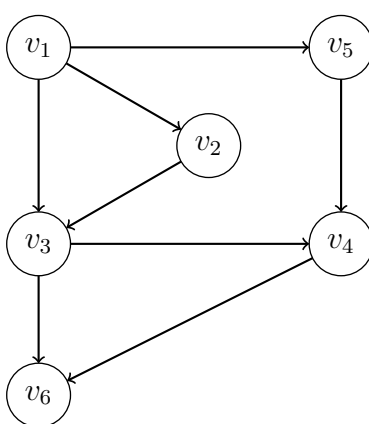


圖 9.7: 入分支度和出分支度

如果是有向圖 G ，我們會計算出分支度 (Outdegree) 和入分支度 (Indegree)，出分支度記為 $d_G^+(u)$ ，代表所有和節點 u 相接的有向邊中，從頂點 u 出去邊的個數，也就是所有 $\langle u, v \rangle$ 的個數；入分支度記為 $d_G^-(u)$ ，代表所有和節點 u 相接的有向邊中，所有 $\langle v, u \rangle$ 的個數。如果一個節點 u 的入分支度為 0，我們稱為源點 (Source)；如果出分支度為 0，我們稱為匯點 (Sink)。

圖 9.7 中，節點 v_3 的分支度為 4，因為它是有向圖，所以會特別討論它的出分支度 $d_G^+(v_3) = 2$ ，因為它有兩條向外的有向邊 $\langle v_3, v_4 \rangle$ 和 $\langle v_3, v_6 \rangle$ ；入分支度為 $d_G^-(v_3) = 2$ ，因為有兩條向內的有向邊 $\langle v_1, v_3 \rangle$ 和 $\langle v_2, v_3 \rangle$ 。此外，因為 $d_G^-(v_1) = 0$ ，節點 v_1 是源點；而 $d_G^+(v_6) = 0$ ，因此節點 v_6 是匯點。

有向圖的分支度和先前普通的分支度也有相似的性質，我們也會定義總入分支度 $d^-(G) = \sum_{u \in V} d_G^-(u)$ ，總出分支度為 $d^+(G) = \sum_{u \in V} d_G^+(u)$ ，我們看有向邊 $\langle u, v \rangle$ 的性質：它端點 u 是出去的方向，因此 u 的出分支度會加 1；而節點 v 是進去的方向， v

的入分支度也會加 1。總和而言，總出分支度會等於總入分支度，同時，這兩個度數也會等於有向邊的邊數。

$$d^+(G) = d^-(G) = |E(G)| \quad (1.2)$$

$$d(G) = d^+(G) + d^-(G) \quad (1.3)$$

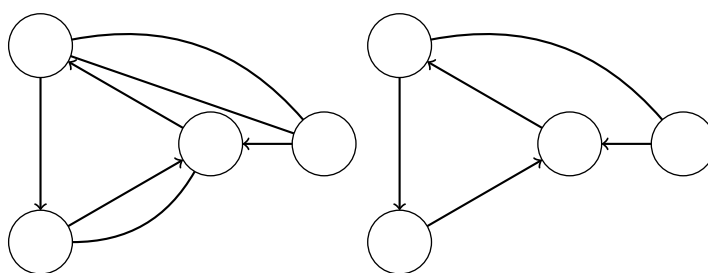


圖 9.8: 複圖和簡單圖

如果一張圖 G ，其中兩個頂點間有一條以上的邊（稱為重邊）或者存在自環（形如 (u, u) 或者 $\langle u, u \rangle$ 的邊），稱為複圖 (Multigraph)，反之如果任兩個頂點有不超過一條的邊，就稱為簡單圖 (Simple Graph)，一般來說我們沒有特別稱呼「圖」的話，就是指簡單無向圖。圖 9.8 表示一個複圖和一個簡單圖。

三、子圖

在任意一張圖 $G = (V_G, E_G)$ 中，有另一張圖 $H = (V_H, E_H)$ ，其中 V_H 是 V_G 的子集合， E_H 是 E_G 的子集合，則稱 H 是 G 的子圖 (Subgraph)。如果 $V_H = V_G$ 的話，則稱 H 為生成子圖 (Spanning Subgraph)。下列數張圖 G 中， $V_G = \{v_1, v_2, \dots, v_9\}$ 、 $E_G = \{e_1, e_2, \dots, e_{11}\}$ ，圖 9.9 是其中一個子圖，而圖 9.11 是生成子圖。

上圖 9.9 的子圖 H 為 $V_H = \{v_1, v_2, v_3, v_4, v_6, v_7, v_8\}$ 、 $E_H = \{e_1, e_2, e_6\}$ 。而下圖 9.10 的生成子圖中 $V_H = V_G$ 、 $E_H = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ 。

在圖 $G = (V_G, E_G)$ 中，如果子圖 H 的頂點集合為 V_H 是 V_G 的子集合，對於 E_G 中所有的邊 (u, v) ，當頂點 u, v 都屬於 V_H ，而邊 (u, v) 屬於 E_H 的話，則稱 H 為導出子圖 (Induced Subgraph)，記為 $G[V_H]$ ，代表圖 H 有一些頂點，而 H 的邊取決於

