第 1 章　通論

發現了！

– Archimedes

1.1.　圖論緣起

歷史證明， 那些贊助過數學——精密科學的共同源頭——發展的帝國君王， 也是統治最英明、榮耀最長久的君王； Euler（1707∼1783）所處的時期正是這樣的一個年代。 1727 年經過 Bernoulli 兄弟的引薦， 俄國皇后 Catherine 召見 Euler 到聖彼得堡， 先是讓他在 1731 年成爲物理教授， 兩年後又增添了數學教授的職位。 在這樣優厚的環境下， Euler 得以全神投入他所喜愛的數學研究； 令人惋惜的是， 因爲努力過頭了， 使得他在 1735 年時右眼失明。 在這種情況下， 不可能會有人繼續放任他無止盡地工作下去， 所以 Euler 的太太便強迫安排全家一起到山明水秀的 Königsberg 去度假， 希望能抒解一下他的工作壓力。 故事就這樣開始了。

東普魯士的 Königsburg 市（今俄羅斯的 Kaliningrad 市）有一條 Pregel 河（今 Pregolya 河）流經， 河的中心有兩個小島， 小島與河的兩岸有七座橋相連接（如圖 1.1 所示）。 當地流傳著這樣一則謎題， 要如何才能從某一塊土地開始、將每一座橋恰好經過一次。

觀察力略爲敏銳的遊客在經過一些嘗試之後， 很快就會感覺到這應該是不可能的。 漫步在橋上的 Euler， 完全把太太安排度假的事情拋在腦後， 邊走還邊構思著一篇關於力學的重大論文； 想累了， 就改來想想到底要怎麼去說服 Bernoulli， 關於所有正整數

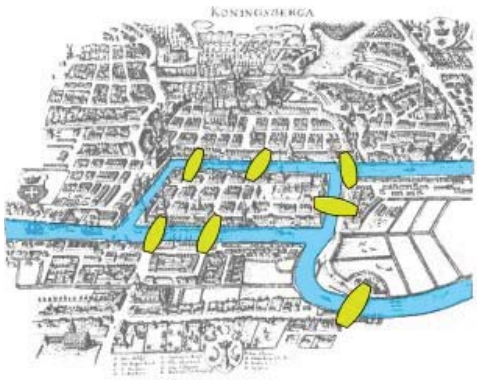


圖 1.1: Königsburg 地圖。

平方倒數和下列的結果

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \ldots = \frac{\pi^2}{6}。$$

當然， 走在這些橋上不可能不去想關於這個小鎮所流傳的七橋問題， 但他不只想要解決七橋問題， 更希望能夠找到所有類似問題的一般性結論。 忽然靈機一動， 他下橋快步回旅館。 小孩們看到爸爸回來都很高興， 靠攏過來依偎著他， 而 Euler 則一面抱起其中最小的一個， 一面鋪紙振筆疾書； 於是， 一個不僅解決了七橋問題、同時更一般的結論就這樣出現了。 1736 年 Euler 的這篇文章 [9] 奠基了圖論這門學問[1]。

從 1736 年到 1936 年這整整兩百年， 可以說是圖論的春秋戰國時代， 不同領域的人在他們各自的崗位上， 以不同的名稱、不同的內容， 探索與 Euler 發現的圖一樣的概念（參見 [3]）。 一直到 1936 年， Kőnig[2] 關於圖論的第一本著作《有限圖與無限圖的理論》[14]， 正式宣告圖論這門學問誕生。 這之後的七十多年以來， 各式各樣圖論的書籍， 以幾何級數的速度產生， 我們這本書就是其中之一。 圖論是一門探討物件之間如何關連的學問。 在前面的故事中我們提到的七橋問題， 其本質上就是在討論陸地之間

[1]Euler 遊橋的故事係屬杜撰， 但他右眼失明、 寫出一篇關於力學的重大論文、 推導出正整數平方倒數和的答案、 喜歡小孩、 寫出關於七橋問題的論文等， 都是歷史上記載的事實。

[2]Kőnig Dénes（依照匈牙利的習慣， 姓氏是擺在前面， 這和中國人的習慣相同； 在歐洲， 匈牙利是唯一有這種習慣的國家） 以及其父親爲了方便在歐洲其他國家的期刊（主要是德文期刊）發表文章， 都是把姓氏中的雙重音字母 ő（常見於匈牙利文）改成了曲音字母 ö（常見於德文）、 以 König 爲名發表； 另一位本書也會多次提到的著名匈牙利數學家 Erdős Paul 早年也是寫作 Erdös， 但是逐漸地隨著排版機制的進步、 和正名運動聲浪的興起， 最終是恢復了 Erdős 的正確寫法。 爲了因應這個潮流以及向這位偉大數學家致敬， 本書中都將以 Kőnig 稱之。

（透過橋樑）的連結關係。 如果撇開 Königsberg 小鎮的房子跟具體的地理形狀不管，我們可以把整個鳥瞰圖抽象成如圖 1.2 中間的型式； 如果我們再進一步、 更簡單地只用一個點來表示一塊陸地， 並且依照它們之間的橋樑連接關係用邊連接起來， 那最後我們會得到如圖 1.2 右邊的示意圖。

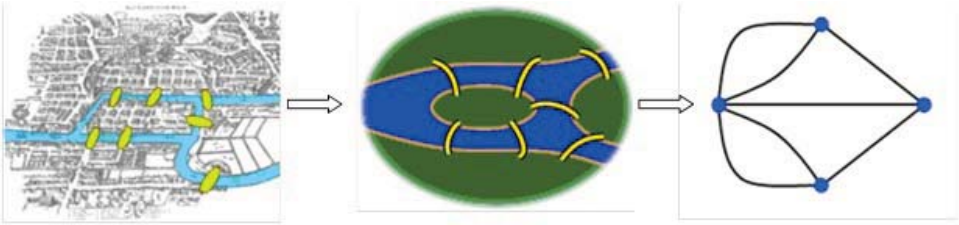


圖 1.2: Königsburg 七橋問題的抽象化。

於是， Königsberg 七橋問題就相當於是要求我們從圖中的某一點出發、 不重複亦不遺漏地將每一條邊都走過一次。 對於所有類似於這種由點和邊連結而成的圖， 都可以問同樣的問題， 這就是河與橋問題的一般化。 圖論要研究的對象就是像這樣的一種事物。 其他例子如電腦網路的連結、 道路鋪設、 人與人之間的互動關係， 都可以透過類似的示意圖來表示。 在這種示意圖當中， 以個別的點表示物件， 而以邊連接標示它們之間的關係。 因此， 簡單來說， 所要討論的圖就是一些點跟邊的組合。 在這種圖當中，長度跟位置都不是最重要的， 關心的重點只有物件之間怎麼連接的問題。 當年 Euler 將這種學問視爲一種特殊的新型幾何， 並提到是由 Leibniz 最早考慮這種幾何的， 稱之爲「位置的幾何學（geometria situs）」。 後來這種不考慮距離只考慮構造的幾何朝著兩種不同的模式發展， 一種變成了拓撲學， 而另外一種則變成了圖論。 兩者雖然有非常密切的關連， 但圖論不同於拓撲的地方是， 圖論當中沒有複雜的幾何結構， 只有點跟邊兩種最單純的元素， 所以其討論往往是非常具離散意味的； 事實上， 圖論通常也是歸類在離散數學的分支之中。 這一章首先介紹圖論的基本要素， 接著討論如何用圖論的觀點解決 Königsberg 七橋問題， 以及一些衍生出來的相關議題。

1.2. 圖的定義

前面一節已經約略地提到了什麼是圖論當中所指的圖， 下面將比較正式地描述「**圖**」（graph）的定義。 首先考慮一般最常遇到的圖， 這種圖只有有限個點， 任兩點之間都

至多只有一條邊相連， 而且不會有一條邊的兩端是連接同一個點， 這樣的圖稱爲**簡單圖**（simple graph）。 嚴格來說， 簡單圖就是一個有序對 $G = (V, E)$， 其中 V 是非空的有限集， 其元素稱爲**頂**點（vertex）， 簡稱爲**點**， E 是一些 V 的相異二元無序對的集合， 即 $E \subseteq \{e\colon e \subseteq V, |e| = 2\}$， 其元素稱爲**邊**（edge）。 有的時候， 用 $V(G)$ 表示圖 G 的點集、 而 $E(G)$ 則表示其邊集。 方便起見， 也經常將一條邊 $e = \{u, v\}$ 直接寫成 uv， 因此， uv 跟 vu 是相同的。 此時說 u 跟 v 是 e 的**端點**（end vertices），也說 e 和 u（及 v）**相連**（incident）， 並說 u 和 v **相鄰**（adjacent）、 或者說 u 是 v 的**鄰居**（neighbor）。 以 $N_G(v)$ 表示圖 G 中點 v 的全體鄰居所構成的集合， 如果討論的圖很明確時可以簡寫作 $N(v)$。 爲了視覺上的方便， 通常會將圖具體地畫出來， 如圖 1.3 表示 $V = \{a, b, c, d, e\}$ 及 $E = \{ab, ae, bc, be, cd, de\}$ 的圖。 如果點的名字暫時不重要時， 也可能不將它們標出來， 一直等到有需要的時候再標示； 如果只有少數點的名字必須用到， 也可能只標示出那些點的名字。

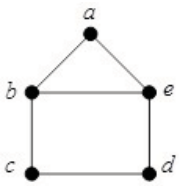


圖 1.3: 有 5 個點及 6 條邊的圖。

可以透過改變集合 V 跟 E 的屬性以得到各種變型的圖的概念。 例如， 若允許 V 是無窮集， 則得到**無限圖**（infinite graph）。 而如果允許 E 是重集（multi-set，即同樣的元素可以重複出現的集合）， 就有所謂**重圖**[3]（multigraph）， 也就是兩點之間可以有多條邊相連的情況， 這樣的邊稱爲**重邊**（multiedge）； 例如前面七橋問題當中所考慮的圖就是重圖。 如果更進一步允許邊 $\{u, v\}$ 是重集， 也就是可以有 $u = v$ 的話，則得到所謂的**近圖**（pseudograph）， $\{u, u\}$ 這樣的邊稱爲**迴邊**（loop）。 如果將邊改成有序對 (u, v)， 那麼 uv 跟 vu 就表示不同的邊， 我們稱這樣的圖爲**有向圖**（directed

[3]可以用另外一種等價的定義方式， 以通常的集合結構來刻畫出重圖。 定義一個圖是由 $G = (V, E, f)$ 構成、 其中 $V = \{v_1, v_2, \ldots, v_n\}$ 是點集、 $E = \{e_1, e_2, \ldots, e_m\}$ 是邊集， 而 f 則是一個從 E 到 $\{\{v_i, v_j\}\colon 1 \le i < j \le n\}$ 的映射、 以表示每一條邊的頂點爲何。 如果 f 是一個嵌射（injection、即不同的元素必映至不同的元素）， 那對應的 G 就是簡單圖； 而如果 f 不是嵌射的話那 G 就是重圖。 此外， 如果將上面的「$<$」改成「$\le$」， 定義出來的就會是近圖。

graph 或簡稱 digraph）。 畫有向圖的時候， 邊通常會加上箭頭以表示方向。

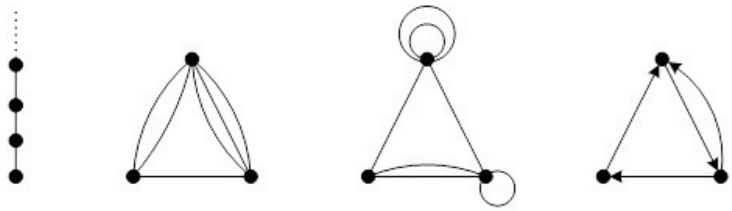


圖 1.4: 無限圖、重圖、近圖、有向圖的例子。

前述各種不同種類的圖都是圖， 不過往後如果沒有特別強調是近圖或者重圖等的話， 「圖」通常都是指簡單圖。 接著來看一些基本的圖的例子。 **n-完全圖**（n-complete graph）是指具有 n 個頂點且任兩點均有邊相連的圖， 即點集 $V = \{1, 2, \ldots, n\}$ 及邊集 $E = \{ij \colon i, j \in V \text{ 且 } i \neq j\}$ 的圖。 這個圖一般記做 K_n。

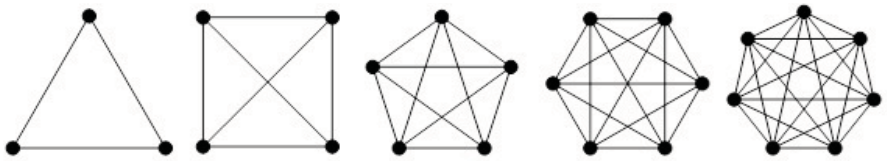


圖 1.5: 完全圖 K_3、K_4、K_5、K_6 及 K_7。

二分圖（bipartite graph）是指一個圖 G、 其點集可以表示成兩個互斥集合 X 跟 Y 的聯集， 使得所有 G 的邊都只存在於 X 跟 Y 之間； 也就是說，X 裡面的點都兩兩不相鄰， Y 裡面的點也一樣。 此時會說 X 跟 Y 是 G 的**二部份**（bipartition）。 例如， **完全二分圖** （complete bipartite graph）$K_{m,n}$ 是指一個點集分成兩部分、 一部份有 m 個點而另一部份有 n 個點， 並將這兩部分任一對點都予以連邊的圖。

圖 1.6: 完全二分圖 $K_{4,3}$ 與 $K_{5,2}$。

二分圖的觀念也可以推廣成爲 **k-分圖**（k-partite graph）， 即一個點集可以分成 k 個部分、 使得個別部分內部都沒有連邊的圖。 **完全 k-分圖**（complete k-partite

graph）相對地就是把這 k 個部分兩兩之間的點全部連邊的圖， 通常以 $K_{n_1,n_2,\ldots,n_k}$ 來表示完全 k-分圖， 其中 $n_1, n_2, \ldots, n_k$ 分別是每個部分的大小。 圖 1.7 顯示的是 $K_{2,2,2}$ 的例子。

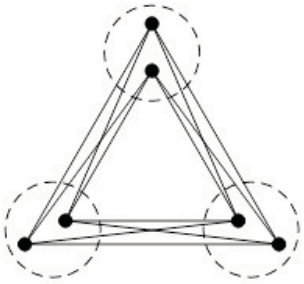


圖 1.7: 完全 3-分圖 $K_{2,2,2}$，其中以虛線圈出此圖的三部分。

n-**圈**（n-cylcle）C_n 是由 n 個點組成的圈狀圖， 其點集爲 $V = \{1, 2, \ldots, n\}$ 而邊集爲 $E = \{12, 23, \ldots, (n-1)n, n1\}$。 其中 C_3 又常被稱爲**三角形**（triangle）。

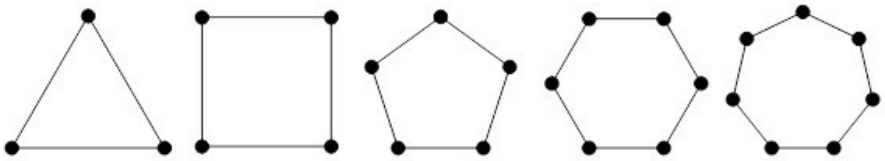


圖 1.8: 圈圖 C_3、C_4、C_5、C_6 及 C_7。。

此外，n-**路徑**（n-path）P_n 是由 n 個點組成的徑狀圖， 其點集爲 $V = \{1, 2, \ldots, n\}$ 而邊集爲 $E\{1, 23, \ldots, (n-1)n\}$。

圖 1.9: 路徑圖 P_2、P_3、P_4、P_5 及 P_6。

不難看出 $K_1 = P_1$、$K_2 = P_2$、$K_3 = C_3$， 因此這些記號都是可以交換使用。 對於任何一個圖， 通常不只一種方法可以把它畫出來， 但是這些不同的圖像所代表的內在結構其實都是相同的， 例如圖 1.10 中的三個圖雖然乍看之下並不相同， 但是其點跟邊之間的連結關係都是一樣的 （例如在三個圖當中，點 1 都是和點 2、點 5 以及點 6 相鄰，等等）， 它們都有一個共同的特別名字， 叫做 **Petersen 圖**。

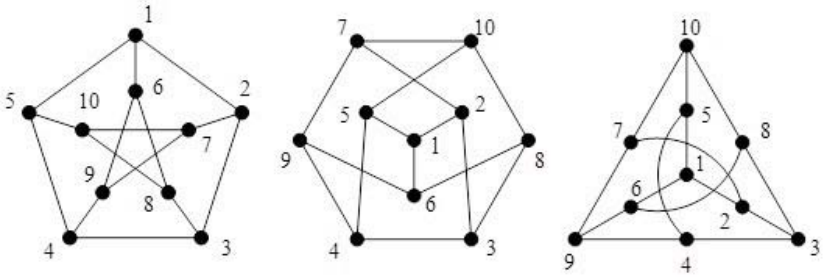


圖 1.10: 三圖同構，都是 Petersen 圖。

這種情況下會說這些圖是**同構**（isomorphic）的。 精確而言， 所謂兩個圖 G 和 H 同構， 是指在 $V(G)$ 和 $V(H)$ 之間存在一個一對一映成函數 f， 使得 $uv \in E(G)$ 若且唯若 $f(u)f(v) \in E(H)$。 圖 G 和 H 同構記作 $G \cong H$。

如果 $V(G) \subseteq V(H)$ 且 $E(G) \subseteq E(H)$， 則說 G 是 H 的**子圖**（subgraph）， 而 H 爲 G 的**父圖**（supergraph）， 這種關係記作 $G \subseteq H$； 如果進一步有 $V(G) = V(H)$， 則稱 G 爲 H 的**生成子圖**（spanning subgraph）、 或 G 是 H 的**因圖**（factor）。 若 G 爲 H 的子圖但不等於 H， 則說 G 是 H 的**眞子圖**（proper subgraph）。

如果 $S \subseteq V(G)$，$E_S = \{uv \in E(G) \colon u \in S, v \in S\}$， 則稱圖 $G[S] = (S, E_S)$ 是 G 的一個（**點**）**導出子圖**（(vertex-)induced subgraph）， 並說 $G[S]$ 是由 S 所導出（induce）的。 如果 G 沒有導出子圖與 H 同構， 我們稱 G 是 **H-免除的**（H-free）。

如果 $S \subset V(G)$[4]， 則稱 $G[V(G) \backslash S]$ 是 G 的**刪除圖**（deletion）， 簡記作 $G - S$。 如果 S 只有一個元素 v， 則 $G - \{v\}$ 亦可簡記作 $G - v$。 如果 $E' \subseteq E(G)$， 則 $G - E'$ 是指圖 $(V(G), E(G) - E')$； 如果 E' 只有一條邊 e， 類似地我們也常將 $G - \{e\}$ 簡記作 $G - e$。 相對地， 如果把另一個圖 H 加入 G、 但是不把 G 跟 H 之間連任何邊， 那麼得到的新圖稱爲它們的**互斥聯集**（disjoint union）或**和圖**（sum），記做 $G + H$。

一個跟這些概念有關的未解問題如下所述。

猜想 1.1.（Ulam 猜想[5] [19]） 假如 $n \geq 3$， 圖 G 跟 H 各自都有 n 個頂點，分別爲 u_i 和 v_i（$1 \leq i \leq n$）。 如果對於每一個 i 都有 $G - u_i \cong H - v_i$，則 $G \cong H$。

[4]本書用 $A \subseteq B$ 表示 A 爲 B 的子集， 而以 $A \subset B$ 表示 A 爲 B 的眞子集、 亦即 $A \subseteq B$ 但 $A \neq B$。 有些作者習慣用 $A \subset B$ 表示 A 爲 B 的子集， 本書將不使用這種記號， 請特別注意。

[5]這個猜想乍看之下好像很直觀地會對， 不過稍微多想一下或許就會發現其眞正的困難處在於， 因爲並沒有要求 $G - u_i$ 跟 $H - v_i$ 必須按照頂點對應的編號作同構映射。

一個近圖 $G=(V,E)$ 當中， 點 v 的**度數**（degree）是指與此點相連的邊數， 其中迴邊 $\{v,v\}$ 對 v 的度數的貢獻爲 2。 以 $\deg_G(v)$ 表示 v 的度數， 或當討論的圖很明確時可以簡寫作 $\deg(v)$。 度數爲奇數的點稱爲**奇點**（odd vertex）， 度數爲偶數的點則稱爲**偶點**（even vertex）， 度數爲 0 的點稱爲**孤立點**（isolated vertex）， 而度數爲 1 的點則稱爲**葉**[6]（leaf）。 往後分別用 $\delta(G)$ 和 $\delta(G)$ 兩個記號表示 G 中頂點的最小度數和最大度數。 如果一個圖 G 中的每個點的度數都相同， 例如對所有 $v\in V(G)$ 都有 $\deg(v)=k$ 時， 就說 G 是 **k-正則的**（k-regular）。 一個跟度數有關的性質是當年 Euler 在他 1736 年的那篇文章 [9] 中提到的， 可以說是跟圖論有關的第一個定理。

性質 1.2. 對任一近圖 $G=(V,E)$， 恆有 $\sum_{v\in V}\deg(v)=2|E|$。

這個性質在直觀上很容易理解， 因爲計算左邊的和時， 每數一個點的度數、 就會把跟它相連的邊都算一次， 這麼一來， 最後每一條邊都將恰好被數兩次， 因爲每條邊都恰與兩點相連 （雖然可能是同一點， 但這並不影響）。 這樣的論證方法叫做**雙邊計數**（two-way counting）， 是一種很有用的方法。 這個性質的一個立即推論如下所述。

推論 1.3. 任一近圖的奇點數目必爲偶數。

1.3. 路徑

到目前爲止還沒有說到要如何將七橋問題的本質刻畫出來。 現在來看如何描述「在圖上行走」的這個行爲。

一個近圖 G 中的一條**道路**（walk）是指一條點邊相間的序列 $v_0e_1v_1e_2v_2\dots e_kv_k$， 其中點 v_{i-1} 和 v_i 是邊 e_i 的端點。 在簡單圖的情況中， 由於 v_{i-1} 和 v_i 會決定唯一的 e_i， 因此這條道路可以只用 $v_0v_1v_2\dots v_k$ 表示。 分別稱 v_0 和 v_k 爲這條道路的**起點**和**終點**， k 稱爲道路的**長**（length，特別注意 k 是邊數而非點數）。 有時爲了強調起點和終點， 也會稱之爲 v_0-v_k 道路。 當 $v_0=v_k$ 的時候， 稱這條 v_0-v_k 道路爲**封閉**（closed）道路， 否則若 $v_0\neq v_k$ 則稱爲**開放**（open）道路。