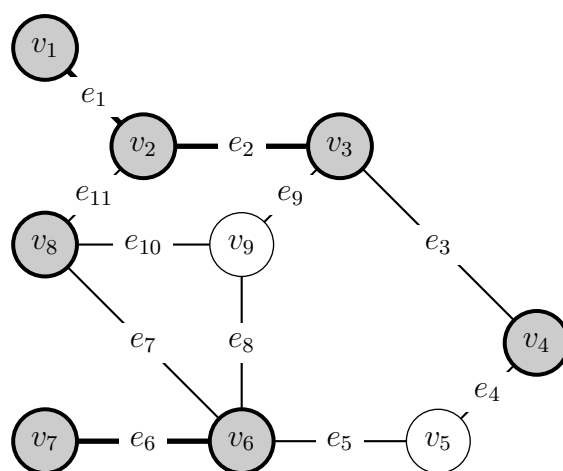


圖 9.9: 子圖

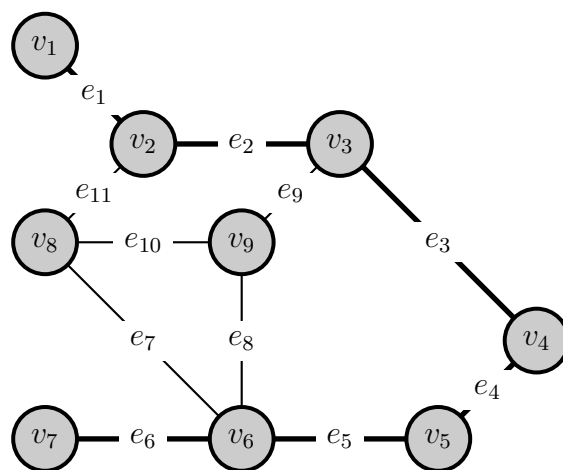


圖 9.10: 生成子圖

圖 G 中頂點在 H 內的所有邊。同樣地，如果子圖的邊集合為 E_H 是 E_G 的子集合，對於所有 V_H 中的頂點 u ，如果 u 在 E_H 其中一條邊的端點上，則 u 在 V_H 中，我們稱 H 為邊導出子圖 (Edge-induced Subgraph)，記為 $G[E_H]$ 。

例如，圖 9.11 是一個導出子圖 H ，它的頂點集合 $V_H = \{v_1, v_2, v_3, v_8, v_9\}$ ，所以邊集合為 $E_H = \{e_1, e_2, e_9, e_{10}, e_{11}\}$ 。反之，像圖 9.9 就不是一個導出子圖，因為頂點 v_2 和 v_8 都屬於 V_H ，但 e_{11} 卻不屬於 E_H ； $e_4 = (v_4, v_5)$ 屬於 E_H ，但頂點 v_4 和 v_5 不屬於 V_H 。

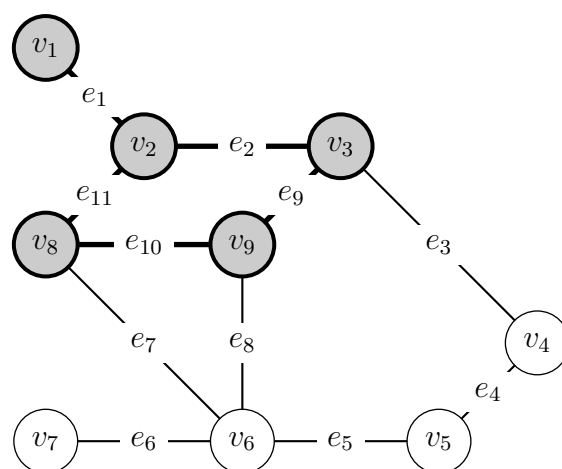


圖 9.11: 導出子圖

四、路徑與環

W 是在圖 $G = (V_G, E_G)$ 中，一個起始於頂點、終於頂點，且頂點和邊交錯的序列 $W = \{v_1, e_1, v_2, e_2, \dots, v_i, e_i, v_{i+1}, \dots, e_{n-1}, v_n\}$ ，其中 v_1, v_2, \dots, v_n 屬於 V_G ， e_1, e_2, \dots, e_{n-1} 屬於 E_G ，且每條邊 e_i 都是 (v_i, v_{i+1}) 或 $\langle v_i, v_{i+1} \rangle$ 。則我們把 W 稱為**道路 (Walk)**，其中 v_1 是**起點**， v_n 是**終點**。如果 v_1 和 v_n 是相同頂點，則稱為**封閉道路 (Close Walk)**；反之，如果 v_1 和 v_n 不同，則稱為**開放道路 (Open Walk)**。在簡單圖中，兩個相同的頂點指的是相同邊，因此我們可以只用頂點來描述一條道路 $W = \{v_1, v_2, \dots, v_i, v_{i+1}, \dots, v_n\}$ 。

道路的長度 L 代表道路中邊的數量。如果是封閉道路，長度 $L = |E(W)| = |V(W)|$ ；如果是開放道路的話，長度則為 $L = |E(W)| = |V(W)| - 1$ 。

如果一條開放道路中，所有的邊都不重複，我們稱為**行跡 (Trail)**，例如圖 9.12 就是一條行跡 $\{v_7, e_6, v_6, e_8, v_9, e_9, v_3, e_3, v_4, e_4, v_5, e_5, v_6, e_7, v_8\}$ 。

如果一條開放道路中，所有的頂點都不會重複的話，我們就稱為**路徑 (Path)**，又稱為**簡單路徑 (Simple Path)**，後者源於古時候，路徑這個名詞被當做任意一條開放道路，在現在的意義上，路徑即代表簡單路徑，為何被稱為簡單路徑呢？那是因為，如果頂點不重複，那麼邊也不會重複，讀者可以好好想想為什麼，依然跟邊的性質有關。一條頂點數為 n 的路徑我們記為 P_n ，下圖 9.13 代表一條路徑 $P_7 = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ 。