一條**行跡**（trail）是指一條邊不重複的道路， 而一條**路徑**（path）則是指一條點不重複的道路； 顯然所有的路徑都是行跡， 且行跡都是道路， 但反之則未必成立。 一個

[6]會有這樣的名稱是跟第 3 章將會提到的「樹」有關。

圈（cycle）是指除了 $v_0 = v_k$ 以外、 點跟邊都兩兩相異的 v_0-v_k 道路。 先前曾提到的 P_n 和 C_n 分別就是只由一條路徑和一個圈所構成的圖。

現在回頭來看七橋問題。

對於一個近圖而言， 如果一條行跡包含了該圖所有的邊， 我們就稱之爲一條 **Euler 行跡**（Euler trail）， 而封閉的 Euler 行跡則稱爲 **Euler 迴路**（Euler tour）。 這個定義並未要求 Euler 行跡必須通過所有點， 因此孤立點的存在並不影響圖有沒有 Euler 行跡； 事實上， 一個有邊的近圖如果有 Euler 行跡， 它就不會通過任何一個孤立點（如果存在的話）。 七橋問題所要問的就是， 下面的這個重圖存不存在一條 Euler 行跡？

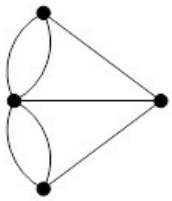


圖 1.11: 七橋問題所對應的重圖。

答案是並不存在。 要解釋這一點， 只需要有點的度數觀念就夠了。 事實上， 假如存在一條開放的 Euler 行跡， 則當沿著這條行跡從某一條邊進入某點之後、 必定要從另外一條邊走出來， 因此與每一點相連的邊都可以依照進去和出來的關係配成對， 除非該點正好就是行跡的起點或終點； 此時會有一條邊是當作出發的邊或者結束的邊。 而對於 Euler 迴路而言， 這兩條邊又恰可以配對。 因此就得到下面這樣的結論。

性質 1.4. 如果一個近圖中存在一條 *Euler* 迴路， 則必定每一點都是偶點。 如果一個近圖中存在一條開放的 *Euler* 行跡， 那麼必定恰有兩點是奇點（且分別爲行跡的起點跟終點）， 其餘都是偶點。

回過頭來看圖 1.11， 很不巧地， 這個重圖剛好四個點通通都是奇點， 這就表示裡面不可能存在開放的 Euler 行跡， 更沒有 Euler 迴路， 故七橋問題無解。

不過， 上面的結論反過來不見得會對， 因爲就算所有的點都是偶點， 可是如果考慮的圖分裂成兩堆、 中間根本沒有道路連接， 那麼當然不可能有 Euler 迴路存在。 因此， 還必須考慮到圖的連通性。

對於一個近圖 G 來說， 如果裡面任兩點之間都有道路連接， 則說 G 是**連通的**（connected）。 其實， 這個定義也可以改寫成「任兩點之間都有路徑連接」， 因爲有如下的性質。

性質 1.5. 對於近圖 G 中的兩點 u 和 v， 存在一條 u-v 路徑的充分必要條件是存在一條 u-v 道路。

證明： 必要性是顯然的， 因爲所有的路徑都是道路。 反過來， 如果存在一條 u-v 道路， 根據良序原理， 可以找到一條長度最短的 u-v 道路 $W\colon v_0e_1v_1e_2v_2\dots e_iv_i\dots e_jv_j\dots e_kv_k$； 如果存在 $i<j$ 使得 $v_i=v_j$， 那麼將 $e_{i+1}v_{i+1}\dots e_jv_j$ 這一段從 W 當中去掉， 將會得到一條更短的 u-v 道路， 這與 W 最短的假設矛盾， 所以 W 本身就是一條路徑。 ■

上述證明中取最短道路 W 的方法， 等同於做數學歸納法， 它就是所謂的**良序原理**（well-ordering principle）， 也就是說， 任何一個由一些自然數組成的非空集合一定有一個最小元素， 在圖論中常常使用這種方法來證明定理。 也就是， 取最大或最小反例， 再導出矛盾， 以此證明定理。

於是， 剛才提到的基本顯然事實就可以描述如下。

性質 1.6. 近圖 G 中存在 *Euler* 行跡的必要條件是， 兩個非孤立點之間必有一條道路。

證明： 取 G 的一條 Euler 行跡 W， 任一非孤立點 v 必和某邊 e 相連， 因爲 e 在 W 上， 所以 v 也在 W 上。 因此，兩個非孤立點之間必有一條道路。 ■

一個近圖 G 中**極大**（maximal）的連通子圖稱爲其**連通部分**（connected component）。 這邊所謂「極大」是指， 如果 C 是 G 的連通子圖、 而且不存在 G 的另一個連通子圖 H 使得 $C\subseteq H$ 但 $C\neq H$， 那就說 C 是極大的連通子圖。 也就是說， 極大的概念是在子圖關係底下決定出來的大小順序。 而如果我們要談的是數量上的大小順序概念（例如根據點或邊的數目多寡）， 就會用**最大**（maximum）這個詞。 類似地可以區隔**極小**（minimal）和**最小**（minimum）之間的概念差異。 最大一定是極大， 但反之未必。

顯然， 如果 C_1 和 C_2 都是 G 的連通子圖而且 $V(C_1)\cap V(C_2)\neq\emptyset$， 則其**聯集**

（union）$C_1 \cup C_2 = (V(C_1) \cup V(C_2), E(C_1) \cup E(C_2))$ 也會是 G 的連通子圖。 於是，有如下的性質。

性質 1.7. 近圖 G 的所有連通部分的點集是 $V(G)$ 的一個**分割**（*partition*， 即一些聯集爲 $V(G)$ 但兩兩不相交的非空集合）、 其邊集也會是 $E(G)$ 的一個分割。

如果圖 G 的一系列的子圖 $H_1, H_2, \ldots, H_k$ 的邊集恰構成 $E(G)$ 的分割， 那會說這些子圖構成了 G 的一個**分解**（decomposition）。 因此， G 的所有連通部分就構成了 G 的一種分解。

在談圖的連通部分時， 也有人會採用另外一種等價的說法， 講起來也許比較麻煩，但卻是比較高觀點的看法。 首先， 在集合 $V(G)$ 中定義這樣的一種**關係**（relation）：

$$x \sim y \iff \text{存在一條 } x\text{-}y \text{ 道路。}$$

很容易驗證， 由這個方式定義出來的 $\sim$ 是一種**等價關係**（equivalence relation）， 即它滿足下面三個性質。

1. **反身性**（reflexive）：若 $x \in V(G)$，則 $x \sim x$。
2. **對稱性**（symmetric）：若 $x \sim y$，則 $y \sim x$。
3. **遞移性**（transitive）：若 $x \sim y$ 且 $y \sim z$，則 $x \sim z$。

因此， 可以應用這個等價關係將 $V(G)$ 分成若干個**等價類**（equivalence class）$V_1, V_2, \ldots, V_r$， 而易知這些等價類所導出的子圖 $G[V_1], G[V_2], \ldots, G[V_r]$ 其實就是 G 的連通部分。

圈是一個相當重要的構造， 在後面的章節中還會出現很多應用到圈的理論。 在本節最後部分， 先舉一個簡單的例子。 如果一個圈的邊數爲奇數， 就稱之爲**奇圈**（odd cycle）， 反之就稱爲**偶圈**（even cycle）。 類似地可以定義奇道路、奇行跡等等。 關於二分圖， 有下述這樣的基本性質。

定理 1.8. 圖 G 是二分圖， 若且唯若它沒有奇圈。

證明： 如果 G 是二分圖， 則任何圈都只能是 $x_1y_1x_2y_2 \ldots x_ky_kx_1$ 這樣的型式， 也就是二部份 X 跟 Y 中的點交錯出現， 所以 G 中的圈都必定是偶圈。 反過來， 如果 G 中沒有奇圈， 則要將點分成 X 和 Y 兩部分。 易知如果 G 的每個連通部分都是二分圖、

那 G 也必定是二分圖， 所以只要討論 G 本身是連通的情況即可。 先任意選取一個點 v， 然後對於任何一個點 u， 由於 G 是連通的， 必存在一條最短的 v-u 道路， 設其長度爲 k， 如果 k 是偶數就將 u 分到 X 類， 反之若 k 是奇數就將 u 分到 Y 類。 這麼一來每一個點都會被唯一地分到某一類。 現在， 假設 X 或 Y 內部有邊爲 uu'， 那麼將剛才選取的最短 v-u 道路（其長度爲 k）、 邊 uu'、 以及倒過來的最短 u'-v 道路（其長度爲 k'、與 k 同奇偶）串在一起， 就會得到一條長度爲 $k+1+k'$ 的封閉奇道路， 可是根據習題 1.6， 此時 G 中必有奇圈， 矛盾。 所以 X 跟 Y 內部都沒有邊。 ■

1.4. Euler 圖

如果近圖 G 中存在 Euler 迴路， 就稱 G 爲 **Euler 近圖**（Euler pseudograph）。 上一節已經舉出了一個近圖 G 爲 Euler 近圖的兩個必要條件 （性質 1.4 和 1.6）。 而有趣的是， 這兩個條件加起來剛好就構成了充分條件， Euler 當年在他的論文 [9] 當中就已經提到了這件事。 用嚴格的尺度來要求， 他的文章並沒有給出這個充分性的「完整」證明， 不過他已經用建構式的方式， 描述了證明的主要精神， 這樣其他人就可以將比較正式完整的證明寫出來， 而且， 他的方法也提供了求 Euler 迴路的演算法。 在那個年代， 像這樣在嚴謹度上有小瑕疵的討論比比皆是， 但是這並沒有構成數學發展的阻礙， 事實上， 十八世紀的數學家仍舊在這種風氣之下醞釀出大量深刻而有內涵的結論。 這個定理也是如此。

定理 1.9. 一個恰爲一孤立點或不含孤立點的近圖 G 爲 *Euler* 近圖的充分必要條件是， G 是連通的、 而且其中所有的點都是偶點。

證明： 必要性已經由性質 1.4 和 1.6 給出。 而關於充分性， 底下準備給出三種不同的證法。 本質上它們都是對 G 的邊數（設爲 m）做數學歸納法。

證法一： 當 $m=0$ 的時候， 定理顯然成立。 現在假設 $m>0$， 且定理對所有邊數小於 m 的近圖都成立。 從某一個非孤立點 u 開始、 在「邊不重複」的前提下盡量行走、 直到不能再繼續爲止， 這給出一條行跡 W。 此時 W 的終點 v 必定也是 u， 不然的話 W 將只用到 v 的奇數條相連邊， 但 v 是一個偶點， 必定還有某條邊可以繼續走， 這跟 W 的取法矛盾。 於是， 因爲 $u=v$， W 當中與任何一點相連的邊數一定是偶數， 所

以 $G' = G - E(W)$ 中各點皆爲偶點， 且 G' 的各個連通部分的邊數皆小於 m。 由歸納法假設， 這些連通部分都有 Euler 迴路， 於是可以將 G' 的連通部分各自的 Euler 迴路合併到 W 當中以得到 G 的一條 Euler 迴路。 亦即， 先沿著 W 行走， 而一旦碰到 G' 的某個連通部分， 就先繞道將該連通部分的 Euler 迴路走完， 然後再繼續沿著 W 走， 這就會得到 G 的一條 Euler 迴路。 值得注意的是， 因爲 G 是連通的， 所以 G' 的任何一個連通部分一定會碰到 W。 □

證法二： 在 G 中選取一條最長的行跡

$$W\colon v_0e_1v_1e_2v_2\ldots e_iv_ie_{i+1}v_{i+1}\ldots e_kv_k。$$

如果 $v_k \neq v_0$， 則 W 只用到奇數條與 v_k 相連的邊， 所以存在某一邊 $e_{k+1} = \{v_k, v_{k+1}\}$ 不在 W 當中， 這導致 $We_{k+1}v_{k+1}$ 是一條更長的行跡， 矛盾； 所以必有 $v_k = v_0$。

接著宣稱 W 就是一條 Euler 迴路； 因爲， 如果 W 沒有用完 G 所有的邊， 則因爲 G 是連通的， 所以 G 中必存在某一邊 $e = \{v, v_i\} \notin W$， 其中 v_i 在 W 當中。 但這麼一來

$$vev_ie_{i+1}v_{i+1}\ldots e_kv_ke_1v_1\ldots e_iv_i$$

會是一條比 W 更長的行跡， 又導致矛盾。 因此 W 本身就是 G 的一條 Euler 迴路。 □

證法三[7]**：** 當 $m = 0$ 的時候， 定理顯然成立。 現在假設 $m > 0$。 當 G 的邊都是迴邊的時候， 所有的迴邊都通過同一點， 因此易看出 G 有 Euler 迴路。 如果 G 至少有某一邊不是迴邊， 例如 $e_1 = uv$ 不是迴邊， 則因爲 u 是偶點， 必定還有 $v' \neq u$ 使得 $e_2 = uv' \in E(G)$。 令 G' 是 G 去掉 e_1 和 e_2 並加入 $e = vv'$ 之後的近圖， 其邊數 $m' = m - 1$。 在這個更動之下， 除了 u 的度數減少 2 之外， 其他點的度數都沒有改變， 因此 G' 的點仍都是偶點。

接著分成兩種情況， 一種是 G' 只有一個連通部分， 另一種情況是 G' 有兩個連通部分。 無論是哪一種情況， v 跟 v' 都會被包含在一個邊數少於 m 的連通部分 C_1 當中， 而根據歸納假設， C_1 有 Euler 迴路 W。 於是， 在第一種情況中， 可以將 W 當中的 e 換成 e_1ue_2 而得到 G 的 Euler 迴路； 而在第二種情況中， 根據歸納假設， 包含 u 的連通部分 C_2 也會有 Euler 迴路 W'（不失一般性可以假設 W' 是 u-u 迴路），於是將 W 中的 e 換成 $e_1W'e_2$ 即得到 G 的 Euler 迴路。 □ ■

[7]這個證法由台大數學系陳聖華提供。

由這個定理可以得到的立即推論如下。

推論 1.10. 一個不含孤立點的近圖 G 中存在開放 *Euler* 行跡的充分必要條件是， G 是連通的、 而且其中恰有兩個奇點。

證明： 必要性一樣已經由性質 1.4 和 1.6 給出。 要證明充分性， 假設 G 中僅有的兩個奇點分別是 u 跟 v， 無論這兩點是否相鄰， 都將 G 再加上一條新的邊 $e = uv$ 而得到 G'， 於是 G' 的點全都是偶點。 根據定理 1.9，G' 有 Euler 迴路 W， 不失一般性可以假設 W 的最後一條邊就是 e， 亦即 $W : ve_1v_1e_2 \dots uev$。 於是， 將 W 去掉最後的 ev 就得到 G 中的一條開放 Euler 行跡， 其起點爲 v、終點爲 u。 ■

以實用的觀點來說， 光是知道判別的準則通常還不夠， 最重要的應該是， 實際給了一個符合定理 1.9 條件的 G， 如何具體地把 Euler 迴路找出來。 所以還要進一步來研究**演算法**（algorithm）的問題。 第 2 章會更仔細地介紹演算法的基本概念， 在這裡只是要來看怎麼樣透過固定的流程來解決問題而已。

上面關於該定理的三種證法當中， 前兩種本質上是相同的， 只是差在第一種證法用的是最一般的數學歸納法、 而第二種證法則是改以良序原理的方式呈現。 以演算法的觀點來說， 第一種證法是比較能夠改寫成演算法的； 第二種證法雖然簡潔， 但跟第三種證法一樣， 都不利於改寫成演算法。

雖然第一種證法比較能夠改寫成演算法， 但是如果眞的照那樣的方法做， 結果並不會很有效率。 首先注意到， 依照這個方法的構想， 一開始會從某個點出發， 盡量地「能走就走」， 最後回到出發點； 此時假如已經把所有的邊都走過那當然最好， 但是， 很有可能偏偏走了捷徑、 很快就回到起點， 留下了一大堆還沒走過的邊， 如此一來還必須先將這些邊扣除、 造出新的圖 G'， 然後再用遞迴程式去做。 這種不斷製造新圖的行爲將會是非常沒效率的方法。 例如以圖 1.12 來說， 我們有可能從 v_1 出發、 走了 $v_1e_1v_2e_3v_3e_2v_1$ 之後就結束， 偏偏這剛好是最短的道路。

關於這個問題， 現今較常見的 Fleury 演算法是在類似的「能走就走」的作法之下， 稍微加上一點選擇的條件， 以避免抄捷徑的情況發生。 他的想法是這樣， 假設 G 本身沒有孤立點； 當走了 i 條邊之後得到 $W\colon v_0e_1v_1e_2v_2 \dots e_iv_i$ 時， 下一條邊 $e_{i+1} = v_iv_{i+1}$ 的選法是、 找一條尙未走過的邊 e_{i+1} 使得 $G - \{e_1, e_2, \dots, e_{i+1}\}$ 是連通的（除了孤立點以外）。 可以證明， 照這樣做確實可以走過 G 所有的邊； 但是在執行

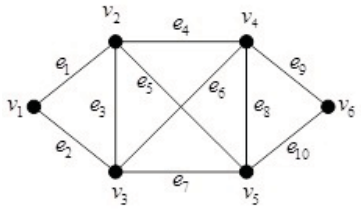


圖 1.12: Euler 圖的例子。

上， 要驗證一個圖是否連通也是一件花時間的事情， 更何況每走一步就要檢查一次連通性， 所以這看起來也不是一個好方法。

其實， 只要將證明一的方法稍微做一點修改， 就可以得到下面這個有效的演算法。 這個方法是在「能走就走」的原則之下， 加上「不能走就回頭到一個較早的點， 從未走過的邊走完之後再插入」的新規則。 下面仍用圖 1.12 來說明。

首先選定 v_1 出發， 能走就走； 假設像剛才一樣走了 $v_1e_1v_2e_3v_3e_2v_1$， 最後走到 v_1 走不下去了， 那麼就沿著這條行跡的最後一點往前找， 第一個點是 v_3， 而 v_3 還有相連的邊可以繼續再走， 於是我們就從 v_3 出發隨便走， 例如走出 $v_3e_6v_4e_8v_5e_7v_3$ 乃至不能再走。 將這一段新的行跡取代原先的行跡中 v_3 的位置， 就得到

$$v_1e_1v_2e_3(v_3e_6v_4e_8v_5e_7\underline{v_3})e_2v_1\text{，}$$

接著輪到 v_3 這點走不下去， 於是就再從這點往前找， 發現 v_5 這點還可以再走， 於是再從 v_5 走出 $v_5e_5v_2e_4v_4e_9v_6e_{10}v_5$， 並再次插入得到

$$v_1e_1v_2e_3v_3e_6v_4e_8(v_5e_5v_2e_4v_4e_9v_6e_{10}\underline{v_5})e_7v_3e_2v_1\text{，}$$

現在是輪到 v_5 走不下去， 於是再次從這點往前找還能走的點， 可是這次發現， 一路往回走直到發點 v_1 之後， 每一點都不能再走了， 這表示這條行跡就是一個圖 1.12 的一條 Euler 迴路。

1.5. Euler 迴路的應用

在舉例說明 Euler 迴路的應用之前， 先將前面的結果推廣到有向近圖的情況。 首先， 在有向近圖當中， 關於道路、 行跡、 Euler 迴路的定義都跟近圖相同， 只是當我們

提到 $v_{i-1}e_iv_i$ 的時候， $e_i = (v_{i-1}, v_i)$ 是有序對。 其次， 說一個有向近圖是**強連通的**（strongly connected）是指對任意兩點 x 和 y， 均存在一條由 x 到 y 的道路[8] ； 而一個有向近圖的**強連通部分**（strongly connected component）是指極大的強連通有向子近圖。 一個點 v 的**出度**（out-degree）是指 v 連出去的邊 vv' 的數目， 記做 $\deg^+(v)$， 而 v 的**入度**（in-degree）則是指 v 連進來的邊 $v'v$ 的數目， 記做 $\deg^-(v)$。 類似地， 我們會以 $N^+(v)$ 和 $N^-(v)$ 表示所有從 v 連出去的鄰居的集合和連進 v 的鄰居的集合。

性質 1.11. 對任一個有向近圖 $G = (V, E)$， $\sum_{v\in V} \deg^+(v) = \sum_{v\in V} \deg^-(v) = |E|$。

證明： 這是因爲每一條邊都只會在計算入度和出度時各被數一次。 ■

相對於 Euler 迴路的定理如下， 其證明和無向近圖的時候一樣， 此處省略其證明。

定理 1.12. 一個恰爲一孤立點或不含孤立點的有向近圖 G 有 *Euler* 迴路的充要條件是， G 是強連通的、 而且其中每一點 v 恆有 $\deg^+(v) = \deg^-(v)$。

接著來談一個有向近圖 Euler 迴路的有趣應用， 即所謂的 **de Bruijn 序列**。

首先，考慮二進位的情況， 即一切東西都是 0 跟 1 組成的， 不過並不準備把它們當數字使用、 而當成是字母看待。 請問有多少種方法可以用這兩個字母拼成長度是 2 的序列？ 很顯然是 00、01、10、11 四種。

考慮 0011 這個序列， 取出所有相鄰 2 個字母組成的子列， 發現裡面剛好出現了 00、01 跟 11， 而如果把這個序列的尾巴跟頭接起來， 那 10 也出現了； 四種組合恰好都在這個序列當中出現一次。 像這樣的序列 0011， 就稱爲是一個 **de Bruijn 2-序列**。

一般而言， 令 $\Sigma = \{0, 1, \ldots, \sigma-1\}$， 其中 $\sigma \geq 2$。 對自然數 n， 一個 **de Bruijn n-序列**是指某一個 Σ 上的序列 $a_0a_1\ldots a_{L-1}$[9]， 使得對任一個 Σ 上長度爲 n 的序列 $b_0b_1\ldots b_{n-1}$ 都存在唯一的 i 使得 $b_0b_1\ldots b_{n-1} = a_ia_{i+1}\ldots a_{i+n-1}$， 其中 a_j 的下標取模 L。 剛才舉出的例子就是 $\sigma = 2$ 時的 de Bruijn 2-序列。 又例如同樣取 $\sigma = 2$、 則

[8]值得注意的是， 如果仍舊把這個關係記做 $\sim$， 則一般而言在有向近圖中未必會有對稱性， 即 $x \sim y$ 並不保證有 $y \sim x$ 成立； 但若有強連通性就一定會成立了。

[9]正式來說， 序列應該表示爲 $(a_0, a_1, \ldots, a_{L-1})$， 而省略分號的 $a_0a_1\ldots a_{L-1}$ 稱爲**字串**。 傳統上， de Bruijn 序列都是用字串的方式書寫， 特別是當 $\sigma < 10$ 時， 這樣寫並不會造成混淆， 好處是用起來比較簡便； 但是當 $\sigma \geq 10$ 時， 正式的序列符號就有其必要了。

$n = 3$ 的一個例子爲 00011101， 這裡面恰好將每一個長度是 3 的二進序列都表現一次。顯然， 對於任何這樣的序列都一定要有 $L = \sigma^n$。

人們感興趣的問題是， 任意給定 σ 和 n， 是否存在這樣的序列？ 而若存在、 又要如何尋找？ 前述有向近圖的 Euler 迴路剛好就可以用來解決這個問題。

考慮一個有向近圖 $G_{\sigma,n}$， 其中

$$V(G_{\sigma,n}) = \{a_0a_1 \dots a_{n-2} \text{: 各 } a_i \in \Sigma\}、$$
$$E(G_{\sigma,n}) = \{(a_0a_1 \dots a_{n-2}, a_1a_2 \dots a_{n-1}) \text{: 各 } a_i \in \Sigma\}，$$

則 $G_{\sigma,n}$ 恰有 σ^{n-1} 個點和 $\sigma^n = L$ 條邊。 這個 $G_{\sigma,n}$ 是有 Euler 迴路的。 首先， 對每一點 $a_0a_1 \dots a_{n-2}$， 恰有 σ 種方法選取 a_{n-1} 使得 $(a_0a_1 \dots a_{n-2}, a_1a_2 \dots a_{n-1}) \in E(G_{\sigma,n})$， 因此 $\deg^+(v) = \sigma$； 同理有 $\deg^-(v) = \sigma$， 因此度數方面的條件滿足。 其次， 對於任兩點 $a_0a_1 \dots a_{n-2}$ 跟 $b_0b_1 \dots b_{n-2}$， 永遠存在這樣的一條道路

$$a_0a_1 \dots a_{n-2} \to a_1a_2 \dots a_{n-2}b_0 \to a_2a_3 \dots a_{n-2}b_0b_1 \to \dots a_{n-2}b_0b_1 \dots b_{n-3} \to b_0b_1 \dots b_{n-2}$$

連接著這兩點， 因此 $G_{\sigma,n}$ 是強連通的， 所以綜合起來就得到它有 Euler 迴路。

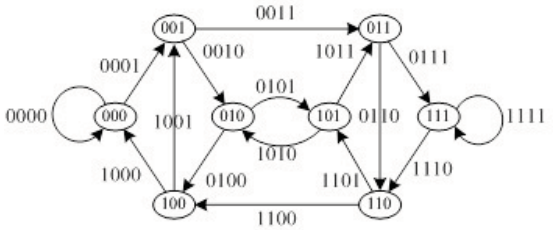


圖 1.13: 有向圖 $G_{2,4}$， 其邊 $(a_0a_1 \dots a_{n-2}, a_1a_2 \dots a_{n-1})$ 簡記爲 $a_0a_1 \dots a_{n-2}a_{n-1}$。

以圖 1.13 的 $G_{2,4}$ 爲例， 有這樣的一條 Euler 迴路

$$000 \to 000 \to 001 \to 010 \to 100 \to 001 \to 011 \to 110 \to$$
$$101 \to 010 \to 101 \to 011 \to 111 \to 111 \to 110 \to 100 \to 000。$$

如果把這樣的一條 Euler 迴路的每一點（除最後一點 000 以外）取最後一個字母串接起來， 結果其實正好就是所要的 de Bruijn 4-序列（爲什麼？）：0010011010111100。

於是， 可以將 de Bruijn 序列跟這種圖（稱爲 de Bruijn 圖）中的 Euler 迴路取得對應。 由上面的討論， 得到如下的結論。

定理 1.13. 對於任何的 $\sigma \geq 2$ 和 $n \geq 2$，*de Bruijn* n-序列都存在。

要具體找出這種序列， 只要把 de Brujin 圖畫出來， 然後按照找 Euler 迴路的方法來找就行了。 不過， 如果從演算法的角度來看， 按照這種想法， 需要構造出一個有 σ^{n-1} 點的有向近圖， 這會隨著 σ 和 n 而成長得非常急遽， 使得資料的儲存造成龐大的負擔。 於是， 要如何在不具體將圖構造出來的情況下找出 de Brujin 序列就變成了一個有趣的問題。

de Brujin 序列可能不只一種 （即便排除序列的輪換或鏡射以及字母的置換）； 例如上面的例子中， 也可以取出 0101101000011110 這樣的 de Bruijn 序列 （對應的是哪一個 Euler 迴路？）， 而這個跟剛才取得的序列本質上是不同的。

1.6. 度序列

一個圖的**度序列**（degree sequence）是指由此圖的度數所排成的序列， 例如圖 1.12 中的圖的度序列爲 $2, 4, 4, 4, 4, 2$[10]。 給定一個序列 d: $d_1, d_2, \ldots, d_n$， 如果存在 G 使得該序列是它的度序列， 我們就說 G **實現**（realize）d。 這樣的 G 不一定是唯一的（就同構而言）， 例如圖 1.14 就是一個例子， 其左邊跟右邊這兩個圖的度序列都是 $2, 2, 2, 1, 1$，但是兩者顯然不同構。

圖 1.14: 最小的例子使得一個度序列對應的圖不唯一。

如果 $d_1, d_2, \ldots, d_n$ 是一度序列， 則由性質 1.2 知道 $\sum_{i=1}^{n} \deg(v_i)$ 必定是偶數， 但是反過來， 即便序列 $d_1, d_2, \ldots, d_n$ 滿足和爲偶數， 也不見得能找得到一個圖 G 實現該序列， 例如若 $n = 1$，則 $d_1 = 2$ 不可能是一個圖的度序列。 不過如果允許 G 是近圖， 那倒是可以找到一個恰有一迴邊的近圖滿足所求。 一般而言， 容易得到近圖的度序列的充分必要條件。

[10]通常會把度序列由大到小排序地寫出來， 但這不是絕對必要的。 基本上， 除了證明時方便起見以外， 在講一個圖的度序列時， 並不在乎其順序。

定理 1.14. 非負整數序列 $d_1, d_2, \ldots, d_n$ 是某近圖的度序列，若且唯若 $\sum_{i=1}^{n} d_i$ 是偶數。

證明： 必要性已由性質 1.2 給出。要證明充分性，假設 $\sum_{i=1}^{n} d_i$ 爲偶數，則其中爲奇數的 d_i 的個數必須是偶數，不妨假設爲 $d_1, d_2, \ldots, d_{2r}$。考慮一個點集 $V = \{v_1, v_2, \ldots, v_n\}$ 的近圖，其中每一點 v_i 上恰有 $\lfloor d_i/2 \rfloor$ 條迴邊，且 $v_i v_{r+i}$ 對 $1 \le i \le r$ 而言都是邊；易看出這個近圖的度序列就是 d。 ■

如果限制 G 是重圖，度序列的充分必要條件就會稍微複雜一點。

定理 1.15. 負整數序列 $d\colon d_1 \ge d_2 \ge \ldots \ge d_n$ 是一個重圖的度序列的充分必要條件是，$\sum_{i=1}^{n} d_i$ 是偶數且 $d_1 \le \sum_{i=2}^{n} d_i$。

證明留給讀者做練習（參見習題 1.27）。如果更進一步要求 G 沒有迴邊又沒有重邊，那麼條件就會更加複雜了。稱一個非負整數序列是**圖序列**（graphic sequence），如果存在一個圖實現該序列的話。Erdős 和 Gallai [8] 在 1960 年給出下面的刻劃定理。早期的證明大都採用網路流的方法，參見 Berge 的書 [2] 的第 6 章；後來 Harary [12] 給出一個直接但略爲冗長的證明，這之後有各種簡化的證明 [6][16][17]。

定理 1.16.（Erdős-Gallai [8]） 非負整數序列 $d\colon d_1 \ge d_2 \ge \ldots \ge d_n$ 是圖序列的充分必要條件是，$s = \sum_{i=1}^{n} d_i$ 是偶數，且對 $1 \le i \le n$ 恆有

$$\sum_{i=1}^{k} d_i \le k(k-1) + \sum_{j=k+1}^{n} \min\{k, d_j\}。 \tag{1.1}$$

證明： ($\Rightarrow$) 設 $V(G) = \{v_1, v_2, \ldots, v_n\}$，其中 $\deg(v_i) = d_i$。同前，$\sum_{i=1}^{n} d_i$ 必爲偶數。對於任意 k，計算 $V_k = \{v_1, v_2, \ldots, v_k\}$ 這個點集的總度數時，可以把與這些點相連的邊分成兩類：一類是在 V_k 內部連接的邊，這種邊每條會被算兩次，所以總共最多有 $k(k-1)$ 條邊；另一類是從 V_k 連接到 $\overline{V_k} = \{v_{k+1}, v_{k+2}, \ldots, v_n\}$ 的邊，或者，反過來說是從 $\overline{V_k}$ 連到 V_k 也可以，注意到對於每一個 $\overline{V_k}$ 中的點 v_j 來說，它至多連 $\min\{k, d_j\}$ 條邊到 V_k，因此就得到 (1.1) 的不等式。

假設序列 d 滿足定理的條件，接著要用兩種方法來證明存在圖 G 實現 d。

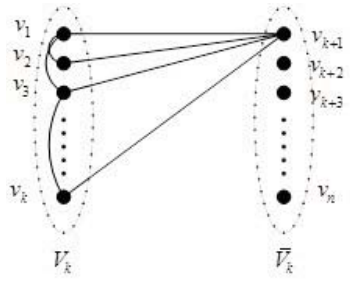


圖 1.15: 定理 1.16 必要性證明的示意圖。

證法一（Harary [12]）： ($\Leftarrow$) 對 n 做數學歸納法。 當 $n = 1$ 時， 式 (1.1) 爲 $d_1 \le 0 + 0$， 因此必有 $d_1 = 0$， 取 $G = K_1$ 便可滿足條件。 現在考慮 $n \ge 2$ 的情況， 由式 (1.1) 可知 $d_1 \le \sum_{j=2}^{n} \min\{1, d_j\}$， 因此 $d_2 \ge d_3 \ge \ldots \ge d_{1+d_1} \ge 1$。

令 d': $d'_1 \ge d'_2 \ge \ldots \ge d'_{n-1}$ 是將 $d_2 - 1, d_3 - 1, \ldots, d_{1+d_1} - 1, d_{2+d_1}, d_{3+d_1}, \ldots, d_n$ 排序後得到的度序列， 我們宣稱它滿足定理的條件。 首先， $\sum_{i=1}^{n-1} d'_i = \sum_{i=1}^{n} d_i - 2d_1$ 爲偶數。 接著， 透過一連串的討論可以驗證 d' 確實滿足式 (1.1) 的條件 （參見習題 1.28；亦可參考 [12]）， 由歸納法假設， 存在圖 G' 實現 d'。 令 G 爲 G' 中加入一個點使其連到度數爲 $d_2 - 1, d_3 - 1, \ldots, d_{1+d_1} - 1$ 的各點所成的圖， 於是 G 實現 d。 □

證法一最繁雜的部分在於驗證 d' 滿足式 (1.1)， 這是因爲 d' 必需重新排序、 而且下標由 i 變爲 $i - 1$ 之故。 下面這個 Choudum [6] 證法的修改版可以避免排序。

證法二： ($\Leftarrow$) 對 $s + n$ 做數學歸納法。 當 $s = 0$ 時， 取 $\overline{K_n}$ 便可滿足條件。 現在考慮 $s \ge 2$ 的情況， 不失一般性， 可以假設 $d_n \ge 1$， 因爲 $d_n = 0$ 時可以考慮去掉此項的序列 d'， 此一只有 $n - 1$ 項的序列顯然滿足式 (1.1) 的條件， 因此由歸納法假設， 存在圖 G' 實現 d'； 則 G' 中加入一個孤立點所成的圖 G 實現 d。

當 d 中各項不全相等時， 取 t 滿足 $d_1 = d_2 = \ldots = d_t > d_{t+1}$， 否則， 取 $t = n - 1$。 考慮 d': $d_1 = d_2 = \ldots = d_{t-1} > d_t - 1 \ge t_{t+1} \ge \ldots \ge d_{n-1} \ge d_n - 1$， 其各項和爲偶數。 接著驗證 d' 滿足式 (1.1)。 以下將用到 $\min\{a, b\} - 1 \le \min\{a, b - 1\}$。

當 $k = n$（或 $t \le k < n$） 時， 式 (1.1) 不等號左邊對 d' 的值比對 d 的值少 2（或 1）、 而不等號右邊對 d' 的值等於對 d 的值（或最多少 1）， 所以， d 的 (1.1) 導到 d' 的 (1.1)。 接下來討論 $1 \le k < t$ 的情況， 因爲 $\sum_{i=1}^{k} d_i = kd_1$， 所以只需驗證

$$kd_1 \le k(k-1) + \sum_{j=k+1, j\neq t}^{n-1} \min\{k, d_j\} + \min\{k, d_t - 1\} + \min\{k, d_n - 1\}。 \quad (1.2)$$

當 $d_1 \le k-1$ 時， (1.2) 顯然成立。 當 $d_1 = k$ 時， 如果 (1.2) 不成立， 因爲 $\min\{k, d_t - 1\} = d_1 - 1$， 只可能是 $k+1 = t = n-1$ 且 $d_n = 1$， 這樣一來， $\sum_{i=1}^{n} d_i = k(k+1)+1$ 爲奇數， 矛盾。 最後考慮 $d_1 \ge k+1$ 的情況。 令 r 是使得 $d_r \ge k+1$ 成立的最大下標， 此時 $r \ge t$， 而有 $\min\{k, d_t - 1\} = \min\{k, d_t\}$。 又 $r < n$， 否則 $\min\{k, d_n - 1\} = \min\{k, d_n\}$， 式 (1.2) 就是 (1.1)。 所以 $\min\{k, d_n - 1\} = d_n - 1$， 且對 $k+1 \le j \le r$ 恆有 $\min\{k, d_j\} = k$， 所以要證明 (1.2) 只需證明

$$kd_1 \le k(k-1) + \sum_{j=k+1}^{n-1} \min\{k, d_j\} + d_n - 1 = k(r-1) + \sum_{j=r+1}^{n-1} \min\{k, d_j\} + d_n - 1。 (1.3)$$

將式 (1.1) 中的 k 用 $k+1$ 取代， 因爲 $k+2 \le j \le r$ 恆有 $\min\{k+1, d_j\} = k+1$，

$$(k+1)d_1 \le (k+1)k + \sum_{j=k+2}^{n} \min\{k+1, d_j\} = (k+1)(r-1) + \sum_{j=r+1}^{n} \min\{k+1, d_j\}。 (1.4)$$

將式 (1.4) 乘以 $k/(k+1)$， 並利用 $\frac{k}{k+1}\min\{k+1, d_n\} < d_n$， 可以得到 (1.3)。

由以上證明， d' 滿足式 (1.1)， 因此由歸納法假設， 存在圖 G' 實現 d'。 當 $v_t v_n$ 不是 G' 的邊時， 在 G' 中加入 $v_t v_n$ 得到一圖 G 實現 d。 當 $v_t v_n$ 是 G' 的邊時， 因爲 $\deg_{G'}(v_t) = d_t - 1 \le n-2$， 所以存在不和 v_t 相鄰的點 v_i， 又因爲 $\deg_{G'}(v_i) \ge \deg_{G'}(v_n)$、 但 v_t 和 v_n 相鄰卻沒和 v_i 相鄰， 所以存在 v_j 和 v_i 相鄰卻不和 v_n 相鄰， 將 G' 中的 $v_i v_j$ 換成 $v_i v_t$ 和 $v_j v_n$ 得到一圖 G 實現 d。 □ ■

定理 1.16 雖然給出了一個完整的條件， 但它卻不利於改寫成演算法。 底下這個由 Havel [13] 及 Hakimi [11] 提出的說法則可以解決演算法上的需要。

定理 1.17. 設 $n \ge 2$， 則非負整數序列 $d\colon d_1 \ge d_2 \ge \ldots \ge d_n$ 是圖序列的充分必要條件是， 序列 $d'\colon d_2 - 1, d_3 - 1, \ldots, d_{1+d_1} - 1, d_{2+d-1}, d_{3+d-1}, \ldots, d_n$ 也是圖序列。

證明： ($\Leftarrow$) 跟定理 1.16 的第一個證法中最後的步驟一樣， 在 d' 對應的圖中加入一點並連邊就可以得到實現 d 的圖。

($\Rightarrow$) 假設 G 實現 d 且對應於序列的點爲 $v_1, v_2, \ldots, v_n$， 令 r 是使得 v_1 和 v_r 不相鄰的最小數。 不失一般性， 可以假設 G 是在所有實現 d 的圖當中使得 r 最大的一個。 如果 $r > 1 + d_1$， 表示 v_1 跟 $v_2, v_3, \ldots, v_{1+d_1}$ 都相鄰， 那直接取 $G' = G - v_1$ 就會是實現 d' 的圖； 而如果 $2 \leq r \leq 1 + d_1$， 則必存在 $s > 1 + d_1$ 使得 v_1 跟 v_s 相鄰； 同時因爲 $d_r \geq d_s$， 且 v_1 跟 v_s 相鄰但跟 v_r 不相鄰， 所有必存在 v_t 跟 v_r 相鄰但跟 v_s 不相鄰， 如圖 1.16 所示。

令 $\widehat{G}$ 是將 G 當中的 v_1v_s 和 v_rv_t 這兩條邊換成 v_1v_r 和 v_sv_t 的圖， 則易看出 $\widehat{G}$ 的度序列也是 d， 但是對應的 $\hat{r} > r$， 矛盾。 這就證明了結論。 ■

圖 1.16: 四點的連結情況。

1.7. 圖論用於 Brouwer 定點定理的證明

最後， 來看看如何將度數的觀念應用在證明分析學的 Brouwer 定點定理（Brouwer fixed point theorem）， 作爲本章的結尾。 假定讀者具備若干分析的基本知識， 例如連續函數與序列的收斂性等等。

考慮平面上的一個三角形 T， 將 T 做三角化（triangulation）使其成有限個三角形， 稱這個三角化爲單純（simplicical）三角化是指任兩個三角形若相交， 則只交在頂點或整條邊上， 見圖 1.17 (a)。 圖 1.17 (b) 則是一個非單純三角化的例子， 因爲左右兩半的三角形並沒有交在整條邊上。

把一個單純三角化中每一個頂點標爲 0, 1 或 2。 如果標號的方式滿足下面的兩個條件， 我們就說是**適當**（proper）的標號。

1. T 的三個頂點分別標爲 0, 1, 2。

2. T 的三邊上的其他頂點的標號與該邊兩端頂點其中之一相同。

例如，圖 1.18 (a) 就是一個適當標號的例子。

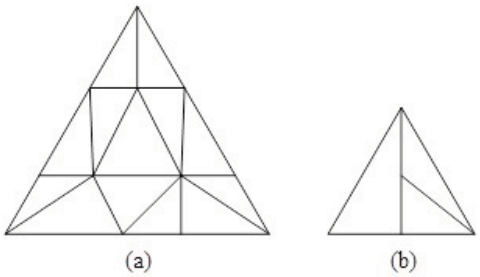


圖 1.17: (a) 單純三角化。(b) 非單純三角化。

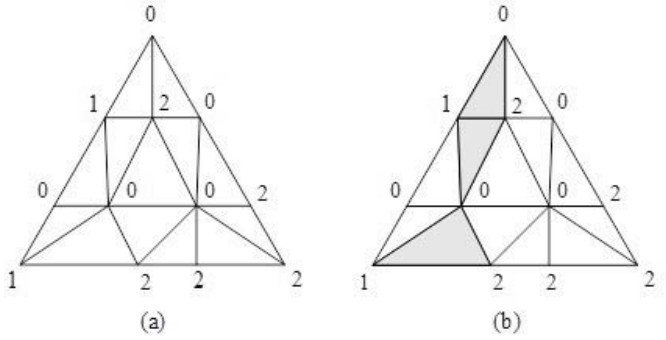


圖 1.18: (a) 圖 1.17 (a) 的一個適當標號。(b) 頂點標號相異的小三角形。

考慮圖 1.18 (a) 中所有頂點恰也是分別被標成 0, 1, 2 的小三角形， 一共有三個， 如圖 1.18 (b) 所示。 一般而言， 這樣的小三角形都會有奇數個。

引理 1.18. （Sperner 引理 [15]） 一個經適當標號的單純三角化的三角形中， 使得三個頂點都標不同號的小三角形的數目必爲奇數。 因此， 至少有一個這種小三角形。

證明： 令 T_0 是 T 外的大區域， 而 $T_1, T_2, \ldots, T_n$ 是 T 內的小三角形區域。 構造一個圖 G， 其點集爲 $V = \{v_0, v_1, v_2, \ldots, v_n\}$ 分別對應於各個區域， 而 v_i 和 v_j 有邊相連的條件是 $T_i \cap T_j$ 的兩頂點各標 0 跟 1。 此時， 對於每一個小三角形 T_i， 有下面三種可能。