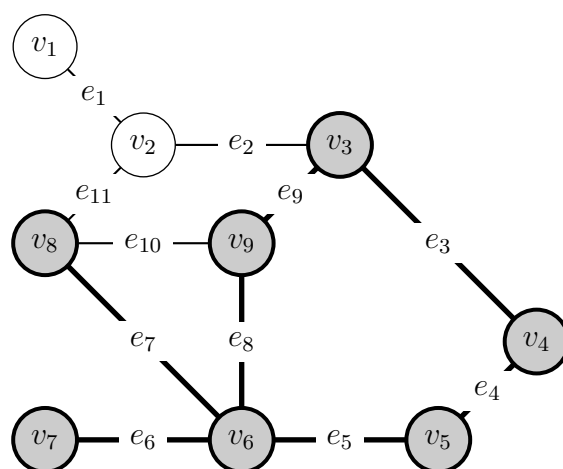


圖 9.12: 行跡

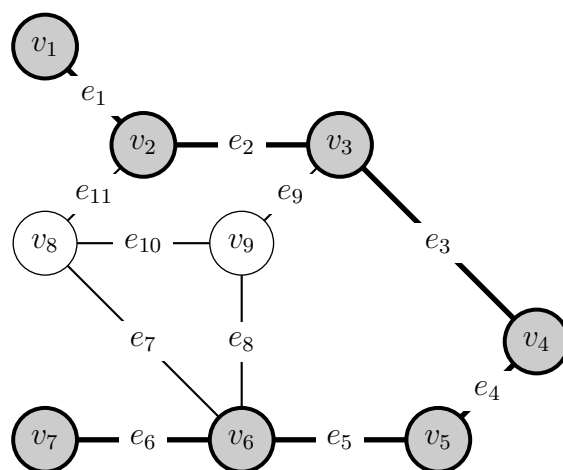


圖 9.13: 路徑

接著，如果一條封閉道路的邊不重複，我們稱為**迴路** (Circuit, Tour)。如果封閉道路中，所有的頂點不重複，則稱為**環** (Cycle) 或是簡單環 (Simple Cycle)，後者的理由和簡單路徑一樣，在現在已不常用這名詞。一個長度為 n 的環記為 C_n (通常 $n > 2$)，下圖 9.15 就是一個 C_4 環 $C_4 = \{v_2, v_3, v_8, v_9\}$ 。

一個環的頂點數量若是奇數的話，稱為奇環 (Odd Cycle)；若為偶數，則稱為偶環 (Even Cycle)。如果一張圖是**無環圖** (Acyclic Graph) 的話，那麼圖上不會有環存在。

五、二分圖與完全圖

圖 G 的頂點數 n ，其中任兩個頂點間恰有一條邊，我們稱為完全圖 (Complete Graph)，記做 K_n 。完全圖有 $n \times (n-1)/2$ 條邊，而且完全圖、路徑和環有以下性質：

$$P_1 = K_1 \quad (1.4)$$

$$P_2 = K_2 \quad (1.5)$$

$$C_3 = K_3 \quad (1.6)$$

下圖 9.16 是一個完全圖 K_5 ，我們可以看見任兩個頂點間都有一條邊。

如果一張圖 $G = (V, E)$ ，它的頂點可以拆成兩個集合 V_A 和 V_B ， V_A 和 V_B 間沒有交集，且 V_A 中的任意兩個頂點都不相鄰， V_B 也是，換句話說就是 E 中所有邊的兩個端點，其中一個在 V_A 、另外一個在 V_B ，這樣形成的圖 $G_B = (V_A, V_B; E)$ 我們稱二分圖 (Bipartite Graph)，如圖 9.17 就是一張二分圖。

我們可以看到二分圖有個很明顯的特性：不管是哪個點集合裡面所有點都不相鄰，如果我們現在隨意給一張圖，那麼如何判定是不是一個二分圖呢？我們可以考慮一個問題：現再有一張圖 G ，我們要對它的頂點塗上紅、藍兩種顏色，條件是——相同顏色的頂點不相鄰，問可不可以有這種著色呢？

我們可以很清楚的看到這個問題的限制條件和二分圖的性質是相同的，也就是說如果該圖是一張二分圖的話，那麼它就可以被塗成紅、藍兩種顏色，此時這張圖被稱為 2-著色圖 (2-colorable Graph)，二分圖同時也是一個 2-著色圖。

我們可以看到圖 9.18 中， V_A 被塗成藍色、 V_B 為紅色，而且同色並沒有相鄰 (別忘了兩個頂點相鄰要有邊相接)，因此我們要知道是否為二分圖，就看是否為 2-著色圖。如果一張圖不是 2-著色圖，那麼它就不能分成藍色和紅色兩群頂點，因此就不會是二分圖，所以我們要做的工作變成判斷一張圖是否為 2-著色圖，那問題點在於——什麼時候不是一張 2-著色圖呢？

答案很簡單，如果有一個點 v ，它相鄰的點中有些是紅色、有些是藍色，那麼頂點 v 該塗紅色還是藍色呢？因此，我們可以知道這時候 2-著色圖沒辦法成立，那麼就可以推得該圖不是一個二分圖。唯一會造成頂點 v 不知道要塗什麼顏色的情形，只有在奇環存在時才會如此，剩下的細節讀者可自行思考，圖 9.19 描述了上面的概念。