1. 沒有任何兩個頂點分別被標爲 0 和 1，此時 $\deg(v_i) = 0$。
2. 三個頂點被標爲 $\{0, 0, 1\}$ 或 $\{0, 1, 1\}$，此時 $\deg(v_i) = 2$。
3. 三個頂點被標爲 $\{0, 1, 2\}$，此時 $\deg(v_i) = 1$。

也就是說， 一個小三角形的三頂點標號相異、 若且唯若對應的頂點是奇點。 現

在， 注意到 T 上連接 0 和 1 的那條邊當中有奇數段的頂點爲 0 和 1（爲什麼？）， 於是 $\deg(v_0)$ 會是奇數， 因爲 T_0 只有在這邊才有機會和小三角形共用頂點爲 0 和 1 的邊。 但是我們知道任何圖的度數和都是偶數， 所以必定有奇數個小三角形對應的頂點是奇點， 這就得到了引理的結論。 ■

Sperner 引理和下述的 Brouwer 定點定理也可以推廣到高爲度的情況。 也有討論有方向型態的 Sperner 引理。

定理 1.19.（Brouwer 定點定理、二維情形） 任何三角形區域到自己的連續映射一定有一個定點。

證明： 假設 f 是從三角形區域 T 映射到自己的連續函數， 其中 T 的三個頂點是 x_0、x_1、x_2， 則 T 中任何一點 x（作爲二維向量）都可以唯一地寫成 $x = a_0x_0 + a_1x_1 + a_2x_2$， 其中 $a_i \geq 0$ 且 $\sum_{i=0}^{2} a_i = 1$。 現在改以這個坐標 (a_0, a_1, a_2) 表示 x， 並且把 f 表示成 $f(a_0, a_1, a_2) = (a_0', a_1', a_2')$。

定義 $S_i = \{(a_0, a_1, a_2) \colon a_i' \leq a_i\}$、 $0 \leq i \leq 2$。 因爲每個點 (a_0, a_1, a_2) 都滿足 $\sum_{i=0}^{2} a_i = \sum_{i=0}^{2} a_i' = 1$， 於是每個點都會屬於某個 S_i。 目標是要說明 $S_0 \cap S_1 \cap S_2$ 非空； 此時， 易知該集合中的點滿足 $(a_0, a_1, a_2) = (a_0', a_1', a_2')$， 也就是一個 f 的定點。

考慮 T 的一系列單純三角化 $T^{(1)}, T^{(2)}, T^{(3)}, \ldots$， 其中 $T^{(n)}$ 內最大小三角形的最長邊長 ℓ_n 當 $n \to \infty$ 時趨近於零。 對於每一個三角化 $T^{(n)}$， 考慮將各個頂點按照它們屬於那一個 S_i 加以標號， 不過這些點當然有機會同時屬於兩個 S_i（如果同時屬於三個 S_i，那證明就已經結束了）， 所以我們要稍加決定如何標號。 注意到對於 T 的頂點、例如 $(1, 0, 0)$， 其函數值的第一個座標不可能變得更大， 所以它一定屬於 S_0， 我們就將它標爲 0。 類似地另外兩個頂點分別被標爲 1 和 2。 接著， 對於 T 的邊上的點而言， 它有其中一個座標爲 0， 這個座標不可能變得更小， 所以它一定可以被標成和該邊的兩端點其中之一同號。 至於 T 內部的頂點隨便選一個號標上去即可。 於是， 就將 $T^{(n)}$ 的各個頂點按照它們所屬的某個 S_i 適當地加以標號完畢。

因此， 根據 Sperner 引理， 每一個單純三角化 $T^{(n)}$ 當中都至少會有一個小三角形、 不妨記作 $A_0^{(n)}A_1^{(n)}A_2^{(n)}$， 使得 $A_i^{(n)} \in S_i$。 由於 T 是一個有限閉區域， 根據分析中的 Bolzano-Weierstrass 定理， $(A_0^{(n)})_n$ 一定有一個收斂到 T 中某一點的子序列

$(A_0^{(n_k)})_k$， 不妨假設 $A_0^{(n_k)} \to A_0$。 但是因爲 $T^{(n)}$ 中的小三角形邊長越來越小， 得到

$$\lim_{k\to\infty} A_i^{(n_k)} = \lim_{k\to\infty} (A_0^{(n_k)} + (A_i^{(n_k)} - A_0^{(n_k)})) = A_0\,;$$

然而， 因爲 f 是連續函數， 所以每個 S_i 都是閉集， 由 $A_i^{(n_k)} \in S_i$ 就得到 $A_0 \in S_i$ 對 $i = 0, 1, 2$ 都成立。 這樣就證明了定理。 ■

1.8. 習題

1.1. 試證明， 假設把 8×8 的西洋棋盤上位於同一條對角線上的兩個頂角格子去掉（於是剩下一個只有 62 格的棋盤）， 則這個棋盤沒辦法分割成若干個 1×2 的長方形。 並請利用同樣的論證描述在二分圖當中的一般性結論。

1.2. 一個圖 $G = (V, E)$ 的**補圖**（complement）是指 $\overline{G} = (V, \overline{E})$， 其中 $\overline{E}$ 是由所有 G 所不包含的邊所構成， 即 $\overline{E} = \{uv\colon u \neq v, uv \notin E\}$。 一個圖 G 稱爲**自反同構**（self-complement）是指 $G \cong \overline{G}$， 例如 P_4 就是一個自反同構的圖。 試證對於自然數 n， 存在 n 點的自反同構圖之充分必要條件爲 $n \equiv 0, 1 \pmod 4$。 （提示：若要構造實例，不妨考慮將 P_4 的構造加以推廣）

1.3. 試證明， 如果完全圖 K_n 可以分解成一些三角形， 則 $n \equiv 1, 3 \pmod 6$。

1.4. 假設 $2n$ 個人（$n \geq 2$）當中每個人至少和其他 n 個人認識。 試證明， 其中至少有四個人、 使得這四個人能夠圍著圓桌而坐、 讓每個人兩旁的人都是他認識的人。

1.5. 平面上有 n 個相異點， 已知任兩點的距離至少爲 1。 試證明， 最多只有 $3n$ 對點的距離恰爲 1。 （提示：考慮將這些點距離恰爲 1 者連邊。 請問在所給的條件之下， 每個點的度數至多爲多少？）

1.6. 試證明， 奇封閉道路必包含奇圈。 偶封閉道路是否一定含圈？

1.7. 一個圖的 **腰圍**（girth）是指其最短圈的長度 （如果沒有圈， 方便起見定義腰圍爲 ∞）。 試證明， 腰圍爲 4 的 k-正則圖至少有 $2k$ 點。 並找出所有恰具有 $2k$ 點的這種圖。

1.8. 試證明， 腰圍爲 5 的 k-正則圖至少有 k^2+1 點。 對於 $k=2$ 或 3， 具體找出一個恰具有 k^2+1 點的這種圖[11]。

1.9. 定義**奇圖**（odd graph）$\mathcal{O}_k$ 如下： 點集爲 $\{1,2,\ldots,2k+1\}$ 的所有 k-子集 （即恰含 k 個元素的子集）， 而兩點相鄰若且唯若它們不相交（disjoint）。 例如 $\mathcal{O}_2$ 就是 Petersen 圖（請檢查看看）。 試證明， 當 $k\geq 3$ 的時候 $\mathcal{O}_k$ 的腰圍是 6。

1.10. 令圖 G_n 的點集是所有 $\{1,2,\ldots,n\}$ 的排序。 兩個排序 $a_1a_2\ldots a_n$ 和 $b_1b_2\ldots b_n$ 相鄰若且唯若兩者可以在交換一對相鄰元素之後變成對方 （參見圖 1.19）。 證明 G_n 是連通的。

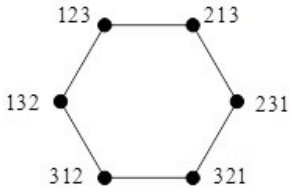


圖 1.19: 圖 G_3。

1.11. 令圖 G 的點集是所有長度爲 n 的二進位字串， 即 $a_1a_2\ldots a_n$、其中 $a_i\in\{0,1\}$。 而兩個字串相鄰的條件是它們恰有兩個對應的位置不同。 試求 G 的連通部分的個數。

1.12. 若圖 G 有 n 個點和 m 條邊使得 $m>\binom{n-1}{2}$， 證明 G 是連通的。 當 $n\geq 2$ 時， 請找出一個非連通圖使得 $m=\binom{n-1}{2}$。

1.13. 若圖 G 有 n 個點且其最小度數 $\delta(G)\geq(n-1)/2$， 證明 G 是連通的。 對所有的 n， 請找一個非連通圖滿足 $\lfloor(n-2)/2\rfloor$。

1.14. 設連通圖 G 爲 P_4-免除且 C_3-免除的， 試證明， G 是完全二分圖。

1.15. 設連通圖 G 爲 P_4-免除且 C_4-免除的， 試證明， G 當中有一個點連到其他所有的點。 （提示：考慮一個擁有最大度數的點）。

[11]Hoffman 和 Singleton 在 1960 年曾證明， 如果腰圍等於 5 且點數恰爲 k^2+1 的 k-正則圖存在， 則 $k=2,3,7$（這些都可以具體構造出來， 其中唯一一個滿足 $k=7$ 的情況的稱爲 **Hoffman-Singleton 圖**）、 或者 57 也有可能， 但至今尚未找到具體的例子。 詳情可參見第 **??** 節。

1.16. (a) 試證明， 圖 G 和其補圖 $\overline{G}$ 中至少有一個是連通的。

(b) 試證明， 若圖 G 是 P_4-免除且非 K_1， 則 G 和 $\overline{G}$ 中有一個是不連通的。

1.17. $K_{1,3}$ 又稱爲爪（claw）。 試證明， 設圖 G 的每個點之度數都是 3， 則 G 可以分解爲爪、 若且唯若 G 是二分圖。

1.18. 試證明， 連通圖 G 的任兩條最長路徑 P 和 Q 至少有一個公共點。

1.19. 試證明， 當 $k \geq 1$ 時， 在恰有 $2k$ 個奇點的連通近圖中， 存在 k 條不共用邊的行跡將此近圖的邊都用完。

1.20. 試證明， 每一個連通近圖都有一條道路， 將此近圖中的每一條邊至少用過一次。 甚至可以進一步要求， 每一條邊都用了一次或兩次。

1.21. 試證明， 每一個強連通有向近圖都有一條道路， 將此有向近圖中的每一條邊至少用過一次。 是否可以進一步要求， 每一條邊都用了一次或兩次？

1.22. 試證明， 對 $1 \leq k \leq 8$ 有向近圖 $G_{2,4}$ 都有一條長度爲 k 的有向圈。 是否對 $1 \leq k \leq \sigma^{n-1}$ 有向近圖 $G_{\sigma,n}$ 都有一條長度爲 k 的有向圈？

1.23. 試求出 $\sigma = 3$ 時的一種 de Bruijn 3-序列。

1.24. 試構造出兩個不同構但都是連通的圖， 使得它們的度序列相同。

1.25. 下面的序列何者爲圖序列？ 若是， 請構造出對應的圖； 若不是， 請說明理由。

$(5,5,4,3,2,2,2,1)$, $(5,5,5,3,2,2,1,1)$, $(5,5,4,4,2,2,1,1)$, $(5,5,5,4,2,1,1,1)$。

1.26. 令 $S = \{2,6,7\}$。 試證明， 存在一正整數 k、 使得一個 S 中每一元素恰用 k 次的序列是圖序列。 滿足這種條件的最小 k 爲何？

1.27. 試證明定理 1.15。 （提示：可以對點數作歸納法， 也可以對 $\sum_{i=2}^{n} d_i$ 做歸納法）

1.28. 試證明， 定理 1.16 的證法一中的 d' 確實滿足式 (1.1)。

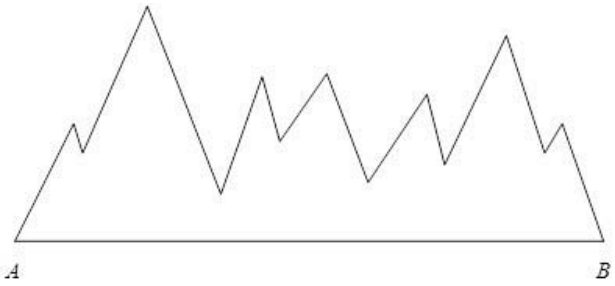
1.29. 對兩個非負整數序列 $a\colon a_1, a_2, \ldots, a_m$ 和 $b\colon b_1, b_2, \ldots, b_n$， 試證明， 存在一個二分重圖其兩部分的度序列分別是 a 和 b 的充分必要條件是， $\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j$。

1.30. 對兩個排序的非負整數序列 $a\colon a_1 \geq a_2 \geq \ldots \geq a_m$ 和 $b\colon b_1 \geq b_2 \geq \ldots \geq b_n$， 試證明， 存在一個二分圖其兩部分的度序列分別是 a 和 b 的充分必要條件是，

$\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j$ 且 $a_1^* + a_2^* + \ldots + a_k^* \geq b_1 + b_2 + \ldots + b_k$ 其中 a_i^* 是滿足 $a_j \geq i$ 的下標 j 的個數、 而當 $k > n$ 時令 $b_k = 0$。

1.31. 設 $d_1 \leq d_2 \leq \ldots \leq d_n$ 是圖 G 的度序列。 如果當 $i \leq n-1-d_n$ 時恆有 $d_i \geq i$， 試證明， G 是連通的。

1.32. 一個山脈是指坐標平面上從 $A = (a, 0)$ 到 $B = (b, 0)$ 在上半平面的一連串折線段。 考慮兩個登山者 A 和 B 分別從兩端點出發， 試證明， 他們有辦法用一種（或許是非常詭異的）方式登山、 使得兩個人在登山過程中的任何時刻都維持在同樣的海拔高度， 而且最後又能碰面。[12] （提示：請試著用一個圖來模擬兩個登山者的移動）



1.9. 參考文獻

[1] E. T. Bell, *Men of Mathematics*, New York: Simon & Schuster, 1986, c1736. 井竹均等譯，大數學家，九章出版社，1998。

[2] C. Berge, *Graphs and Hypergraphs*, Translated by E. Minieka, North-Holland, Amstendam, 1973.

[3] N. L. Biggs, E. K. Lloyd and R. J. Wilson, *Graph Theory 1736-1936*, Clarendon Press, Oxford, 1998.

[4] J. A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, American Elsevier, New York, 1976. （本書已由作者放在網路上供免費下載）

[12]這是由 Hoffman 提出的有趣結果。 可以注意的是， 在這種登山方式裡頭， 其中一個人可能經常必須在某一段路上來回數次。

[5] G. Chartrand and P. Zhang, *Introduction to Graph Theory*, McGraw-Hill, Boston, 2005.

[6] S. A. Choudum, A simple proof of the Erdős-Gallai theorem on graph sequence, *Bull. Austral. Math. Soc.*, vol. 33 (1986), pp. 67-70.

[7] R. Diestel, *Graph Theory, Second Edition*, Springer-Verlag, New York, 2000.

[8] P. Erdős and T. Gallai, Graphen mit Punkten vorgeschriebenen Graphs, *Mat. Lapok*, vol. 11 (1960), pp. 264-274.

[9] L. Euler, Soluto problematics ad geometriam situs pertinentis, *Commentaii Academiae Scientiarum Impericalis Petropolictanae*, vol. 8 (1736), pp. 128-140.

[10] R. Gould, *Graph Theory*, the Benjamin/Cummings, Menlo Park, CA, 1988.

[11] S. L. Hakimi, On the realizability of a set of integers as degrees of the vertices of a graph, *SIAM J. Appl. Math.*, vol. 10 (1962), pp. 496-506.

[12] F. Harary, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969.

[13] V. Havel, A remark on the existence of finite graphs (Czech.), *Časopis Pěst. Mat.*, vol. 80 (1955), pp. 477-480.

[14] D. Kőnig, *Theory of Finite and Infinite Graphs*, translated by R. Mcloart with commentary by W. T. Tutte, Birkhäuser, Boston, 1990. (Originally published as *Theorie der Endlichen und Unendlichen Graphen*, Akademische Verlagsgesellschaft Leipzig 1936. German Edition 1986.)

[15] E. Sperner, Neuer Beweis für die Invarianz der Dimensionszahl und des Gebietes, *Hamburger Abhand.*, vol. 6 (1928), pp. 265-272.

[16] A. Tripathi and H. Tyagi, A simple criterion on degree sequences of graphs, *Discrete Appl. Math.*, vol. 156 (2008), pp. 3515-3517.

[17] A. Tripathi, S. Venugopalan and D. B. West, A short constructive proof of the Erdős-Gallai characterization of graphic lists, *Discrete Math.*, vol. 310 (2010), pp. 843-844.

[18] W. T. Tutte and C. St. J. A. Nash-Williams, *Graph Theory*, Addison-Wesley, Menlo Park, CA, 1984.

[19] S. M. Ulam, *A Collection of Mathematical Problems*, Wiley, New York, 1960, pp. 29.

[20] D. B. West, *Introduction to Graph Theory, Second Edition*, Prentice Hall, Upper Saddle River, NJ, 2001.

[21] 王樹禾， 圖論及其算法， 中國科學技術大學出版社，合肥，1990。

第 2 章　演算法簡介

能對電腦解釋清楚的東西叫做科學，其他的東西叫做藝術。

– Donald Knuth（高德納）

2.1.　演算法起源

演算法是數學裏隨著電腦的興起被擴大研究的領域， 而演算法本身卻可以追溯到古代。早在古希臘時期， 爲了解決各種算術上的問題就已經有許多不同的演算法， 包括至今在數論上仍然重要的 Euclid 演算法、即輾轉相除法。 演算法在中國古代文獻中稱爲「術」， 最早出現在《周髀算經》及《九章算術》， 例如《九章算術》給出四則運算、最大公約數、 最小公倍數、 開平方根、 開立方根、 求質數的篩法、 線性方程組求解的演算法。 三國時代的劉徽給出求圓周率的演算法： 劉徽割圓術。 自唐代以來， 歷代更有許多專門論述「算法」的專著： 唐代的《一位算法》一卷、《算法》一卷， 宋代的《算法緒論》一卷、《算法秘訣》一卷、 楊輝的《楊輝算法》， 元代的《丁巨算法》， 明代程大位的《算法統宗》， 清代的《開平算法》、《算法一得》、《算法全書》。 而英文名稱 algorithm 來自於 9 世紀波斯數學家花拉子米 （比阿勒·霍瓦里松，拉丁轉寫 al-Khwarizmi）， 因爲比阿勒·霍瓦里松在數學上提出了演算法這個概念；al-Khwarizmi 後來音轉爲 algorism， 在 18 世紀演變成 algorithm。

簡單來說， **演算法**就是一個有限而明確定義的指令清單， 用途是要針對一個給定的輸入， 在有限的步驟內輸出解決特定問題的答案或特定工作的結果。 第 1 章有幾個跟圖論相關的演算法的例子， 例如，尋找 Euler 迴路的演算法、 尋找 de Bruijn 序列的演算法、 以及判斷圖序列的演算法等。 圖論是一門具有高度應用性的數學分支， 爲了能夠將理論拿來解決實際的問題， 必須設計出各種對應的演算法， 其難度絕不亞於理論本

身。 演算法除了要求正確之外， 更希望它能快速運作而且節省記憶體空間。 有很多圖論問題至今仍然沒有辦法找到充分有效率的演算法。

不僅圖論的理論可以用來產生演算法， 反過來演算法也經常被用來證明各種圖論的定理， Euler 定理的證明就是一個例子， 在往後的章節裡將會看到更多這樣的例子。 由於這是一本談論圖論以及相關演算法的書籍， 在更進一步介紹各種相關理論之前， 這一章要先介紹一些關於演算法的基本概念、 以及當我們要將演算法實際應用在圖論問題上時所需要注意的課題。

2.2. 演算法的複雜度

演算法主要關心的課題有兩個， 一個是問題的**演算法可解性**（computablility）， 另一個是演算法的**計算複雜度分析**（computational complexity analysis）。

第一個問題所探究的是， 給定一類型的問題， 是否存在一個演算法可以解決這個類型當中的所有問題？ 早期的數學家很快就意識到這並非永遠可能。 最基本的例子就是所謂的**停機問題**（halting problem）， 摘要而言， 這個問題在問， 存不存在一個演算法能夠判斷一個任意給定的程式是否總是會終止 （有些程式基於設計上的錯誤， 可能永遠不會停止）， 可以證明這樣的演算法是不可能存在的， 細節請參閱介紹演算法的書籍。 另外一個經典的例子是 1970 年的 Matiyasevich 定理， 他證明了沒有一個演算法可以判斷任意的 Diophantus 方程 （即數論中的整係數方程式求整數解）是否有解。 在這兩個例子中， 這些結論並不意味著這兩個問題無解， 只表示不存在可以解它們的演算法， 也就是說， 就算它們可解， 也會需要無限多的方法才能夠解決該類問題中的每一個問題； 而這是演算法辦不到的， 因爲根據定義， 一個演算法只能含有有限條指令。 通常，要證明一個問題是演算法可解， 辦法就是實際構造出一個演算法、 並且證明它能解決該問題（不過偶爾也有純粹只是存在性的證明）。

至於複雜度分析， 著重的在於一個演算法需要用多少時間和空間（記憶體用量）才能解決對應的問題。 演算法除了確實能夠解決問題之外， 還必須要在能夠被接受的時間及空間的限制下解決問題。 一般比較著重在時間複雜度的分析， 因爲空間複雜度的分析相對比較容易。 複雜度要從三個層次來討論， 一個是問題本身、 一個是演算法、 最後是演算法的**實作**（implementation）。

一個**問題**（problem）可能包含一個待回答的疑問， 一個待滿足的要求， 或是一個待求得的最佳可能情況或架構、 即問題的**解答**（solution）。 問題本身可能會含有一些尚未定值的**參數**（parameter）或**變數**（variable）， 藉由設定這些參數以得到問題的不同**範例**（instance）； 舉例來說， 假設要解決的問題是「判斷一個程式 P 會不會停止」， 設定其中的參數 P =「無窮迴圈程式」的時候， 就得到了問題的一個範例是「判斷無窮迴圈程式會不會停止」， 而這個範例的對應解答是「不會」。 對於一個給定的問題， 一個針對此問題的演算法就是指根據問題的範例來產生解答的逐步程序， 而演算法的實作則是指將演算法轉化爲可以實際讓電腦運作的程式。 不同的演算法和不同的實作都可能導致效率的差異。