如果二分圖的兩個點集間都有一條邊，我們稱為**完全二分圖 (Complete Bipartite Graph)**，如果集合 V_A 的點數為 a 、 V_B 的點數為 b ，完全二分圖記為 $K_{a,b}$ ，下圖 9.20 是一個 $K_{3,4}$ 。

六、連通

有時候，我們會想要了解一張圖 G 是否**連通 (Connected)**，一張圖是**連通圖 (Connected Graph)** 代表圖上任意兩個頂點 u 、 v 都存在至少一條路徑相連。

但常常一張圖並不會都是連通的，因此我們會探討它的子圖是否連通，例如上圖 9.21 中的圖 $G = (V_G, E_G)$ 中，子圖 $H = (V_H, E_H)$ ， $V_H = \{v_2, v_3, v_5\}$ 、 $E_H = \{e_2, e_3\}$ ，因為 H 的任意兩點間都至少有一條路徑，因此稱 H 為**連通子圖 (Connected Subgraph)**。

但僅僅知道這樣子是不夠的，我們想要知道一張圖中所有極大的連通子圖有哪些，所謂的極大即是：假設圖 G 中的一個頂點 v ，那麼包含頂點 v 的最大連通子圖就是其中一個極大的連通子圖，此時稱為**連通元件 (Connected Component)**，圖 9.21 中，子圖 $V_H = \{v_7, v_8, v_9\}$ 、 $E_H = \{e_7, e_8, e_9\}$ 就是其中一個連通元件，而整張圖 G 共有 4 個連通元件，這四個連通元件的頂點分別是： $\{v_1, v_2, v_3, v_4, v_5\}$ 、 $\{v_6\}$ 、 $\{v_7, v_8, v_9\}$ 和 $\{v_{10}, v_{11}, v_{12}\}$ 。

我們第八章有提過樹的定義：若一張圖 G 是無環連通圖，那麼就是一棵樹，此時葉子 (Leaf) 的分支度為 1。

七、權重

一張圖有時會帶**權重 (Weight, Cost)**：權重是一個實數，通常是一個非負整數或是一個有理數，若權重在邊上，就代表邊的一種性質，讀者可以想像成是這條邊的長度；或者這條邊是一條水管，而上面的權重代表流量限制，……等等。有時候權重也會在頂點上，它可以代表這個頂點的大小、顏色、重量、……之類，這些條件端看題目怎麼設計。不管是邊有權重還是點有權重，我們都稱為**帶權圖 (Weighted Graph)**，假設有一條邊 $e = (u, v)$ ，其權重我們會記為 $\omega(e)$ 或是 $\omega(u, v)$ 。圖 9.22 是典型的帶權圖：

第二節 圖的表示法 (Representation)

如同樹一樣，我們也需要適當的資料結構來儲存圖的資訊，一張圖的頂點我們通常將其編號為 $0 \sim n-1$ 或 $1 \sim n$ 兩種，因此在記錄頂點的資訊上只要開一個陣列就可以處理，因此我們主要的重點是如何儲存邊，使用不同儲存邊的方法會有不同的優缺點和用途。

下面我們會以圖 9.8 (有向圖)、圖 9.21 (無向圖)、圖 9.22 (帶權圖) 為例，簡介各種儲存方式的差異和技巧，特定的演算法需要不同的資料結構，所以可以配合之後的章節來看。

一、相鄰矩陣 (Adjacency Matrix)

相鄰矩陣是以頂點為主的儲存方法，主要是一個二維陣列 A ，陣列的兩個維度是頂點 u 和 v ， $A[u, v]$ 代表邊 (u, v) 的情形。如果是簡單無向圖的話， $A[u, v]$ 記錄頂點 u 和頂點 v 之間是否有邊，如果是帶權圖的話，則會記錄 $A[u, v]$ 的權重。

相鄰矩陣的優點是實作上很方便，儲存特性較為直觀，而且可直接查詢 $A[u, v]$ 是否有邊。缺點是當我們在做圖遍歷時效率較低，並且如果圖的邊數較少，那麼矩陣中大量的儲存格都沒用到，因此會造成空間上的浪費，最後，如果題目有重邊的情形，相鄰矩陣是無法表現出重邊，因此相鄰矩陣適用在邊數很大 (尤其是完全圖) 的時候。

範例 9.22 我們可以看到圖 9.21 用相鄰矩陣的情形，為了某些便利性以及直觀等因素，當 (u, v) 有邊時，我們可以儲存為 1，並且無向圖有一個特點：對於 (u, v) 這條邊，因為我們從頂點 u 到頂點 v 可以經由這條邊連通，因此 $A[u, v] = 1$ ，反之我們也可以從頂點 v 藉由 (u, v) 走回頂點 u ，因此 $A[v, u] = 1$ 。所以我們可以得到結論：

$$\text{對於無向邊 } (u, v), A[u, v] = A[v, u] = 1$$

此外，我們也可以從範例 9.23 看出，這個矩陣的空白處甚多，空白處也就是我們沒有用到的格子，此例因為邊數少而造成空間上不必要的浪費，這類邊數少的圖我們又稱稀疏圖 (Sparse Graph)。空白處的部分，一般來說我們習慣填 0，但由於题目的差異，我們可以填入不同的數字，來區別空白格子和有邊的格子，常見的數字有以下三類：

- 0：一般來說都填 0，代表 (u, v) 之間沒有邊

- 負數 (如 -1)：和上面用途類似，只是當帶權圖中的邊為非負整數時較常用
- 無限大 INF：我們不可能真正存進無限大，只是我們可以算出一個題目怎樣也達不到的數字，來當作無限大，存這個數字通常是要套用一些特定的演算法