圖論相關的演算法問題可以粗分成兩類： 一種稱爲**決定性問題**（decision problem）， 這種問題只要針對範例回答「是」或「否」即可； 另一種稱爲**最優化問題**（optimization problem）， 這種問題必須針對範例找出一個滿足條件的最好解答， 至於什麼是最好則是視問題的要求而定。 乍看之下， 最優化問題似乎比決定性問題難，但是其實只要後者會做、 前者就一定會做。 第 1 章曾經談過分解的概念， 底下就以這個來舉這兩種問題的例子：

完全圖分解問題（最優化）

參數： 一個給定的圖 G。

問題： 最少要用幾個完全圖， 才可以把 G 分解成這些完全圖？

完全圖分解問題（決定性）

參數： 一個給定的圖 G 和一個給定的正整數 k。

問題： 是否可以把 G 分解爲最多 k 個完全圖？

一個最優化版的完全圖分解問題的演算法， 顯然可以導出一個決定性版的完全圖分解問題的演算法。 反過來， 假設我們有一個決定性版的完全圖分解問題的演算法， 那麼對於最優化問題， 我們只要逐一代入 $k = 1, 2, 3, \ldots$ 去解決定性問題， 直到第一次出現「是」的時候， 當時的 k 值就是最優化問題的答案。 如果 G 有 m 條邊， 那麼顯然 G 可以分解成 m 個 K_2， 因此我們最多只要執行 m 次決定性版的演算法就能解決最優化問題。 利用二分法， 可以最多只要執行 $\log_2 m$ 次決定性版的演算法就能解決最優化

問題。 而如果我們有辦法平行地進行這 m 次的執行， 那麼解決最優化問題的時間就變得僅跟決定性問題一樣而已。

習慣上， 會用範例當中設定的參數之大小的函數來表示演算法的複雜度。 假設某一個問題 Π 的範例參數之大小 （也稱爲是此問題的**輸入大小**（input size））爲 n， 而且存在一個和 n 無關的常數 $c > 0$， 使得對應的演算法總是能夠在不超過 $cf(n)$ 的計算步驟之內得到解答（其中 f 是某一個函數）， 那麼就會說這個演算法能夠在 $O(f(n))$ 的時間內執行完畢； 而這個演算法的時間複雜度， 則是取滿足前敍述的最小函數 f （這邊所謂最小， 粗略而言， 是指成長的速度最慢）； 空間複雜度的表示法也是類似。 而問題本身的複雜度， 則是指所有能夠解決該問題的演算法當中複雜度最小的演算法的複雜度。

以第 1 章求一個圖的 Euler 迴路的問題爲例， 如果圖 G 有 n 個頂點及 m 條邊， 則此問題的輸入大小就是 $n + m$； 第 1 章給出的是一個需時爲 $O(n + m)$ 的演算法。 再以求 de Bruijn 序列的演算法來說， 輸入大小是 $1 + \lfloor \log_2 \sigma \rfloor + 1 + \lfloor \log_2 n \rfloor$， 因爲一般而言， 需要用 $1 + \lfloor \log_2 m \rfloor$ 的記憶體空間來儲存一個正整數 m； 第 1 章所提供的方法中， 光是圖 $G_{\sigma,n}$ 就有 σ^{n-1} 個點和 σ^n 條邊， 所以這個方法基本上是一個 $O(\sigma^n)$ 的演算法， 是一個「超指數型」時間的方法。

爲了讓讀者有大略的感覺， 圖 2.1 列出一個數字成長比較表。 從這個表中可以看出來， 只要 $n = 100$， 2^n 就是一個大得可怕的數字。 舉例來說， 如果電腦每秒鐘可以執行 10^{20} 個運算， 以一年約有 3×10^7 秒估計的話， 一個花費 2^{100} 步運算的演算法就得用上 300 年的時間， 這並不是一個人在有生之年可以看到答案的計算。 相對的， 如果演算法的時間是 $O(n^3)$， 對於輸入大小 $n = 100$ 的問題， 不到一秒鐘就能算出解答。

n	$n \log n$	n^2	n^3	2^n
10	2.3×10	10^2	10^3	10^3
10^2	4.6×10^2	10^4	10^6	10^{30}
10^3	6.9×10^3	10^6	10^9	10^{300}
10^4	9.2×10^4	10^8	10^{12}	10^{3000}
10^5	1.2×10^6	10^{10}	10^{15}	10^{30000}

圖 2.1: 數字成長比較表。

近年來， 研究學者致力的， 是針對各式各樣問題， 尋求更快、更有效率的演算法。 許多問題本來只有**指數時間演算法**（exponential-time algorithm）、 即複雜度爲 $O(a^n)$ 的演算法、 其中 $a > 1$ 爲常數， 後來改進成爲**多項式時間演算法**（polynomial-time algorithm）、 即複雜度爲 $O(n^k)$ 的演算法、 其中 $k \geq 1$ 爲常數， 最後可能還可以改進成爲**線性時間演算法**（linear-time algorithm）、 即複雜度僅爲 $O(n)$ 的演算法。 這三種程度的演算法的效率， 如圖 2.1 所顯示， 隨著輸入參數 n 的大小的成長而有極爲顯著的差異。

首先， 如果是指數時間演算法， 只要參數 n 的大小稍微增加， 需要的運算時間會立刻變得過份龐大， 一般認爲這樣的演算法是不切實際的。 相對地， 多項式時間演算法比較能夠把運算時間控制在一個合理的成長幅度之內， 一般我們會認爲這樣的演算法才具有實作的價值。 當然啦， 如果一個演算法需要的時間是 $O(n^{100})$， 就算這確實是多項式演算法， 我們也不會覺得這是一個堪用的演算法。 一個演算法究竟有沒有實用價值不能只看複雜度的層級， 也必須考慮在目前的電腦能力所及的範圍內是誰比較快的問題。 有些時候， 指數演算法反而會比多項式演算法管用， 因爲雖然當 n 比較大的時候它們總是遠遠不如多項式演算法， 但是在 n 比較小的時候卻可能比較快。 最後， 如果問題存在線性時間演算法那當然最好， 但是這種情況並不是常常有的。

一個貼近日常生活的例子是數字的**排序**（sorting）， 也就是給定一實數列 $x_1, x_2, \ldots, x_n$， 要將它從大到小重排爲 $x_{[1]} \geq x_{[2]} \geq \ldots \geq x_{[n]}$。 一個簡單的方法是， 先從這 n 個數中取出最大的當做 $x_{[1]}$， 接著由剩下的 $n-1$ 個數中取出最大的當做 $x_{[2]}$， 依此類推， 就能順利排序， 其所用的時間是 $n+(n-1)+\ldots+1 = n(n+1)/2$， 即爲 $O(n^2)$。 其實有比較複雜、 但是更有效率的方法叫做**堆排序**（heap sort）， 所花的時間爲 $O(n \log n)$。 當 $n = 10^5$ 時， 這兩種方法相差上千倍， 或著至少幾百倍。 想想看， 臺灣每年十萬高中學生入大學的考試， 考完之後， 就要先將所有學生的成績排序， 才能分發志願。 如果用到好的方法， 排序的工作可能是一天， 如果用 $O(n^2)$ 的方法， 光是排序可能就要一年， 那就是說， 七月大考完了之後， 要等上一年才能分發， 如果是這樣的話， 最好的方法就是考完之後大家先去當一年兵， 退伍之後就知道考上那間大學了。 當然， 臺灣的七月大考從來沒發生過這個現象， 依靠的就是演算法的技術不差。

再舉一個例子來說明， 對一個問題精益求精、 尋找更有效率演算法的過程。 給定

實數列 $x_1, x_2, \ldots, x_n$， 考慮從這個數列中找出連續的一段 $x_i, x_{i+1}, \ldots, x_j$、 使其和爲最大的問題。 一個從題目敍述直接推導出來的演算法是， 對所有滿足 $1 \le i \le j \le n$ 的 i 和 j， 求 $s_{i,j} = x_i + x_{i+1} + \ldots + x_j$、 然後再從所有的 $s_{i,j}$ 裡面去找出最大的一個。 由於所有的這樣的 i, j 共有 $O(n^2)$ 種選擇、 而每次計算 $s_{i,j}$ 要做 $O(n)$ 次加法， 因此用這種演算法需用 $O(n^3)$ 個步驟， 這是最直觀的想法所導致的結果。 但稍微改進的話， 可以用如下的方法： 對於一個固定的 i， 首先 $s_{i,i} = x_i$， 然後當要求 $s_{i,j}$ 的時候， 改用 $s_{i,j} = s_{i,j-1} + x_j$ 這個式子來做遞迴計算， 如此一來總共只需要 $O(n)$ 次加法就可以把 $s_{i,i}, s_{i,i+1}, \ldots, s_{i,n}$ 都算出來， 於是就把總共的需時改進成爲 $O(n^2)$。 更進一步， 是否還有更快的方法呢？ 請參見習題 2.5。

當一個演算法被構造出來時， 就得到了對應問題的複雜度的一個上界。 而透過一些數學論證， 能給出問題複雜度的下界， 也就是去證明不管用什麼方法、 都至少需要一定的步驟才有可能解決該問題。 假如得到的上界和下界一致， 那就表示找到的已經是最佳的演算法， 而且問題的複雜度也跟著確定。 但是這樣的例子並不多， 對於大部分的問題而言， 能求得的上界和下界都是相差很遠的， 所以一般僅能知道問題複雜度落在那一個範圍而已。

在圖論領域中， 有許多問題至今都還沒有一個充分有效率的演算法， 因此會想知道究竟只是那樣的演算法還沒被找到、 或者說不定其實根本不存在。 這會牽涉到所謂 P 和 NP 的概念。 關於這個部分， 之後在第 15 章會有更完整的探討， 不過這裡先給一個簡單的介紹。 基本上， 給定一個決定性問題， 如果這個問題的答案可以在多項式時間內被確定出來， 那就稱爲 P 問題。 NP 的完整定義比較複雜， 概括而言， 針對一個給定的決定性問題， 假如當問題的答案爲「是」的時候我們有辦法在多項式時間內驗證答案的確爲「是」， 那就稱爲 NP 問題[1]。 容易看出 P 問題都是 NP 問題（即 $\mathrm{P} \subseteq \mathrm{NP}$ ）， 此外， 因爲答案一般來說都只有有限種可能， 因此只要有辦法弄到充分多台的電腦同時檢查那所有可能的答案， 就有辦法在多項式時間內解出一個 NP 問題 （這是另外

[1]但是， 如果問題的答案爲「否」， 那就不見得有辦法一樣輕易地驗證答案的正確性。 如果一個決定性問題當答案爲「否」的時候可以在多項式時間內被驗證， 就稱爲是一個**餘-NP**（co-NP）問題。 有些問題同時屬於 NP 和餘-NP， 例如整數分解問題： 判斷一個給定的整數 m 是否含有不超過 n 的眞因數（這個問題屬於 NP 是容易的， 屬於餘-NP 則困難許多）。

P 和 NP 分別是指「polynomial time」和「non-deterministic polynomial time」， 即「多項式時間」和「不定型多項式時間」之意， 後者的由來是因爲 NP 最初的定義是指可以在不定型 Turing 機上以多項式時間解決的問題。 詳情請見第 15 章。

一種 NP 的簡單說法）。 但是好奇的是， 有沒有可能只用一台電腦就在多項式時間內解出任何的 NP 問題？ 這就是演算法中至今尚未能解答的「P = NP 是否成立？」這個有名的問題[2]。

要如何證明「任何的 NP 問題」都可以在多項式時間內解決呢？ 這需要用到 **NP-困難**（NP-hard）以及 **NP-完全**（NP-complete）的概念。 假設有一個問題具有這樣的特性：「只要它存在多項式演算法、 那麼任何 NP 問題都可以藉助該演算法在多項式時間內解決」， 那麼就說這是一個 NP-困難問題。 如果一個問題同時是 NP-困難問題、也是 NP 問題， 那就說它是一個 NP-完全問題。

NP-完全的理論是由 Cook [2] 在 1971 年建立， 他透過舉出實例證明了 NP-完全問題存在。 在那之後， Karp [7] 也陸續提出了許多其他 NP-完全問題的例子。 而以圖論的例子來說， 可以證明， 如果隨便給予兩個圖、 要判斷是否其中一個是另一個的子圖，那麼這會是一個 NP-完全問題 （參見習題 15.**??**）。 在演算法和圖論領域中都存在有許許多多的 NP-完全問題， 只要有人能夠證明那當中任何一個可以在多項式時間內解決，那麼就證明了 P = NP， 且必定會帶來重大的震撼衝擊。

只是， 到目前爲止沒有人能做到這一點， 因此現在大部分的人都相信應該是 P ≠ NP 才對、 雖然這也沒人能證明。 換句話說， 假設已經知道某個問題是 NP-完全問題，那麼大致是不用指望它能存在一個充分有效率的演算法了。

等到在這本書沿途當中累積了足夠多的圖論基礎之後， 第 15 章會將介紹圖論中的 NP-完全問題的例子， 屆時，也會利用類似 Karp 的化簡手法去證明它們的 NP-完全性。

---

[2]千禧年大獎難題（Millennium Prize Problems）， 是七個由美國克雷數學研究所（Clay Mathematics Institute, CMI）於 2000 年 5 月 24 日公布的數學難題，「P = NP 是否成立？」是其中第一個問題。 根據克雷數學研究所訂定的規則， 所有難題的解答必須發表在數學期刊上， 並經過各方驗證，只要通過兩年驗證期， 每解破一題的解答者， 會頒發獎金 100 萬美元。 七個難題中的第三個問題、 龐加萊猜想（Poincaré conjecture） 2006 年由俄羅斯數學家格里戈里·佩雷爾曼（Grigori Perelman、俄語 Григорий Яковлевич Перельман）完成最終證明， 他也因此在同年獲得菲爾茲獎 （Fields Medal），但並未現身領獎， 2010 年 CMI 頒給他百萬獎金， 他也拒絕領獎。

這些難題是呼應 1900 年德國數學家 David Hilbert 在巴黎提出的 23 個歷史性數學難題， 經過一百年， 許多難題已獲得解答。 而千禧年大獎難題的破解， 極有可能爲密碼學以及航天、 通訊等領域帶來突破性進展。

2.3. 資料結構

演算法的目標是要讓電腦處理圖論的問題， 而電腦是被設計來處理數字的， 可是現在要處理的物件是頂點和邊， 爲了能夠讓設計出來的演算法被電腦執行， 需要用一個好的方式來組織那些會用到的資料。 在演算法學中， 探討如何將程式中的資料整理得有系統、使得程式運作起來更有效率的學問就叫做**資料結構**（data structure）。

人們最熟知的資料結構是**陣列**（array）， 在數學中就相當於有下標的變數。 0-維陣列就是單一的變數， 也就是電腦中的一個記憶位置； 遞迴來說， d-維陣列就是一個有限序列的、 大小相同的 $(d-1)$-維陣列。 於是， 一般我們以 1-維陣列表示數學中的向量、 而以 2-維陣列表示矩陣， 依此類推。

陣列最大的意義在於可以利用下標來存取許多個不同的記憶體位置。 一般來說，陣列中的每個元素都佔有同樣大小的空間， 例如都是 s 位元。 因此， 對於 1-維陣列 $A_1, A_2, \ldots, A_{m_1}$， 如果 A_1 的記憶體位置爲 B， 那麼 A_i 的位置就是 $B+(i-1)s$。 而高維度的陣列可以視爲是許多一維陣列的串連， 例如一個 $m_1 \times m_2$ 矩陣可以說成是一個 1-維陣列 （依照列的次序排列）

$$A_{1,1}, A_{1,2}, \ldots, A_{1,m_2}, A_{2,1}, A_{2,2}, \ldots, A_{2,m_2}, \ldots, A_{m_1,1}, A_{m_1,2}, \ldots, A_{m_1,m_2}，$$

因此如果 $A_{1,1}$ 的位置是在 B， 那 $A_{i,j}$ 的位置就會是在 $B+(i-1)m_2s+(j-1)s$。 類似地也可以考慮以行的次序優先的排法、 以及更高維度的陣列。

前一節中 n 個實數排序的例子， 假設這 n 個實數存在 1-維陣列 X 中， 前述的方法寫爲如下的程式。 因爲程式有兩個迴圈， 容易看出來時間複雜度是 $O(n^2)$。

```
for i = 1 to n−1 do
{    t ← i ;
     for j = i+1 to n do
          if (X[t] < X[j]) then t ← j ;
     m ← X[t] ;
     X[t] ← X[i] ;
     X[i] ← m ;
}
```

另一個從給定的實數列中找出和最大的連續一段， 由定義求答案的程式如下。

```
M ← X[1] ;
for i = 1 to n do
for j = i to n do
{     s ← 0 ;
      for k = i to j do s ← s+X[k] ;
      if (M < s) then M ← s ;
}
```

因爲程式有三個迴圈， 時間複雜度是 $O(n^3)$。 改進到 $O(n^2)$ 的程式如下。

```
M ← X[1] ;
for i = 1 to n do
{     s ← 0 ;
      for j = i to n do
      {     s ← s+X[j] ;
            if (M < s) then M ← s ;
      }
}
```

以陣列爲基礎， 可以演化出其他各種不同的資料結構。 仔細說起來的話， 這些衍生的資料結構在本質上也是以陣列方式存放， 只是它們額外附加了一些資料以便協助演算法進行特定操作， 這使得它們在概念上具備了不同於陣列的型態。

堆疊（stack）和**佇列**（queue）是最簡單的兩種衍生資料結構， 它們最常見的用途是用來做工作的排程。 假定現在櫃臺陸陸續續來了一些人要辦理業務， 我們應該要以什麼順序來接待這些人呢？ 現實中約定俗成的辦法就是叫他們排隊、 然後先到的先辦， 這樣的安排方式基本上就是佇列； 每次有新的工作進來的時候就排在最後面， 而每次都是從最前面的資料開始處理， 處理完了就移除。 現實中似乎就只有這種作法是合理的， 但是在演算法中， 另外一種逆其道而行的作法也一樣重要； 堆疊採取的是後到先處理的作法， 每次有新的工作進來時也是排在最後面， 但是每次卻都是從最後面的資料先處理。

要實現堆疊和佇列的基本辦法就是用 1-維陣列。 堆疊就照字面上的意思， 每次新的資料進來就附加在陣列的末端、 然後每次都從末端先取出資料（並從陣列中移除）即可。 佇列同樣是以陣列的形式存在於記憶體當中， 每次新的資料也是附加在陣列的末端， 取出的時候則是從陣列最前面開始取走。

在堆疊當中， 通常會把資料的末端稱爲**頂端**（top）， 並且用一個變數「TOP」來記錄頂端的位置， 而將資料存在一個（例如叫 DATA 的）陣列當中。 於是， 每次新的資料就是放在 DATA[TOP+1] 的位置， 並且將 TOP 更新爲 TOP+1； 而移出資料時則是取走 DATA[TOP] 的資料， 並將 TOP 更新爲 TOP−1。 在這個操作時，不用眞的把移出的資料從記憶體中清除沒關係， 畢竟下次有新的資料進來時它就會自動被覆蓋掉。 圖 2.2 展示了堆疊資料型態的結構。

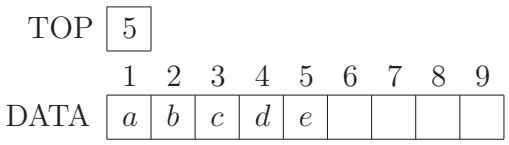


圖 2.2: 一個堆疊的例子。

而在佇列當中， 除了要知道**尾端**（rear）在那裡之外， 也需要知道**開頭**（front）在那裡， 因此會用兩個變數「REAR」和「FRONT」來儲存。 此時資料的分佈就是從開頭位置到尾端位置所標示的範圍內。 每次有新的資料進來就是放在 DATA[REAR+1] 的位置， 並將 REAR 更新爲 REAR+1， 而移出資料時則是取走 DATA[FRONT]的資料， 並將 FRONT 更新爲 FRONT+1。 圖 2.3 展示佇列資料型態的結構。

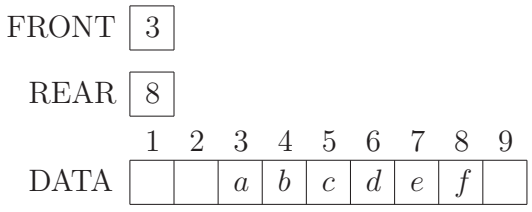


圖 2.3: 一個佇列的例子。

不過， 這樣的作法有一個小問題是， 整個佇列在記憶體空間中會不斷地往後移動，

當 FRONT 和 REAR 一直增加時， DATA 預留的空間慢慢會被用完， 而 FRONT 前面的空間則閒置沒用。 如果可以事先知道這個佇列最多只會有幾項， 例如至多 n 項， 那麼就可以宣告一個大小爲 n 的 1-維陣列、 然後循環地使用這個記憶體空間； 也就是說， 當 FRONT（或是 REAR）是 n 時 FRONT+1（或是 REAR+1）就設爲 1。 因爲已經事先預估好資料最多只會有 n 項， 因此這樣的作法不會導致資料相撞。

堆疊不像佇列那樣， 可以自然地用來處理排隊辦理業務， 但是它在電腦科學的各個地方都很有用。 下面以電腦的編譯器（compiler）處理遞迴程式爲例說明。 遞迴是數學上方便又省力的一種思考方式， 例如，在定義 $n!$ 時， 可以說 $n! = 1 \times 2 \times \ldots \times n$， 也可以採用如下的遞迴定義： $0! = 1$； 當 $n > 0$ 時， $n! = (n-1)!n$。 寫成程式就是

F(n):
  if $(n = 0)$ **then return** 1 ;
  else return F(n−1) $*$ n ;

這可以用一個更簡單的程式完成：

ANSWER = 1 ;
for i = 1 **to** n **do** ANSWER = ANSWER $*$ i ;