接著我們看有向圖 9.8，因為有向邊 $[u, v]$ 只能從頂點 u 走到頂點 v ，因此 $A[u, v] = 1$ ，而我們不會修改到 $A[v, u]$ ，如範例 9.24。

至於帶權圖的話，通常我們會對 (u, v) 這條邊儲存 $A[u, v] = w(u, v)$ ，如範例 9.25，我們會直接儲存邊的權重，需要注意的是 $(3, 6)$ 這條邊，因為 $w(3, 6) = 0$ ，所以並不是空白格子都填 0 就結束了，主要是如何區別有邊和沒邊。

虛擬碼 9.26 表示我們建立一個無向圖的相鄰矩陣，第 05 行我們對相鄰矩陣 A 初始化，接著在第 09 行建邊，時間複雜度 $O(n^2 + m)$ ，通常一張簡單無向圖的邊數不超過 $n(n + 1) / 2$ 、有向圖不超過 $n(n + 1)$ ，因此 $m = O(n^2)$ ，所以建出一個相鄰矩陣的複雜度為 $O(n^2)$ 。

二、相鄰串列 (Adjacency Lists)

由於相鄰矩陣的缺點，在於頂點數很大時會浪費大部分的空間，此時便不適合使用相鄰矩陣，加上有些時候題目給的邊並沒有那麼多，於是我們便從「儲存邊」的方向下去思考。

相鄰串列是記錄點和邊之間的關係——記錄從頂點 u 出去的邊有哪些，因為這個方法多要使用動態陣列，因此多以 STL 呈現。

我們同樣從無向圖 9.21 開始觀察：

三、 邊列表 (Edge List)

四、 前向星法 (Forward Star)

五、 鏈式前向星

六、 綜合比較

第三節 遍歷與排序

一、 圖的遍歷 (Traversal of Graph)

二、 時間戳記法 (Time Stamp)

三、 有向無環圖 (Directed Acyclic Graph, DAG)

四、 拓撲排序法 (Topological Sort)

練習題

✓ UVa 10305: Ordering Tasks

第四節 最小生成樹 (Minimum Spanning Trees)

一、 問題定義

生成樹 (Spanning Tree) 是指在一張圖 G 上的生成子圖，且此生成子圖恰好是樹。而我們在討論最小生成樹 (Minimum Spanning Tree) 時，通常是在帶權圖 G_w 上討論。一般來說這張帶權圖是在邊上，「最小」的定義是：生成樹上所有邊權重總和為最小。

以一開始圖 9.22 為例，生成樹第一個條件是生成子圖，也就是子圖 H 的頂點集 V_H 恰好就是原圖 G 的頂點集 V_G ，其次這個生成子圖必須是樹，因此我們可以找出一棵生成樹。

二、Kruskal's 演算法

演算法 9.1 Kruskal's 演算法

```

1: procedure KRUSKAL( $G, \omega$ )
2:   for  $e \in E(G)$  do
3:      $L.push(e, \omega(e))$  ▷  $L$  是邊列表
4:   for  $u \in V(G)$  do
5:      $MAKESET(u)$  ▷ 利用并查集
6:   依照  $\omega(e)$ ，對  $L$  遞增排序
7:    $T_{MST} \leftarrow \emptyset$  ▷ 初始化 MST
8:   for  $i = 1 \cdots |L|$  do ▷ 依照邊的權重依序取出
9:      $L[i] = \{(u, v), \omega(u, v)\}$ 
10:    if  $FINDSET(u) \neq FINDSET(v)$  then
11:       $UNIONSET(u, v)$ 
12:       $\omega_{MST} \leftarrow \omega_{MST} + \omega(u, v)$ 
13:       $T_{MST} \leftarrow T_{MST} \cup \{L[i]\}$ 
14:    if  $|T_{MST}| < |V(G)| - 1$  then ▷ 少於  $|V(G)| - 1$  條邊代表不是樹
15:      return “不存在最小生成樹”
16:    return  $T_{MST}$ 

```

三、Prim's 演算法

練習題

✓ UVa [11631](#): *Dark roads*

✓ UVa [10034](#): *Freckles*

更多練習題

 UVa [908](#): *Re-connecting Computer Sites*

四、次小生成樹

練習題

✓ UVa 10600:

第五節 最短路徑問題

一、問題定義

二、Bellman-Ford 演算法

演算法 9.2 Bellman-Ford 演算法

```

1: procedure BELLMANFORD( $s, G, \omega$ )
2:                                     ▷  $path$  紀錄所有點到  $s$  當前的最短路徑
3:   for  $v \in V(G)$  do                                     ▷ 初始化  $path$ 
4:      $path[v] \leftarrow \infty$ 
5:    $path[s] \leftarrow 0$ 
6:   for  $i = 1 \cdots |V(G)| - 1$  do                         ▷ 執行  $|V(G)| - 1$  次迴圈
7:     for  $e = \langle u, v \rangle \in E(G)$  do                     ▷ 每次對所有邊擴張看看
8:       if  $path[u] + \omega(u, v) < path[v]$  then
9:          $path[v] \leftarrow path[u] + \omega(u, v)$ 
10:  for  $e = \langle u, v \rangle \in E(G)$  do                         ▷ 檢查有無負環
11:    if  $path[u] + \omega(u, v) < path[v]$  then                 ▷ 還能擴張代表不合理
12:      return “存在負環”
13:  return  $path$ 

```

練習題

✓ UVa 558:

✓ UVa 10449:

✓ UVa 10557:

✓ UVa 11280:

三、 SPFA

演算法 9.3 Shortest Path Faster Algorithm (SPFA)

```

1: procedure SHORTESTPATHFASTERALGORITHM( $s, G, \omega$ )
2:                                     ▷  $path$  紀錄所有點到  $s$  當前的最短路徑
3:   for  $v \in V(G)$  do                                     ▷ 初始化  $path$ 
4:      $path[v] \leftarrow \infty$ 
5:    $path[s] \leftarrow 0$ 
6:    $Q.push(s)$                                              ▷  $Q$  是 queue
7:   while  $Q.size() \geq 1$  do
8:      $u \leftarrow Q.front()$ 
9:      $Q.pop()$ 
10:    for  $v \in Adj(u)$  do                                   ▷ 所有和  $u$  相鄰的點
11:      if  $path[u] + \omega(u, v) < path[v]$  then
12:         $path[v] \leftarrow path[u] + \omega(u, v)$ 
13:        if  $v \notin Q$  then
14:           $Q.push(v)$ 
15:  return  $path$ 

```

練習題

✓ UVa 10986:

四、 Dijkstra's 演算法

練習題

✓ UVa 929:

✓ UVa 10801:

五、Floyd-Warshall 演算法

演算法 9.4 Floyd Warshall 演算法

```

1: procedure FLOYDWARSHALL( $G, \omega$ )
2:                                     ▷  $path$  紀錄點  $i$  到  $j$  當前的最短路徑
3:   for  $i \in V(G)$  do                                     ▷ 初始化  $path$ 
4:     for  $j \in V(G)$  do
5:       if  $i \neq j$  then
6:          $path[i][j] \leftarrow \infty$ 
7:       else
8:          $path[i][i] \leftarrow 0$                                ▷ 先初始化為 0，若後面有自環再討論
9:       for  $e = \langle i, j \rangle \in E(G)$  do
10:         $path[i][j] \leftarrow \omega(i, j)$ 
11:      for  $k \in V(G)$  do                                     ▷ Floyd-Warshall 本體
12:        for  $i \in V(G)$  do
13:          for  $j \in V(G)$  do
14:            if  $path[i][k] + path[k][j] < path[i][j]$  then
15:               $path[i][j] \leftarrow path[i][k] + path[k][j]$ 
16:      return  $path$ 

```

練習題

✓ UVa 821:

✓ UVa 11015:

第六節 圖的連通性 (Connectivity)

一、問題定義

二、關節點 (Articulation Point)

練習題

✓ UVa 315: Network

✓ UVa 10199: Tourist Guide

三、橋 (Bridge)

練習題

✓ UVa 610:

四、強連通分量 (Strong Connected Component, SCC)

在有向圖 G 中，我們定義：對於任意的頂點 u 、 v ，如果有一條路徑從 u 到 v ，且也有一條路徑可以從 v 到 u ，我們就稱 G 為**強連通圖** (Strong Connected Graph)。

這種定義很像是有向圖版本的連通性，由於有向圖的邊只能單向連通，而無向圖的邊可以雙向連通，所以連通性在有向圖中並不夠好：對於任意兩點 u 、 v ，只要有路徑就稱為連通圖，要是在有向圖中存在一個 u ，它走得到頂點 v ，但從 v 點走不回去 u 點呢？見下圖。

從圖可以看出，連通圖的定義在有向圖中並不夠好，因此我們需要一個更強、但類似的定義使得我們可以說明「在有向圖連通」這件事——只因我們認為「無向圖連通」可以到處走來走去，那麼在有向圖應該也會有類似的性質。

定義強連通後，它和無向圖連通有類似的概念：我們有強連通子圖，代表一張子圖是強連通的。當然，我們並不在乎一個子圖是否強連通，更多時候我們更想知道它極大的強連通子圖——也就是**強連通分量** (Strong Connected Component, SCC)。

SCC 其實可以看做有向圖中的環，如上圖，我們會重視 SCC 主要的一個原因是，當我們在解決有向圖上的問題時，這些問題往往在無環有向圖 (也就是 DAG 圖) 中會有很好的解決方法，因此一張有向圖，我們只要針對有向環做處理，我們就可以把一張圖轉化為一棵樹。

練習題

✓ UVa 11770: *Lighting Away*

✓ UVa 11504: *Dominos*

更多練習題

 UVa 11709: *Trust groups*

 ZJ b201: *F. 國家*

五、雙連通分量 (Bi-Connected Component, BCC)

練習題

✓ Ural 1557:

第七節 匹配問題 (Matching)

一、問題定義

練習題

✓ UVa 10004:

✓ UVa 11706:

演算法 9.5 Tarjan's 演算法

```

1: procedure SCC-TARJAN( $G$ )
2:    $Stamp \leftarrow 0$                                 ▷ 用變數  $Stamp$  計算進入節點的次數
3:    $S \leftarrow \emptyset$                                 ▷  $S$  是 stack
4:   for  $u \in V(G)$  do                                ▷  $A_{SCC}$  記錄縮點後的結果
5:      $A_{SCC}[u] \leftarrow u$                                 ▷ 初始化
6:      $dfn[u] \leftarrow 0$                                 ▷ 記錄節點  $u$  被 DFS 的時間
7:      $low[u] \leftarrow 0$                                 ▷ 記錄到目前為止  $u$  可抵達最小的  $dfn$ 
8:   for  $u \in V(G)$  do
9:     if  $dfn[u] = 0$  then
10:      DFSVISIT( $u$ )
11:   return  $A_{SCC}$ 
12:
13: procedure DFSVISIT( $u$ )
14:    $Stamp \leftarrow Stamp + 1$ 
15:    $dfn[u] \leftarrow Stamp$ 
16:    $low[u] \leftarrow Stamp$ 
17:    $S.push(u)$ 
18:   for  $v \in Adj(u)$  do                                ▷ 遍歷所有與  $u$  相鄰的點
19:     if  $dfn[v] = 0$  then                                ▷ 沒有遍歷過的節點，就沒有  $dfn$  和  $low$ 
20:       DFSVISIT( $v$ )
21:     if  $v \in S$  then
22:        $low[u] \leftarrow \text{MIN}(low[u], low[v])$ 
23:   if  $dfn[u] = low[u]$  then                                ▷ 此節點不能再抵達編號更小的節點
24:     repeat
25:        $u' = S.top()$ 
26:        $S.pop()$ 
27:        $A_{SCC}[u'] \leftarrow u$ 
28:   until  $u' = u$ 

```

二、交錯軌道 (Alternating Path) 和增廣路徑 (Augmenting Path)

(一) 交錯軌道

(二) 增廣路徑

三、最大二分匹配 (Maximum Bipartite Matching)

演算法 9.6 最大匹配演算法

練習題

✓ UVa 663: *Sorting Slides*

✓ UVa 670: *The dog task*

✓ UVa 10080: *Gopher II*

✓ UVa 10092: *The Problem with the Problem Setter*

✓ UVa 10349:

✓ UVa 11138: *Nuts 'n' Bolts*

✓ UVa 11418: *Clever Naming Patterns*

✓ UVa 10122:

✓ UVa 10804: *Gopher Strategy*

✓ UVa 11262: *Weird Fence*

✓ UVa 10615:

✓ UVa 11159: *Factors and Multiples*

✓ UVa 11419:

四、 最大權二分匹配——匈牙利演算法 (Hungary Algorithm)

第八節 網路流 (Network Flow)

一、 問題定義

二、 最大流量問題 (Maximum Flow)

三、 Ford-Fulkerson 演算法

四、 Edmond-Karp 演算法

練習題

✓ UVa 820:

五、 網路流建模

練習題

✓ UVa 10330: *Power Transmission*

✓ UVa 11045: *My T-shirt suits me*

六、 最小切割 (Minimum Cut)

七、 最大流最小切割 (Max-flow Min-cut)

八、 最小成本最大流 (Min-cost Max-flow)

第十章

計算幾何

第一節 向量

一、向量運算

向量是數學上常用的一種思維，它包含了大小及方向兩種概念，無論是在二維平面、或是三維空間中，我們通常以一個箭頭來表示一個向量。舉例來說，小明今天往北走了 5 公里，那麼「往北走了 5 公里」這句話就代表著一個向量——大小是 5 公里而方向是北方；另一個例子，假設小明往東北走了 10 公里，那麼這時大小是 10 公里、方向是東北方。

爲了以數學來描述向量，我們利用座標平面，來描述二維平面上向量所擁有的特性。假設 x - y 平面上，以 x 軸正向爲東邊， y 軸正向爲北邊，以第一個例子來說，假設小明一開始在原點 $(0,0)$ 的位置，往北走 5 公里後，小明的位置會在 $(0,5)$ 的地方，這時 $(0,0)$ 和 $(0,5)$ 間的直線距離就會「產生」一條向量，方向是從 $(0,0)$ 指向 $(0,5)$ ，如圖 10.1。

讀者可能會以爲，向量就是小明所走過的路徑。其實並不盡然，假設小明的好朋友小華，他昨天從原點往東邊走了 4 公里到達 $(4,0)$ ，休息一下過後，接著再往北走 3 公里抵達 $(4,3)$ ，依照上面的法則，我們可以像圖 10.2 一樣畫出兩條藍色的，分別是 $(0,0)$ 指向 $(4,0)$ ，和 $(4,0)$ 指向 $(4,3)$ 的兩條向量。

但是總和來說，我們可以看做是起點是 $(0,0)$ ，終點是 $(4,3)$ 的一條向量，如圖 10.3 的紅色向量。我們把兩條藍色向量「合併」成一條紅色向量，這就是向量的「加

法」。向量加法可以看做是兩段路程相加所造成的結果。

爲了精準描述向量，我們定義向量——意即箭頭——的兩端，三角形的箭頭表示「終點」的概念，而另外一端是「起點」的概念，如圖 10.4。

那麼我們用一組數字 (x,y) 來描述一個向量：定義一個向量的起點在 $(0,0)$ 的位置，終點在 (x,y) 的位置，那麼這個向量的值爲 (x,y) 。這麼定義的好處是：我們可以很方便作向量上的一些運算，與一開始提到向量是有「大小」和「方向」的概念有些微不同，但是經由一些運算，我們也是能求出向量所擁有的大小和方向。

我們回頭看小明，小明原本起點在 $(0,0)$ 的位置，終點在 $(0,5)$ ，因此小明距離原點的向量爲 $(0,5)$ ；而小華一開始在原點 $(0,0)$ 的位置，經過 $(4,0)$ 後，最後「終點」在 $(4,3)$ ，注意！向量只看「起點」和「終點」的直線距離，不管路程爲何，因此小華總和的向量爲 $(4,3)$ ，即是圖 10.3 的紅色向量。