並不是所有的遞迴程式都能那麼簡單的轉換爲非遞迴程式。 以著名的**河內塔**（Hanoi's towers）問題爲例， 傳說河內有一座廟， 廟裡有三根神柱， 姑且編號爲 1、2、3， 最初第 1 根柱子上有編號 1 到 64 的圓盤， 號碼小的在上、大的在下。 神要和尙把圓盤從第 1 根柱子搬到第 3 根柱子， 每次搬動時是將某根柱子最上方的圓盤搬到另一根柱子、 並且放在新柱子最上方， 規定是， 任何時候不能把大號的圓盤放在小號圓盤的上方。 據說， 當所有圓盤都被搬到第 3 根柱子後， 世界末日就到來了。 河內塔問題可以用下面的遞迴程式、 執行 Hanoi(1, 2, 3, 64) 完成。

Hanoi(A, B, C, n):
  if $(n > 0)$
  then Hanoi(A, C, B, n−1);
    print "move disk n from tower A to tower C" ;
    Hanoi(B, A, C, n−1) ;

要把 Hanoi 改寫成一個不是遞迴的程式並不是一件容易的事。 一般來說， 電腦的編譯器是利用堆疊來執行遞迴程式， 以 F(n) 爲例， 用比較簡便的方式說明如下。

編譯器會留一段空間儲存所有變數的值， 可以將它看成一個堆疊 DATA、頂端爲 TOP。 假定程式在某個地方要執行「A = F(2)」， 也就是要算 2!、 然後把答案放入 A 中。 如果在執行 A = F(2) 之前 TOP = 3， 而 DATA[3] 是存 A 的值的位置， 一開始 DATA 的情況如圖 2.4 第 1 行所示。 爲了清楚， 圖中被圈住的位置是 TOP 的位置， $*$ 表示隨意數， 目前不重要。 要計算 F(2) 時， 進入程式 F(n) 開頭， 編譯器會留 3 位置給 F 的變數： F(n) 的值、目前還未知， n 的值、目前爲 2， 舊的 TOP、目前爲 3； 新的 TOP 則變爲 6， 如圖 2.4 第 2 行所示。 當 F 執行到第 2 行要算 F(1) 時， 遞迴程式 F 不能把原來變數的資料蓋掉， 因此編譯器會往後面找 3 個位置， 當做處理 $n = 1$ 時用， 新的 TOP 變爲 9， 如圖 2.4 第 3 行所示。 F(0) 的處理亦同， 新的 TOP 變爲 12， 如圖 2.4 第 4 行所示。 算完 F(0) 後， DADA[10] 變爲 1， 如圖 2.4 第 5 行所示。 算完 F(0) 後， 程式返回 F(1)； 算完 F(1) 後， 程式返回 F(2)； 算完 F(0) 後， 程式返回 A = F(2)； TOP 值也一路換回舊的 TOP 值， 如圖 2.4 第 6、7、8 行所示。

預留給 F(n) 的空間：

F(n)	n	舊 TOP

DATA	1	2	A 3	4	5	6	7	8	9	10	11	12	13	14	15
執行 A = F(2) 前 TOP = 3	$*$	$*$	(*)												
$n = 2$ 進入 F 時 TOP = 6	$*$	$*$	$*$	$*$	2	(3)									
$n = 1$ 進入 F 時 TOP = 9	$*$	$*$	$*$	$*$	2	3	$*$	1	(6)						
$n = 0$ 進入 F 時 TOP = 12	$*$	$*$	$*$	$*$	2	3	$*$	1	6	$*$	0	(9)			
$n = 0$ 結束 F 前 TOP = 12	$*$	$*$	$*$	$*$	2	3	$*$	1	6	1	0	(9)			
$n = 1$ 結束 F 前 TOP = 9	$*$	$*$	$*$	$*$	2	3	1	1	(6)	1	0	9			
$n = 2$ 結束 F 前 TOP = 6	$*$	$*$	$*$	2	2	(3)	1	1	6	1	0	9			
執行 A = F(2) 後 TOP = 3	$*$	$*$	(2)	2	2	3	1	1	6	1	0	9			

圖 2.4: 被圈住的位置是 TOP 的位置，$*$ 表示隨意數，目前不重要。

2.4. 表列與圖的表示法

前一節介紹的堆疊和佇列， 都只是陣列的稍微變形。 這一節要看一個在圖論演算法中更有用的概念。

表列（list）是由一些大小相同的**記錄**（record）線性連接而成， 其中每個記錄含有一個或多個**欄位**（field）， 有些存的是**資料**（data）、 有些則是存**指標**（pointer）。 圖 2.5 所示的是兩個**單連表列**（singly linked list）。

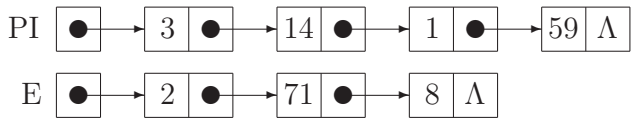


圖 2.5: 單連表列 PI 和 E。

表列和陣列最大的不同在於， 陣列的資料是依序存在電腦的記憶體中， 但表列的資料可以到處放置、 並以指標將它們串連。 舉例來說， 上面的兩個單連表列中， 我們可以利用圖 2.6 的陣列來存放它們， 其中 Λ 表示未定義的符號。

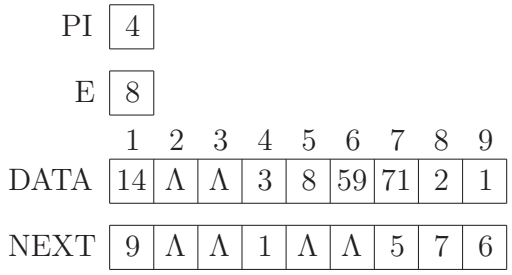


圖 2.6: 用陣列表達表列 PI 和 E。

這個例子的解讀這樣的。 首先有兩個變數 PI 和 E， 裡面的值分別代表對應表列在記憶體中、陣列 DATA 的第一個位置在那裡。 例如 PI 的值是 4， 就表示 PI 這個表列的開頭放在 DATA 的第 4 個位置， 也就是 3。 如果想要存取下一個元素， 就去看對應的位置在 NEXT 陣列中是多少 （這個陣列就是代表指標）， 此例中爲 1， 就表示下一

個資料被存放在 DATA 的第 1 個位置， 其值爲 14。 如此就可以依序讀出 PI 的內容爲 3、14、1、59， 直到讀到 NEXT 爲 Λ 的時候， 就知道 PI 表列到此結束了。 單連表列 E 也可以類似地讀出來。

任何資料結構都有好有壞， 要看需求來決定要用那一種資料結構。 比方說， 表列有一個明顯的缺點在於讀取資料的過程比較複雜而沒有效率， 例如若要讀取 PI 的第 3 個元素， 那就無法直接知道它被放在那裡， 必須從第一個元素開始逐一地找到第 3 個才知道； 相對地， 因爲陣列是依序排放， 那一個元素放那裡只要計算一下就知道， 所以要找資料就比較容易。 但是， 表列相較於陣列的好處在於它可以很容易地插入資料。 以 PI 來說， 假設想要在 14 跟 1 中間插入一個新的資料爲「5」， 那首先找一個地方存放「5」這個資料 （以上面的例子來說， DATA 的第二個位置是空的， 所以可以選來用）， 然後將「14」指向它、 並由「5」繼承「14」原本的指標， 就完成插入的動作了。 在這個操作過程中只更動了三個記憶體位置的資料， 如圖 2.7 中圈起來的位置所示。 然而， 假如在陣列中要做同樣的事情， 想像一個有上千個元素的陣列， 若要在第 1 個位置插入資料， 就必須將陣列所有的元素全部往後搬 s 位元， 才有辦法進行插入。

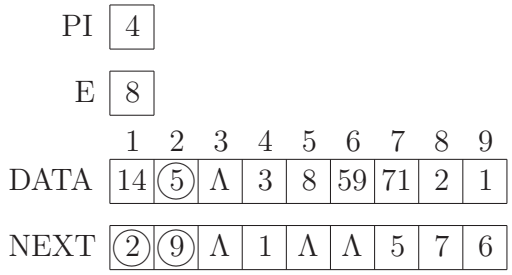


圖 2.7: 一個插入的動作，只需更動三個（圈起來的）記憶體中的資料。

下面這段程式將一個新的資料 x 插入單連表列的某一項 i 的後面， 其中 avl 是 DATA 中一個空而未用的地方。。

insert(DATA, NEXT, x, i, avl)；
    DATA[avl] ← x；
    NEXT[avl] ← NEXT[i]；
    NEXT[i] ← avl；

如果希望表列這樣的好處也可以發揮在刪除資料的功能上， 那比較好的作法是再多設一個指標 PREV， 用來表示每一個記錄的「前一個」是在何處。 這樣一來， 如果要刪除一個記錄， 只要將它對應的 PREV 和 NEXT 連接、 並將自己清除即可。 這樣的表列稱爲**雙連表列**（doubly linked list）； 圖 2.8 和圖 2.9 是分別將圖 2.5 和圖 2.6 改爲雙連表列後的樣子。 有時候， 也可以視需要，使用更複雜的結構。

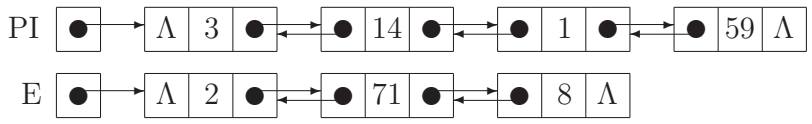


圖 2.8: 雙連表列 PI 和 E。

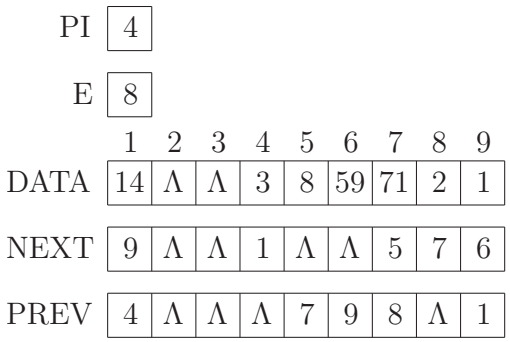


圖 2.9: 雙連表列 PI 和 E 的陣列表達。

有了陣列和表列的概念之後， 接著來要來看怎麼用資料結構表示一個圖。

假設 $G = (V, E)$ 是一個圖， 有 n 個頂點 $1, 2, \ldots, n$。 我們定義 G 的**相鄰矩陣**（adjacent matrix）$A = [a_{ij}]$ 是一個 $n \times n$ 矩陣， 其中

$$a_{ij} = \begin{cases} 1, & \text{若 } ij \in E\text{；} \\ 0, & \text{若 } ij \notin E\text{。} \end{cases}$$

由定義可知， A 是一個主對角線各元素均爲 0 的對稱 (0,1)-矩陣。 利用這個矩陣對應的陣列結構， 可以將一個圖的所有資訊儲存起來。 如果要表示一個重圖， 可以用 a_{ij} 表示點 i 和點 j 之間的重邊個數。 在近圖的情況， 如果點 i 上面有 s 條迴邊， 則將 a_{ii} 設

爲 $2s$。 最後， 在有向圖的情況， a_{ij} 只代表從點 i 連向點 j 的邊， 所以 a_{ij} 可能不等於 a_{ji}。

這樣的表達方法有好有壞。 好處是， 可以很簡單地判斷兩點是否相鄰、 且新增邊或刪除邊的操作也都很容易做。 不過， 如果要問點 i 的全體鄰居有那些點， 那就要將其他各點都看過一遍， 得花 $O(n)$ 的時間， 而如果要看這個圖總共有多少邊， 那就得用 $O(n^2)$ 的時間。 當 $\Delta(G)$ 遠比 n 小、 或者邊數遠比 n^2 小的時候， 這種表示法很沒效率。 而相鄰矩陣需要的儲存空間是 $O(n^2)$， 這一般來說當 n 比較大的時候也是大了點。

當演算法需要常常檢驗某一頂點的所有鄰居時， 最好選擇以**相鄰表列**（adjacent list）來表示圖。 相鄰表列是針對每一個頂點建立一個表列， 裡面存放的是該點各個鄰居的編號。 圖 2.10 展示了這兩種不同的表示法。

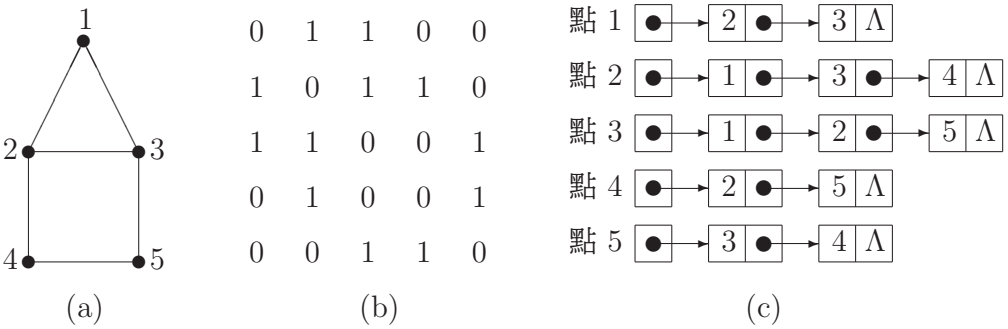


圖 2.10: (a) 有 5 點 6 邊的圖 G。(b) 圖 G 的相鄰矩陣。(c) 圖 G 的相鄰表列。

利用相鄰表列的話， 設 G 的邊數爲 m， 則需要的儲存空間會是 $O(n+m)$。 因此就空間的考量上相鄰表列的表現較好。 而如果要將一個點 v 的鄰居都選出、 或者將所有的邊都選出， 分別也只需要 $O(\deg(v))$ 和 $O(m)$ 的時間。 不過， 相對地它要判斷兩點是否相鄰就需要花上 $O(\Delta(G))$ 的時間， 刪除邊也不像用相鄰矩陣那麼快速。 這些效能上的差異， 都是在實作的時候必須審愼考慮的。

2.5. Euler 迴路的案例

這一節將用第 1 章找 Euler 迴路的方法爲例子， 詳細寫出圖的表示法、 以及程式。 在往後的章節中， 談論演算法的時候則以摘要式的程式爲主。 這裡是用相鄰表列表示一個

圖。 但是爲了表示圖的某一邊是否已經被走過， 額外需要用一個陣列「MARK」來註記： MARK = 0 表示該邊尚未被走過， 而爲 1 則表示該邊已經被走過。 但是， 一條邊 ij 出現兩次， 一次在點 i 的相鄰表列中， 另一次在點 j 的相鄰表列中； 當 Euler 迴路由點 i 經由此邊走到點 j 的時候， 自然需要把點 i 的相鄰表列中的 ij 邊註記、 但同時也需要把點 j 的相鄰表列中的 ji 邊註記。 爲達到這個目的， 需要在 ij 和 ji 之間設置雙連指標， 以便很容易就能找到對方。 爲此， 再多設一個指標「SAME」。

圖 2.11 是一個有 4 個點與 9 條邊的重圖， 演算法輸入的資料爲「$V = 4$」以及其 9 條邊：「1 2, 1 2, 1 2, 1 4, 2 3, 2 3, 2 4, 3 4, 3 4」， 在這些邊後面，會以 0 0 表示輸入完畢， 並利用如下程式 input-graph 得到相鄰陣列的內容：

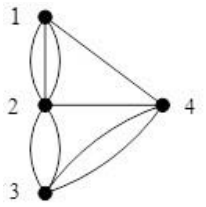


圖 2.11: 有 4 個點與 9 條邊的重圖 G。

```
input-graph :
    input V ;
    for i = 1 to V do ADJ[i] ← 0 ;
    NEW ← 1 ;
    input i, j ;
    while ( i ≠ 0 )
    {   DATA[NEW] ← j ;          DATA[NEW+1] ← i ;
        NEXT[NEW] ← ADJ[i] ;     NEXT[NEW+1] ← ADJ[j] ;
        ADJ[i] ← NEW ;           ADJ[j] ← NEW+1 ;
        SAME[NEW] ← NEW+1 ;      SAME[NEW+1] ← NEW ;
        MARK[NEW] ← 0 ;          MARK[NEW+1] ← 0 ;
        NEW ← NEW+2 ;            input i, j ;
    }
```

透過程式 input-graph 輸入圖 G 的資料， 會得到其相鄰表列的結構如圖 2.12 所示， 爲了畫圖的方便起見，指標 SAME 只畫出 1 2, 1 2, 1 2, 3 4, 3 4 這五條邊。 圖 2.13 是其陣列表示。

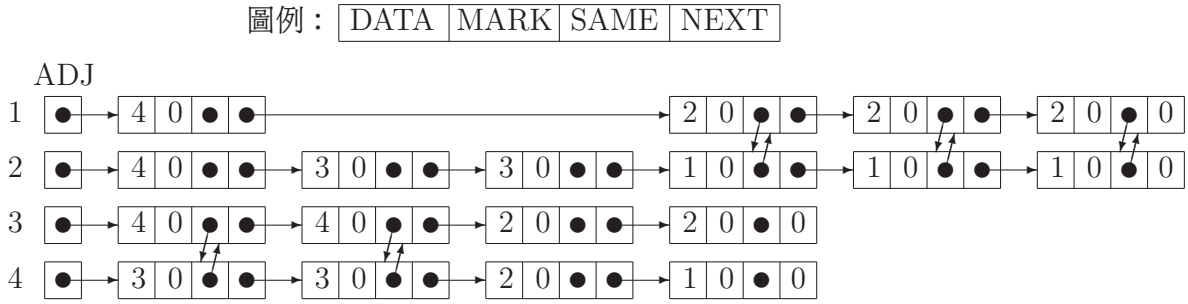


圖 2.12: 圖 G 的相鄰表列，方便起見指標 SAME 只畫出 1 2, 1 2, 1 2, 3 4, 3 4 這五條邊。

ADJ	7	13	17	18																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
DATA	2	1	2	1	2	1	4	1	3	2	3	2	4	2	4	3	4	3		
NEXT	0	0	1	2	3	4	5	0	6	0	9	10	11	8	12	14	15	16		
SAME	2	1	4	3	6	5	8	7	10	9	12	11	14	13	16	15	18	17		
MARK	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		

圖 2.13: 圖 G 的相鄰表列之陣列表示。

在上述的表示法中， 對於重邊我們並未分別命名， 只用 1 2, 1 2, 1 2 表示三條重邊， 事實上， 因爲它們的資料分別存在不同的位置， 就已經達到分別命名的功能。 另外， 雖然這裡所舉的例子沒有迴邊， 但是若圖中有迴邊 ii 也是可以表示清楚的， 只是這時候， 點 i 的相鄰表列中會出現兩次 i。

以下的程式 Euler-tour 是將第 1 章找一個圖的一條 Euler 迴路的方法、 也就是「能走就走、不能走則回頭」的方法， 仔細寫出來的的結果， 它將 Euler 迴路存在一個雙連表列 TOUR 中。

```
Euler-tour (V, ADJ, DATA, NEXT, SAME, MARK, TOUR, TOUR_N, TOUR_P) :
    for i = 1 to V do A[i] ← ADJ[i] ;
    TOUR[1] ← 1 ;   TOUR_N[1] ← 0 ;   TOUR_P[1] ← 0 ;
    CUR ← 1 ;  NEW ← 2 ;
    while (CUR ≠ 0)
    {       i ← TOUR[CUR] ;
            if (A[i] ≠ 0) then
                    TOUR[NEW] ← DATA[A[i]] ;
                    t ← TOUR_N[CUR] ;   TOUR_N[CUR] ← NEW ;
                    TOUR_P[NEW] ← CUR ;   TOUR_N[NEW] ← t ;
                    if (t ≠ 0) then TOUR_P[t] ← NEW ;
                    MARK[A[i]] ← 1 ;   MARK[SAME[A[i]]] ← 1 ;
                    j ← NEXT[A[i]] ;
                    while (j ≠ 0 and MARK[j] = 1) j ← NEXT[j] ;
                    A[i] ← j ;   CUR ← NEW ;   NEW ← NEW+1 ;
            else CUR ← TOUR_P[CUR] ;
    }
```

2.6. 聯集尋找問題

利用指標除了可以達到前面所述的好處之外， 還有更多的應用。 底下來看其中一個經典的用法， 是演算法學當中的「**聯集尋找**（union-find）」問題， 這在演算法的設計中有許多用途， 本書主要是在第 3.6 節求最小生成樹時可以用。

假設有一個宇集合 $U = \{1, 2, \ldots, n\}$、 這個集合在我們處理演算法的過程中被分割成一些互斥的非空子集合； 一般來說， 初始情況的分割一般是 $\{1\}, \{2\}, \ldots, \{n\}$， 此時以 i 來當作集合 $\{i\}$ 的名稱。 在演算過程， 經常需要做兩件事： 將兩個既有的集合聯集而形成一個新的集合、 以及判斷一個特定的元素在那一個集合裡面。 這樣的一種處理機制在很多演算法當中都看得到。

一種單純的作法是， 建立一個長度爲 n 的陣列 NAME， 其中 NAME[i] 的值就表

示 i 這個元素所屬的集合的名稱。 這麼一來， 要判斷一個元素 i 在那個集合， 只要看 NAME[i] 就立即可以完成， 但是若要將集合 a 和集合 b 做聯集並合併並以 a 命名， 就必須一一看過 NAME[1], NAME[2], ..., NAME[n]、 如果發現值爲 b 的就換成 a。 於是， 每做一次聯集的動作就需要 n 個步驟， 這麼一來， 假如在演算法的過程當中總共需要做 $O(n)$ 次聯集的動作， 就會需要 $O(n^2)$ 的時間， 這樣的時間略嫌長一點。

一種稍微改進的方法是， 除了建立上述的陣列 NAME 以外， 對每個集合我們再用一個單連表列來表示其所有的元素。 這個時候， 假如我們要將集合 a 與集合 b 聯集起來並以 a 命名， 我們只要把 b 對應的單連表列看一次、 將裡面的 i 都把對應的 NAME[i] 都改成 a， 然後把兩個表列串在一起即可。 這麼一來， 每次聯集所需的時間就從 n 改進成爲 b 的大小。 雖然這樣的改進， 大部分時候都很管用， 但是可能出現一種極端的情況， 那就是， 第一次 b 的大小是 1、 第二次是 2、 ... 一直到第 $n-1$ 次時是 $n-1$， 這樣的話， 整體用的時間還是 $O(n^2)$。

可以再做一點點小小的改進， 那就是， 再定義一個陣列 SIZE、 其中 SIZE[a] 用來記錄每個集合 a 的大小。 在上述聯集的過程中， 如果發現 SIZE[a] < SIZE[b]， 那我們就改將集合 a「併入」集合 b 中， 也就是說每次我們都只去改比較小的那個集合的元素所對應的 NAME。 這個看起來很小的修改， 就可以使得整體的演算法複雜度降低到 $O(n \log n)$ 了 （見習題 2.17）。

由以上的討論可以看出來， 聯集尋找問題經典之處就在於， 聯集和尋找這兩種動作剛好有點衝突， 要是資料結構設計得使得其中一邊的執行速度快， 另一邊往往就會慢。 下面介紹一個更有效率的方法。

假設先著眼於聯集這個動作， 那麼一個好的想法是採用指標結構。 把每個集合的元素用指標關係串連成一個樹狀的結構（**有根樹**、rooted tree）， 其頂端（**根**、root）就作爲這整個集合的代表。 也就是， 有一個長度是 n 的 1-維陣列 P， P[i] 表示 i 在樹中的父親， 而當 P[i] = 0 時就表示 i 是某一棵樹的頂端， 這棵樹所代表的集合的名子就是 i。 這時候， 如果想判斷一個點 i 屬於那一個集合， 只要沿著指標走： i、P[i]、P[P[i]]、...，直到 $\mathrm{P}^s[i]$ 爲 0 樹根就知道了。 而做聯集動作就更簡單了， 只要把其中一個集合的代表指向另外一個集合的代表， 一個動作就完成了， 參見圖 2.14 的例子[3]。 這樣的指標結構比之前的單連表列要複雜一點， 但是利用同樣的想法， 也不難

[3]樹的根在自然界是長在下方， 在圖論裡面， 爲了方便常常將根畫在上方， 可比失根的蘭花。

用陣列實際將它表現出來。

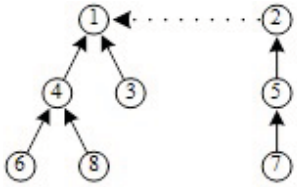


圖 2.14: 一個聯集的動作。

在圖 2.14 中， 根據這個結構， 6 跟 3 都是屬於「1」所代表的集合當中， 而 5 跟 7 則是在 2 所代表的集合裡。 如果我們想把這兩個集合做聯集， 只要把 2 再次指向 1， 一切就完成了， 也就是， 將 P[2] 由原來的 0 變成 1。

這個想法是很不錯， 但是有個小缺點， 那就是這些集合的樹狀結構可能會拉得很長； 也就是說， 每個資料離代表元素的距離有可能會非常遠， 這就導致尋找的速度變慢， 每次都要走很遠才能確定一個元素到底屬於那一個集合。

不過這個問題是有辦法解決的， 只要有辦法維持這些結構不要太長就行了。 這邊有兩個小技巧。 第一個和前面一樣， 我們用陣列記錄每一個集合的大小[4]， 而當我們要做聯集時， 永遠把比較小的集合指向大的集合的代表， 這樣出來的樹狀結構會好一點。 另外一個技巧稱爲**路徑壓縮**（path compression）。 路徑壓縮的想法是， 既然尋找的動作會花上較多時間， 那就來想辦法減少尋找的時間。 辦法是這樣， 以圖 2.14 中合併後的樹爲例， 當我做「尋找 7」這個動作時， 它會循著 7、5、2 最後找到 1， 而在這個過程當中， 我們就順便把 7、5 也一起指向 1：

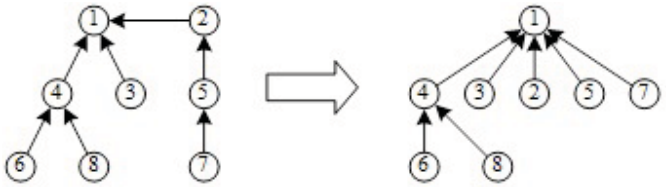


圖 2.15: 進行尋找時順便做路徑壓縮。

圖 2.15 顯示出經過尋找之後順便改變了指標的結構。 這麼一來， 要是下次再有機

[4]直觀上應該是要記錄樹的高度才對， 但是在相關證明中可以發現， 樹的大小也可以達到相同目的。而且在下面用到路徑壓縮的技巧時， 樹的高度是不容易計算的、而樹的大小保持不變。

會尋找路過 7 的時候， 它離代表元素就只有一步之遙， 大大地節省了時間。 下面是相關的程式：

```
union-find-initialization :
    input n ;
    for i = 1 to n do { P[i] ← 0 ; SIZE[i] ← 1 ; }
union(i, j) :
    if (SIZE[i] > SIZE[j])
    then { SIZE[i] ← SIZE[i] + SIZE[j] ; P[j] ← i ; }
    else { SIZE[j] ← SIZE[i] + SIZE[j] ; P[i] ← j ; }
find(i) :
    r ← i ; while (P[r] ≠ 0) r ← P[r] ;
    s ← i ; while (s ≠ 0) { t ← P[s] ; P[s] ← r ; s ← t] ; }
    return r ;
```

把這些概念全部整合的演算法稱爲**高速互斥集聯集法**（fast disjoint-set union algorithm）， 這個方法相當地快， 演算法上可以證明（有興趣的讀者可參見 [1]），如果我們要用這個方法處理 n 個元素的集合之聯集與尋找動作各 $O(n)$ 次， 則所需的時間是 $O(nG(n))$。 $G(n)$ 是什麼呢？ 它是這樣定義的： 考慮一個遞迴定義的函數 $F(n)$， 其中 $F(0) = 1$、 $F(i) = 2^{F(i-1)}$； 於是 $F(1) = 2$、 $F(2) = 4$、 $F(3) = 16$、 $F(4) = 65536$ 到 $F(5) = 2^{65536}$ 已經是個近兩萬位的天文數字了； 而 $G(n)$ 則定義爲使得 $F(k) \geq n$ 的最小 k 值。 這是一個成長得極慢的函數， 因此我們說， 應用這個演算法所需的時間幾乎是線性的。 實際應用上， 我們幾乎永遠沒有機會遇到 n 使得 $G(n) = 6$ 的那一天。

2.7. 習題

2.1. 用 $\gcd(m, n)$ 表示非負整數 m 和 n 的最大公因數。 試證：如果 $m = an + r$， 其中 a 和 r 是整數且 $0 \leq r < n$， 則 $\gcd(m, n) = \gcd(n, r)$。 並證明：這可以用來構造出一個用 $O(\log m + \log n)$ 個除法的演算法。

2.2. 要驗證一個正整數 n 是否爲質數， 可以檢驗從 2 到 $n-1$ 的整數 i 是否整除 n， 這是一個用 $O(n)$ 個除法的演算法。 請設計一個用 $O(\sqrt{n})$ 個除法的演算法。

2.3. 針對同一個問題， 假設演算法甲對於參數大小爲 n 的範例需用 n^2 個步驟來執行、 而演算法乙則需要 2^n 個步驟。

(1) 如果目前的電腦每秒可執行 10^{20} 步， 那麼一分鐘內上述演算法可以完成大小多少的範例？ 一小時呢？ 一年呢？

(2) 如果電腦進步到速度變爲原來的一百萬倍， 那麼上述演算法的容量改進爲何？

2.4. 用 $n-1$ 次比較可以找出 n 個實數中最大（或最小）的數， 所以用 $2n-2$ 次比較可以找出 n 個實數中最大的數及最小的數。 請設計出用更少次比較找出 n 個實數中最大的數及最小的數。 這是可能的，例如，只要用 1 次比較就可以找出 $n=2$ 個實數中最大的數及最小的數， 用 3 次比較就可以找出 $n=3$ 個實數中最大的數及最小的數。

2.5. 給定實數列 $x_1, x_2, \ldots, x_n$， 要從中找出一段連續的若干項使其和爲最大。 第 2.2 節中已經舉出了一個需時 $O(n^2)$ 的方法， 試找一個比這個更有效率的演算法。

2.6. 設 A 是大小爲 $m_1 \times m_2 \times \ldots \times m_d$ 的 d-維陣列， 其中每個元素都佔 s 位元， 而 $A_{1,1,\ldots,1}$ 所在的記憶體位置爲 B， 試求 $A_{i_1,i_2,\ldots,i_d}$ 的位置。

2.7. 單連表列 A 和 B 的資料存在陣列 DATA 中、 而其指標維陣列是 NEXT。 下面這個程式將這兩個單連表列聯合起來放在 A 中。

t = A;

while NEXT[t] $\neq$ 0 **do** t = NEXT[t];

NEXT[t] = B。

請修改資料結構， 使得將兩個單連表列聯合起只需做 $O(1)$ 次動作。

2.8. 雙連表列 A 的資料存在陣列 DATA 中、 而其指標維陣列是 NEXT 和 PREV。

(1) 請寫出程式，將新資料 x 加入 A 中、放在 i 後面。

(1) 請寫出程式，將 A 中的一項 i 刪除。

2.9. 假設圖 $G=(V,E)$ 有 n 個頂點 $v_1,v_2,\ldots,v_n$ 及 m 條邊 $e_1,e_2,\ldots,e_m$， 定義 G 的**相連矩陣**（incidence matrix） $M=[b_{ij}]$ 是一個 $n\times m$ 矩陣， 其中

$$b_{ij}=\begin{cases}1, & \text{若 } v_i \text{ 和 } e_j \text{ 相連；}\\ 0, & \text{若 } v_i \text{ 和 } e_j \text{ 不相連。}\end{cases}$$

試證： $MM^{\mathsf{T}}=A+D$， 其中 A 爲 G 的相鄰矩陣、 D 爲 $d_{ii}=\deg(v_i)$ 而 $i\neq j$ 時 $d_{ij}=0$ 的 $n\times n$ 矩陣。

2.10. 假設有向圖 $G=(V,E)$ 有 n 個頂點 $v_1,v_2,\ldots,v_n$ 及 m 條邊 $e_1,e_2,\ldots,e_m$， 定義 G 的**相連矩陣**（incidence matrix） $M=[b_{ij}]$ 是一個 $n\times m$ 矩陣， 其中

$$b_{ij}=\begin{cases}1, & \text{若 } v_iv_{i'}=e_j\text{；}\\ -1, & \text{若 } v_{i'}v_i=e_j\text{；}\\ 0, & \text{若 } v_i \text{ 和 } e_j \text{ 不相連。}\end{cases}$$

試證明：M 是**完全單模**（totally unimodular）， 也就是、M 的任意子矩陣的行列式是 0、1 或 -1。

2.11. (1) 請找出一序列無窮多個圖、其相連矩陣都不是完全單模。

(2) 試證明：二分圖的相連矩陣都是完全單模。

2.12. 令 A 爲圖 G 的相鄰矩陣。 對於非負整數 k， 設 $A^k=[c_{ij}]$ 而 $(A+I)^k=[d_{ij}]$。 請問 c_{ij} 和 d_{ij} 分別代表什麼量？

2.13. 假設圖 $G=(V,E)$ 中 $V=\{1,2,\ldots,n\}$ 而 E 則以 m 個有序對表示。 我們可以用**連續空間**（sequential space）來儲存 G 中各點的鄰居， 也就是在陣列 DATA 中先存 $N(1)$、 然後接續著存 $N(2)$、 ... 依此類推直到放完 $N(n)$ 爲止； 我們以 b_i 表示資料 $N(i)$ 在 DATA 中的起始位置， 於是 $N(i)$ 所在的範圍就是從 DATA$[b_i]$ 到 DATA$[b_{i+1}-1]$ 這一段 （其中定義 b_{n+1} 爲資料結束後的下一個位置）。 b_i 的值存放在陣列 BEG 內。 例如， 設 $n=9$， 而

$$E=\{31,\ 41,\ 59,\ 26,\ 53,\ 58,\ 97,\ 93,\ 23,\ 84\}\text{，}$$

則此時 DATA 和 BEG 的資料分別如下：

試寫一個演算法， 能讓使用者輸入 n 和 E， 產生出對應的 DATA 和 BEG。

BEG	1	3	5	9	11	14	15	16	18	21											
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
DATA	3	4	6	3	1	5	9	2	1	8	9	3	8	2	9	5	4	5	7	3	
	N_1		N_2		N_3				N_4		N_5			N_6	N_7	N_8		N_9			

圖 2.16: 連續空間儲存法，其中 N_i 表示 $N(i)$。

2.14. 在第 2.5 節 Euler 迴路的案例中， 改用連續空間表示圖 2.11 的圖、 但同時要含 SAME 和 MARK 兩個欄位。

2.15. 在第 2.5 節 Euler 迴路的案例中， 利用連續空間表示圖、據以改寫程式 input-graph。

2.16. 在第 2.5 節 Euler 迴路的案例中， 利用連續空間表示圖、據以改寫程式 Euler-tour。

2.17. 在聯集尋找問題中， 我們若採用 NAME、 集合表列與 SIZE 的方法， 試證明， 若演算法總共做了 $n-1$ 次聯集的運算， 那需時會是 $O(n \log n)$。

2.8. 參考文獻

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

[2] S. A. Cook, The complexity of theorem-proving procedures, *Proc. 3rd Ann. ACM Symp. on Theory of Computing Machinery*, New York, 1971, pp. 151-158.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Second Edittion, The MIT Press, Cambridge, Massachusetts, 2002.

[4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, California, 1978.

[5] M. C. Golumbic, *Algorithmic Graph Theory and Perfect Graph*, Academic Press, New York, 1980.

[6] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Coumpter Science Press, 1978.

[7] R. M. Karp, Reducibility among combinatorical problems, in R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 85-104.

[8] U. Manber, *Introduction to Algorithms, A Creative Approach*, Addison-Wesley, Reading, Massachusetts, 1989.

[9] D. Hillis 著，林遠志、陳振男譯，《電腦如何思考》，天下文化，1999 年。

[10] N. Macrate 著，楊昌裔譯，《馮紐曼—現代電腦發展的先驅》，牛頓出版公司，民國 85 年。

第 3 章　樹

愚者如我作詩，上帝造樹。

– Joyce Kilmer

3.1.　樹是簡單但重要的圖

一個圖裡面如果有許多圈， 則點與點之間就會有各式各樣不同的路徑連接它們， 整個圖的結構因此就變得複雜。 反過來， 不含圈的圖， 結構就簡單許多， 大部份問題如果限制在這樣的條件下， 研究起來就特別容易。 繼 Euler 迴路有關圖論起源的介紹， 本書就從沒有迴圈的連通圖、 也就是樹[1]（tree） 開始。

在圖論的研究裡， 樹可以扮演一種特殊的角色， 那就是， 在探討一個問題的時候， 如果對一般圖無從下手， 可以先把問題限制在樹， 當作暖身開始。 如果問題在樹這類圖可以解， 再嘗試更一般的圖。 如果問題在樹這類圖都沒辦法解， 那表示這個問題可能很難。

第 1 章提到的 Ulam 猜想 [20] 就是一個很好的例子， 這是一個看似容易， 至今還無人能解的難題， Kelly [11] 首先證明了 Ulam 猜想對於樹是會成立的， 後來陸續有人研究得出其他各圖類上[2] Ulam 猜想的解答， 但是對一般的圖還是沒有答案。

[1]這樣命名， 是因爲這種圖長得像樹木、 從根往上長出來又不形成迴圈的植物。 動植物相關的名詞經常在數學中出現， 例如， 代數幾何當中也有所謂的莖跟芽， 演算法中有種子， 圖論裡除了樹、葉之外，還有花、仙人掌、毛毛蟲等。

[2]含： 正則圖（regular graphs）、 非連通圖（disconnected graphs）、 單位區間圖（unit interval graphs）、 不含葉的可分離圖（separable graphs without end vertices）、 極大平面圖（maximal planar graphs）、 極大外平面圖（maximal outerplanar graphs）、 外平面圖（outerplanar graphs）、臨界區塊（critical blocks）； 參見 [2] [3] [7] [8] [9] [12] [14] [15] [16] [18] [19] [21] [22]。

圖論也有如下述， 限制在樹就已經是 NP-難的問題。 假設 A 是圖 G 的相鄰矩陣， **帶寬問題**（bandwidth problem）就是要將 A 的某些列調換、 同時也將對應的行同步調換， 使最後的矩陣中非 0 的元素盡量靠近對角線。 在這樣的結構下， 也可以用比較簡單、比較少的空間將圖儲存起來。 這個問題的另外一種描述方法是， 要將圖 G 的點排列成 $v_1, v_2, \ldots, v_n$， 使得

$$\max_{v_iv_j \in E(G)} |i-j|$$

的值越小越好。 舉例來說， n-圈 C_n 的點集是 $\{1, 2, \ldots, n\}$、 邊集是 $\{12, 23, \ldots, (n-1)n, n1\}$； 如果點排成 $1, 2, \ldots, n$ 則 $\max_{v_iv_j \in E(G)} |i-j| = n-1$ 很大； 但如果將點排成 $1, 2, n, 3, n-1, 4, n-2, \ldots$ 則 $\max_{v_iv_j \in E(G)} |i-j| = 2$ 達到最小可能。 圖的帶寬問題即使限制在樹都已經是 NP-難的問題， 對於一般的圖就更難了。

樹因爲結構簡單， 也成爲建模的一個好工具。 例如， 化學中比較簡單基本的分子， 其形狀都像樹。 計算同分異構物數目相關的問題， 就是問有 n 點的樹有多少個。 這種問題， 對於點名稱重要的**標號樹**（labelled trees）有解， 點名稱不重要的樹、 也就是同構的算同一顆樹、 沒有好的解。

在資訊領域裡， 樹更扮演多重的角色。 例如， 第 2 章用有根樹的結構表達集合， 得到聯集尋找問題的快速演算法。 設計圖的演算法時， 常用到的深度優先搜尋（depth-first search）和廣度優先搜尋（breadth-first search）， 對應的深度優先樹和廣度優先樹， 在演算法裡扮演重要的角色。

在架設電腦網路時會遇到如下的問題， 可以轉化成圖論的最優化問題。 考慮由 n 個電腦工作站連接成的網路， 一種連接的方法是， 把任兩個工作站之間都牽線， 此時對應的圖連成 K_n 的樣子。 但是這樣做是沒有必要的， 更重要的是、這樣做花費很大， 而且牽線操作困難。 簡單的方法是， 只要對應的圖是連通的， 網路就能夠運作， 所以其實只要按照一個 K_n 的連通生成子圖來架設網路即可。 有個問題； 基於工作站所處的地點與條件的差異， 在各組不同的工作站之間架設網路線所需要的施工成本也不盡相同。 那麼， 我們應該如何架設， 才能使得網路既是連通的、 又是所有可能的架設方法當中成本最低的？ 這就是， 最小生成樹的問題。

其他利用樹解結問解的例子包括， 最短路徑問題、 編碼理論的 Hoffman 樹等， 也都是很實用的圖論最優化問題。

3.2. 樹的基本性質

一個無圈的連通圖稱爲**樹**（tree），各個連通部分都是樹的圖稱爲**林**（forest）；林跟**無圈圖**（acyclic graph）的意思是一樣的。圖 3.1 列出、同構是爲相同的、所有 11 棵含有 7 個點的樹。

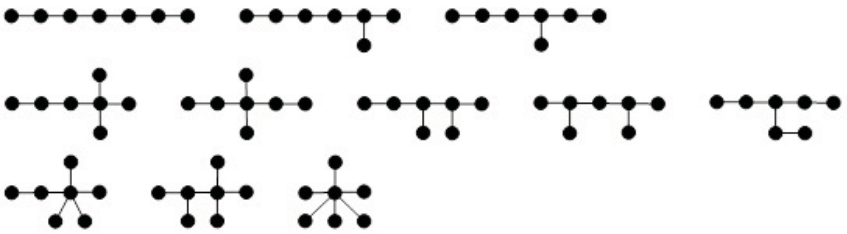


圖 3.1: 所有 11 棵恰有 7 點的樹。

樹除了上面的定義以外，還有許多不同的等價定義，在證明這些定義等價之前，先看幾個有用的性質。

性質 3.1. 若樹 T 最少含有兩點，則 T 最少有兩片葉。

證明： 從 T 當中選一條最長的路徑 P: $v_0, v_1, \ldots, v_r$。因爲 T 至少有一條邊，所以 $r \geq 1$。注意到 v_0 和 v_r 一定是葉，因爲如果它們還有跟其他點相鄰，該點必不在 P 當中，不然就會出現圈，於是 P 就可以再延長，矛盾。 ■

性質 3.2. 若 v 是樹 T 中的葉，則 $T - v$ 也是樹。

證明： 顯然 $T - v$ 中沒有圈。對於任何 $x, y \in V(T - v)$，因爲它們也是 T 中的點，由 T 的連通性知道，在 T 中存在一條 x-y 路徑，而 v 不會在這條路徑上（因爲它在 T 中的度數是 1），所以該路徑也是 $T - v$ 中的 x-y 路徑；這樣就證明了 $T - v$ 是連通的，所以它是樹。 ■

性質 3.2 是很重要的基本事實，這在利用數學歸納法證明樹的一些定理的時候常常用到。而在設計跟樹有關的演算法時，也常常一片葉一片葉地處理。意思是說，令 $T_1 = T$，而在 T_1 中取一片葉 v_1，然後令 $T_2 = T_1 - v_1$，並在 T_2 中取一片葉 v_2，如

此反覆下去， 直到最後 T_n 只剩一點 v_n 爲止。 也可以用下面的性質來描述： 一個圖 G 的**樹序**（tree ordering）是指將圖的點排爲 $v_1, v_2, \ldots, v_n$， 使得

對所有 $1 \le i < n$，點 v_i 恰有一鄰居 v_j 滿足 $i < j$。

例如考慮圖 3.2 (a) 左邊的這個樹， 我們將它重新安排成右邊的這個樣子； 即選定中間的那個點當作**根**（root）， 而其他的枝條跟葉都是從這點逐一長出來的。 依照這種安排， 將樹從最底層開始逐層編號， 如圖所示。 易看出這個順序滿足樹序的條件。