我們再來仔細分析小華的例子：我們剛剛從圖 10.3 得到的結論——圖中紅色向量是兩條藍色向量「相加」的結果。那我們要怎樣知道這兩條藍色向量「相加」會得到後來的紅色向量呢？我們先從那兩條藍色向量開始觀察。第一條藍色向量，是從原點走到 $(4,0)$ ，因此第一段向量的值毫無意外是 $(4,0)$ 。

第二段向量就麻煩了，它的起點是 $(4,0)$ ，終點是 $(4,3)$ ，跟我們原來向量起點在 $(0,0)$ 的定義並不相同，這時我們如果想要知道它的向量，我們就得「假裝」把它的起點搬回原點，如圖 10.5，我們把整段向量搬回起點時，終點位置的座標也會跟著改變，不難想像整段向量是「平移」回原點的，因此起點座標從 $(4,0)$ 變成 $(0,0)$ ，那終點座標如何改變呢？

我們觀察起點座標的數值變化，我們發現：當起點從 $(4,0)$ 搬回 $(0,0)$ ， x 座標減了 4。因此，終點座標在平移的過程中， x 座標也減去 4，最後終點座標變成了 $(0,3)$ ，由此可知第二段向量的值就是 $(0,3)$ 。

我們知道兩段藍色向量的值 $(4,0)$ 和 $(0,3)$ ，我們可以發現，這兩段向量的 x 值相加，並且 y 值相加，就會是最後的紅色向量 $(4,3)$ 。其實，向量加法也可以從原本的敘述中找出：小華原本在原點，一開始先「向東走 4 公里」（第一段藍色向量），接著「往北走 3 公里」（第二段藍色向量），因此最後結果就會變成——

$$(4,0) + (0,3) = (4+0, 0+3) = (4,3)$$

更廣義來說，如果兩個向量 (x_1, y_1) 和 (x_2, y_2) 相加，他的結果為：

$$(x_1 + x_2, y_1 + y_2)$$

此外，剛剛我們還運用到了「向量減法」的概念，同樣的道理，如果有兩個點 (x_2, y_2) 和 (x_1, y_1) 相減，他的結果會是：

$$(x_2 - x_1, y_2 - y_1)$$

向量減法可以看做是以 (x_1, y_1) 做為起點、 (x_2, y_2) 做為終點時向量的值。

二、 內積

是向量的一種乘法，常常被稱為內積 (Inner Product)。

演算法 10.1 內積

```
procedure DOT( $O, A, B$ ) ▷  $O, A, B$  是點座標
  return  $(A.x - O.x) \times (B.x - O.x) + (A.y - O.y) \times (B.y - O.y)$ 
```

演算法 10.2 向量內積

```
procedure DOTVECTOR( $A, B$ ) ▷  $A, B$  是向量
  return  $A.x \times B.x + A.y \times B.y$ 
```

三、 外積

常常被稱為外積 (Outer Product)。

演算法 10.3 外積

```
procedure CROSS( $O, A, B$ ) ▷  $O, A, B$  是點座標
  return  $(A.x - O.x) \times (B.y - O.y) - (A.y - O.y) \times (B.x - O.x)$ 
```

演算法 10.4 向量外積

```
procedure CROSSVECTOR( $A, B$ )  
    return  $A.x \times B.y - A.y \times B.x$ 
```

▷ A, B 是向量

第二節 計算幾何初步

一、基本操作

計算幾何中，通常要注意的有兩件事：第一、就是浮點數誤差；而第二、就是我們如何減少計算量來提高求出答案的效率。

首先，我們會遇到的問題是：怎樣判斷一個浮點數為零？

演算法 10.5 判斷浮點數為零

```
procedure IsZERO( $x$ )  
    if  $-\epsilon < x < \epsilon$  then  
        return TRUE  
    else  
        return FALSE
```

▷ x 是一個浮點數

▷ ϵ 是你「定義」的一個很小的常數

索引

2-著色圖, 24

二分圖, 24

偶環, 23

入分支度, 19

出分支度, 19

分支度, 18

前向星, 17

前驅, 17

匯點, 19

圖, 15

圖論, 15

大小, 15

奇環, 23

子圖, 20

完全二分圖, 25

完全圖, 24

封閉道路, 22

導出子圖, 20

帶權圖, 25

強連通分量, 33

強連通圖, 33

後向星, 17

後繼, 17

星, 17

最小生成樹, 28

有向圖, 18

有向邊, 16

權重, 25

混合圖, 18

源點, 19

無向圖, 17

無向邊, 15

無環圖, 23

環, 23

生成子圖, 20

生成樹, 28

相接, 16

相鄰, 16

端點, 16

簡單圖, 20

簡單無向圖, 20

簡單路徑, 22

總分支度, 18

自環, 20

葉子, 25

行跡, 22

複圖, [20](#)
路徑, [22](#)
迴路, [23](#)

連通, [25](#)
連通元件, [25](#)
連通圖, [25](#)
連通子圖, [25](#)

道路, [22](#)
邊, [15](#)
邊導出子圖, [21](#)
重邊, [20](#)
開放道路, [22](#)
階數, [15](#)
頂點, [15](#)

插圖目錄

4.1	一維陣列(左)和二維陣列(右)，或稱數組、矩陣	8
9.1	圖不考慮大小、長度、形狀	16
9.2	無向邊和有向邊	16
9.3	編號	17
9.4	前驅和後繼	17
9.5	有向圖	18
9.6	計算分支度	18
9.7	入分支度和出分支度	19
9.8	複圖和簡單圖	20
9.9	子圖	21
9.10	生成子圖	21
9.11	導出子圖	22
9.12	行跡	23
9.13	路徑	23