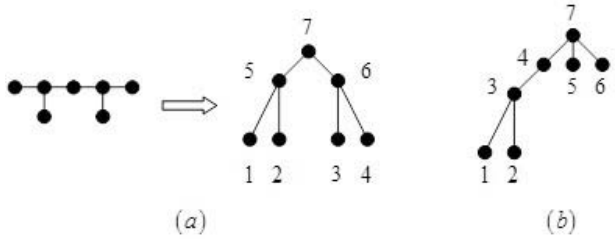


圖 3.2: 同一棵樹的兩種不同樹序。

一個圖的樹序不一定是唯一的， 例如圖 3.2 (b) 就是同一個圖的另外一種樹序， 其中我們選了另外一個點當作樹根。 基本上， 易看出樹中的任何一個點都可以當作樹根， 因而也給出了各種不同的樹序。

性質 3.3. 圖 G 有樹序，若且唯若 G 是樹。

證明： 從前面的構造過程中可以看出樹都有樹序。 反過來， 假如 G 有樹序， 根據條件， 對於所有的 v_i、$1 \le i < n$， 它都與唯一的 v_j 相鄰， 其中 $i < j \le n$。 因此易知每一點逐一的往編號比較大的點連、 最後就有一條路徑通往 v_n， 所以 G 是連通的。 接著， 假設 G 當中有圈 C， 則考慮這個圈在樹序當中排在最前面的那個點， 於是它必然與後面至少兩點相鄰， 矛盾； 所以就得到 G 是樹。 ■

於是， 在設計各種跟樹有關的演算法時， 經常會用迴圈「**for** $i = 1$ **to** n **do**」、按照給定的樹序一個一個地處理 v_i。 此時，當一個點被處理的時候， 如果不看那些已經處理過的點， 則它是一片葉， 這在設計演算法的時候會有幫助。 回憶起 G 的一個生成子圖是指包含 G 的所有點的子圖， 應用這個概念， 會有所謂的**生成樹**（spanning tree）。 有一些時候， 可以藉由研究 G 的生成樹來間接了解 G 的性質。

性質 3.4. 若連通圖 G 中有一個圈 C， 則對於 C 中的任一邊 e， $G-e$ 仍是連通的。

證明： 對於 $G-e$ 中的任意兩點 u 和 v， 它們也都是 G 中的點， 而因爲 G 是連通的， 它們之間存在一條 u-v 道路 W。 對於 W 當中任何有用到 e 的地方， 把它換成 $C-e$， 這將得到一條 $G-e$ 當中的 u-v 道路， 所以 $G-e$ 是連通的。 ■

推論 3.5. 任何連通圖都有一生成樹。

證明： 設 G 爲連通圖， 取其中一個邊數最少的連通生成子圖 H。 若 H 不是樹則 H 必有圈， 但這麼一來將這個圈的其中一條邊去掉， 根據性質 3.4 就得到一個邊數更少的連通生成子圖， 矛盾， 所以 H 必爲樹。 ■

最後， 來介紹一些樹的等價定義。 對於任何一類特殊的圖， 往往都不只一種方式可以刻畫它們， 而這些不同的刻畫方式往往會在不同的場合發揮作用， 所以多了解一些刻畫方式經常會有幫助。 底下介紹六種等價定義， 不過只證明前四種的等價性， 後面兩種留給讀者（參見習題 3.1）。 首先， 一個常用的性質是：

性質 3.6. 若樹 T 有 n 個點， 則 T 恰有 $n-1$ 條邊。

證明： 當 $n=1$ 時性質顯然成立。 假設 $n\geq 2$ 且性質對 $n-1$ 成立， 由性質 3.1 可以找到一片葉 v， 由性質 3.2 可知 $T-v$ 是樹， 而且比 T 少一個點及一條邊。 $T-v$ 恰有 $n-1$ 個點， 故由歸納法假設它有 $n-2$ 條邊， 於是就得到 T 有 $n-1$ 條邊。 ■

定理 3.7. 設圖 G 有 n 個點， 則下列敍述等價：
(1) 圖 G 是樹。
(2) 圖 G 無圈且恰有 $n-1$ 條邊。
(3) 圖 G 連通且恰有 $n-1$ 條邊。
(4) 圖 G 中任意兩點之間恰有一路徑。
(5) 圖 G 無圈，且任意加入一條新的邊都必然使得 G 有圈。
(6) 圖 G 連通且任意刪除一條邊必使得 G不連通。

證明： (1) $\Rightarrow$ (2)： 根據定義， G 本來就沒有圈， 而性質 3.6 則保證 G 有 $n-1$ 條邊。

(2) ⇒ (3)： 只需證明 G 連通。 假設 G 恰有 r 個連通部分 $G_1, G_2, \ldots, G_r$， 且分別有 $n_1, n_2, \ldots, n_r$ 個點， 於是 $n = \sum_{i=1}^{r} n_i$。 根據定義， 每一個 G_i 都是樹， 故由性質 3.6 知道 G_i 皆恰有 $n_i - 1$ 條邊， 於是 $n - 1 = \sum_{i=1}^{r}(n_i - 1) = n - r$， 得到 $r = 1$，所以 $G = G_1$ 是連通的。

(3) ⇒ (1)： 由推論 3.5， G 有生成樹 T， 由性質 3.6 知道 T 有 $n - 1$ 條邊。 但是 G 也只有 $n - 1$ 條邊， 所以 $G = T$ 是樹。

(1) ⇒ (4)： 由連通性， 任兩點之間都有一條路徑。 假設存在 u 和 v 使得其間有兩條相異路徑 P 和 Q， 不妨選取 u, v, P, Q 使得 P 和 Q 的長度和 s 是所有這樣的組合中最小的。 如果 P 和 Q 除了 u 跟 v 之外還有公共點 w， 則或者 u 和 w 之間有兩條相異路徑、 或者 w 和 v 之間有兩條相異路徑， 但無論哪一種情況， 都得到兩點使得它們之間有長度和小於 s 的相異路徑， 矛盾。 所以 P 跟 Q 沒有除了 u 跟 v 以外的共用點， 於是就形成了一個圈， 矛盾。

(4) ⇒ (1)： 根據定義， G 是連通的。 如果 G 中有圈， 則這個圈上的任意兩點之間沿著圈必有兩相異路徑， 矛盾。 ■

利用性質 3.1 和 3.2 證明性質 3.6 時所用的歸納法， 在很多跟樹有關的論證當中都會用到， 底下再舉一個例子說明。

性質 3.8. 設 T 是一個恰有 k 條邊的樹， 而圖 G 的最小度 $\delta(G) \geq k$， 則 G 有一個子圖與 T 同構。

證明： $k = 0$ 時性質顯然成立。 設 $k > 0$， 而且性質對於 $k - 1$ 成立。 找一個 T 中的葉 v、 並設它在 T 中的鄰居爲 u， 令 $T' = T - v$， 於是 T' 是有 $k - 1$ 條邊的樹。 由歸納法假設， G 中有一個子圖 G' 和 T' 同構， 假設 G' 中對應於 u 的頂點是 u'。 則因爲 $\deg_G(u') \geq \delta(G) \geq k = |V(G')|$， 所以 u' 在 G 中必有一個鄰居 v' 不在 $V(G')$， 於是將 G' 加上點 v' 與邊 $u'v'$ 就會是與 T 同構的子圖。 ■

Erdős 和 Sós 提出一個比性質 3.8更強的猜想：

猜想 3.9. （Erdős-Sós 猜想 [5]） 如果圖 G 有 n 個點和 m 條邊、 而樹 T 有 k 條邊， 則在 $m > n(k-1)/2$ 的條件下， G 會有一個與 T 同構的子圖。

容易看出， 當 $\delta(G) \geq k$ 時 $m > n(k-1)/2$ 一定會成立， 但反之則不一定。

3.3. 樹的中心問題

圖 G 中兩點 u 和 v 之間的**距離**（distance）、記作 $d_G(u,v)$ 或 $d(u,v)$，是指從 u 到 v 的道路中長度最小者之長度。如果 u 到 v 之間沒有道路，方便起見規定 $d(u,v)=\infty$；如果 G 是連通圖，$d(u,v)$ 都會是有限的。在這個定義中，把「道路」換成「路徑」並沒有差別，因爲在性質 1.5 的證明當中已經知道，最短的道路一定是一條路徑。對於任一連通圖 $G=(V,E)$，距離函數 d 是定義在 V 上的一個**量度**（metric），即它滿足下面三個條件：

(M1) **正定性：** 恆有 $d(u,v)\geq 0$，且 $d(u,v)=0$ 若且唯若 $u=v$；

(M2) **對稱性：** $d(u,v)=d(v,u)$；

(M3) **三角不等式：** $d(u,v)+d(v,w)\geq d(u,w)$。

(M1) 顯然成立。(M2) 成立是因爲一條 u-v 道路的反向是一條 v-u 道路。而要檢查 (M3) 成立，只要注意到任何的 u-v 道路和 v-w 道路串接起來都是 u-w 道路即可。

圖 G 中一個點 u 的**離心率**（eccentricity）$e_G(u)$、或是簡記爲 $e(u)$，是指從 u 到其他點的最大距離。而一個圖 G 的**半徑**（radius）$\mathrm{rad}(G)$ 是所有點的最小離心率，**直徑**（diameter）$\mathrm{diam}(G)$ 則是最大離心率。請特別注意，雖然直觀上圖的半徑跟直徑與這兩個名詞在幾何上的意義有一些類似之處，但是並不能互通[3]，例如圖的直徑不見得是其半徑的兩倍；這個等一下會有更仔細的討論。

如果 $e(u)=\mathrm{rad}(G)$，就說 u 是 G 的一個**中心點**（central vertex）；而**中心**（center）$C(G)$ 則是指所有中心點所構成的集合。有時爲了方便敍述，$C(G)$ 也有可能是指由 $C(G)$ 所導出的子圖。

舉例來說，完全圖 K_n 的每一點 u 均有 $e(u)=1$，所以 $\mathrm{rad}(K_n)=\mathrm{diam}(K_n)=1$。完全二部圖 $K_{m,n}$ 的每一點 u 均有 $e(u)=2$，所以 $\mathrm{rad}(K_{m,n})=\mathrm{diam}(K_{m,n})=2$。圈圖 C_n 的每一點 u 均有 $e(u)=\lfloor n/2\rfloor$，所以 $\mathrm{rad}(C_n)=\mathrm{diam}(C_n)=\lfloor n/2\rfloor$。路徑圖 P_n、其點集爲 $\{v_1,v_2,\ldots,v_n\}$、邊集爲 $\{v_1v_2,v_2v_3,\ldots,v_{n-1}v_n\}$，圖中點 v_i 有 $e(v_i)=\max\{i-1,n-i\}$，所以 $\mathrm{rad}(P_n)=\lceil n/2\rceil$、$\mathrm{diam}(P_n)=n-1$ 且 $C(P_n)=\{v_{\lfloor(n+1)/2\rfloor},v_{\lceil(n+1)/2\rceil}\}$。

[3]其實，在平面上圓的半徑，不是圓上所有點到圓上各點最大距離的最小值、而是平面上所有點到圓上各點最大距離的最小值，這與圖的定義不同。只有將圓換成圓區域，才會得到與圖相同精神的定義。

性質 3.10. 任何圖 G 都是某一個圖的中心； 也就是說， 能夠構造出一個圖 H， 使得 $C(H)$ 與 G 同構。

證明： 令 H 是在 G 中加上四個新點 x, y, z, w、 以及 xy, yu, uz, zw（其中 u 過 G 的所有點）這些邊所成的新圖（見圖 3.3）。 易看出 $e(x) = e(w) = 4$、 $e(y) = e(z) = 3$， 且對所有 $u \in V(G)$ 有 $e(u) = 2$ ， 因此 G 就是 H 的中心。 ■

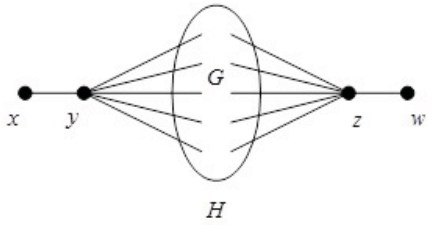


圖 3.3: 圖 H 是圖 G 再加上四點所成的圖。

由上面的性質可以知道， 一個圖的中心可以是任何樣子的圖； 然而有些特殊構造的圖的中心卻相當簡單， 例如 K_n、$K_{m,n}$、C_n、P_n 等都是。 一般的樹也是一個好例子。

定理 3.11.（Jordan [10]） 樹的中心或者恰爲一點、 或者恰含兩相鄰點。

證明： 這裡提供兩個證明， 一個較簡短， 一個有利於設計演算法求樹的中心。

證法一： 假設樹 T 有兩個不相鄰的中心 x 和 y， 以 $P(x, y)$ 表示在樹裡面 x 到 y 的唯一路徑。 取 $P(x, y)$ 中異於兩端點的一點 z。 令 w 是距離 z 最遠的一點、 也就是 $e(z) = d(z, w)$。 如果 $P(z, w)$ 的第二點是 $P(z, y)$ 的第二點， 則 $P(z, w)$ 不會包含 $P(z, x)$ 中異於 z 的點， 否則就會產生圈， 所以 $P(z, w) \cap P(z, x) = \{z\}$； 如果 $P(z, w)$ 的第二點不是 $P(z, y)$ 的第二點， 則同裡 $P(z, w)$ 不會包含 $P(z, y)$ 中異於 z 的點， 所以 $P(z, w) \cap P(z, y) = \{z\}$； 不失一般性可假設前者成立， 此時 $P(x, z)$ 和 $P(z, w)$ 合成 $P(x, w)$， 所以

$$\mathrm{rad}(T) = e(x) \geq d(x, w) = d(x, z) + d(z, w) > d(z, w) = e(z)，$$

矛盾。所以定理得證。 □

證法二： 利用數學歸納法證明定理。 假設樹 $T = (V, E)$ 有 n 個頂點。 當 $n \leq 2$ 時定理顯然成立； 當 $n \geq 3$ 時， T 至少有一點不是葉 （否則， $n = \sum_{v \in V} \deg(v) = 2|E| = 2(n-1)$ 將導致 $n = 2$）， 此時令 L 爲 T 中所有葉的集合， 而 $T' = T - L$。 由性質 3.2， T' 是一棵樹。

對於任一點 $x \in L$， 若它在 T 中的鄰居是 y（此時必定 $y \notin L$）， 則因爲 T 中以 x 爲起點的道路都會通過 y， 所以 $e_T(x) = e_T(y) + 1$， 也就是說 $C(T) \cap L = \emptyset$。 底下對於任意的 $x \in V - L$， 將證明 $e_{T'}(x) = e_T(x) - 1$， 因此得到 $C(T') = C(T)$， 而由歸納法就可以得知 T 的中心不是一點就是相鄰兩點， 視其化簡到最後是一點還是兩點而定。

令 $r = e_T(x)$ 且 $r' = e_{T'}(x)$。 假設 $v_0, v_1, \ldots, v_r$ 是 T 中以 $x = v_0$ 爲出發點的一條最長路徑， 此時對於 $0 \leq i \leq r-1$ 恆有 $v_i \notin L$， 所以 $v_0, v_1, \ldots, v_{r-1}$ 是 T' 中的路徑， 由樹中兩點之間路徑的唯一性可知， $e_{T'}(x) \geq r - 1 = e_T(x) - 1$。 反過來， 令 $u_0, u_1, \ldots, u_{r'}$ 是 T' 中以 $x = u_0$ 爲出發點的一條最長路徑， 則 $u_{r'}$ 是 T' 中的葉或者是 T' 唯一的點， 不然 $u_{r'}$ 就會和 T' 中異於 $u_{r'-1}$ 的某點 u 相鄰， 因此 $u_0, u_1, \ldots, u_{r'}, u$ 是 T' 中的一條路徑， 這導致 $e_{T'}(x) \geq r' + 1$ 而矛盾。 既然 $u_{r'}$ 是 T' 中的葉或唯一點， 但又不在 L 中， 因此 $u_{r'}$ 必和某 $w \in L$ 相鄰， 得到 $e_T(x) \geq r + 1 = e_{T'}(x) + 1$。 合起來即得到 $e_{T'}(x) = e_T(x) - 1$。 □ ■

上述的證法二除了有利於設計演算法求樹的中心之外， 因爲對於 $x \in V - L$ 恆有 $e_{T'}(x) = e_T(x) - 1$， 所以推得 $\mathrm{rad}(T') = \mathrm{rad}(T) - 1$。

另一方面， 當 x 是 T 的葉而和 y 相鄰時， 不但如上述可以證得 $e_T(x) = e_T(y) + 1$， 若再多一點論證， 其實也能說 $e_T(x) = e_{T'}(y) + 2$， 更進一步可以推導出 $\mathrm{diam}(T') = \mathrm{diam}(T) - 2$。

所以， 其實不但可以說明樹的中心的長相， 也可以確定其直徑跟半徑之間的關係：

推論 3.12. 於任一樹 T， 或者 $C(T)$ 恰含一點， 此時 $\mathrm{diam}(T) = 2\,\mathrm{rad}(T)$； 或者 $C(T)$ 恰含相鄰兩點， 此時 $\mathrm{diam}(T) = 2\,\mathrm{rad}(T) - 1$。

對於一般的連通圖 G 而言， 我們只能說 $\mathrm{rad}(G) \leq \mathrm{diam}(G) \leq 2\,\mathrm{rad}(G)$。 而且， 一般來說， 若兩非負整數 a 和 b 滿足 $a \leq b \leq 2a$， 我們都有辦法找到一個圖 G 使得 $\mathrm{rad}(G) = a$ 且 $\mathrm{diam}(G) = b$ （參見習題 3.7）。

Jordan 在他的文章中還提到了一個中心的變型概念。 定義一棵樹 T 在一點 u 的一個**分叉**（branch at u）是指一棵包含 u 爲葉的 T 之極大子樹， 也就是 $T-u$ 的一個連通部分、 再加上點 u 以及 u 連到這個連通部分的唯一邊。 點 u 的**分叉權重**（branch weight）$b(u)$ 是指 u 的分叉當中邊數最多者之邊數， 而 T 的所有點當中分叉權重最小的點則稱爲是 T 的**質心點**（centroid vertex）。 在圖 3.4 這個有 16 條邊的樹當中， 已經標示出所有點的分叉權重， 例如分叉權重爲 8 的點有 3 個分叉， 分別有 8、6、2 條邊。

值得注意的是， 一個點是否爲質心點和它是否爲中心點是無關的。 例如圖 3.4 中， 分叉權重爲 8 的點是唯一的質心點卻不是中心點， 而分叉權重爲 9 的點是唯一的中心點卻不是質心點。

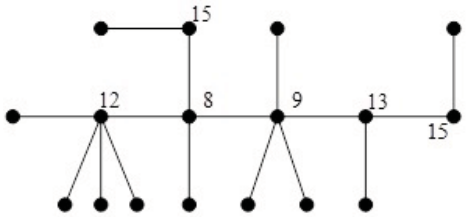


圖 3.4: 一棵樹以及其所有點的分叉權重（各葉的權重皆爲 16）。

Jordan 給出了下面這個和定理 3.11 類似的結論：

定理 3.13.（Jordan [10]） 任何樹或者恰有一質心點、 或者恰有兩相鄰質心點。

中心點的概念還可以有各種變型， 端示對於「中心位置」的解讀而定。 前面利用 $e(u) = \max_{v\in V(G)} d(u,v)$ 定義出來的中心， 是考慮離 u 最遠的距離越小越好， 這照顧到最差狀況。 但是， 也許有些圖裡頭， 最差狀況其實很少， 例如 P_n 的一個端點 v_1 多連出 n^2 個葉的新圖中， 感覺起來 v_1 更像一個中樞點。 所以， 如果是用點 u 到各點的平均距離、 等價於到各點的距離總合當做判斷標準， 可以定義 $s(u) = \sum_{v\in V(G)} d(u,v)$， 而圖 G 的**中段**（median）是指由 $s(u)$ 最小的所有 u 所構成的集合。 這種觀念照顧到平均狀況。

類似的也有下面的結論：

定理 3.14. 樹的中段或者恰爲一點、或者恰含兩相鄰點。

3.4. 樹或圖的遍歷搜尋法

在設計圖論演算法的時候， 經常需要針對給定的樹或圖， 依某種特定的搜尋規則遍歷所有頂點。 如果給定的是連通圖， 這個遍歷搜尋法會產生該圖的一個生成樹及一些相關資訊， 這可以用來了解該圖的基本架構。

視演算法的需求， 可以設計不同的遍歷搜尋規則， 目前最常被採用的主要有兩種方法， 分別是**深度優先搜尋**（depth-first search，簡稱 DFS）和**廣度優先搜尋**（breadth- first search，簡稱 BFS）。 利用這兩種方法及產生的生成樹去處理一個圖，使得許多演算法會變得更容易理解、 執行得更快更有效率。

在各種遍歷搜尋法當中， 給定一個連通圖 G， 會先從 $V(G)$ 中選取一點 v 當作**樹根**（root）， 從它出發， 依照給定的搜尋規則遍歷所有頂點， 在遍歷頂點的過程中，從一個點經過一條邊到下一個點， 這條邊就是一條**樹邊**（tree edge）， 所有樹邊構成一個生成樹。 在生成樹架構當中， 每一個點 u 和樹根 v 之間都有一條唯一的路徑， 這條路徑上的所有點（含 u 在內）通稱爲 u 的**長輩**（ancestor）， 其中與 u 相鄰的長輩則稱爲 u 的**父點**（parent）； 反過來， 所有以 u 爲長輩的點（包括 u 自己）稱爲 u 的**晚輩**（descendant）， 而與 u 相鄰的晚輩則通稱爲 u 的**子點** （child）。 邊集 $E(G)$ 中沒有被選爲樹邊的邊可以分成兩種： 如果一條邊的兩個端點當中有其中一個是另一個的長輩， 就稱爲是**反向邊**（back edge）； 反之如果兩個點都不是對方的長輩， 那就稱爲**跨越邊**（cross edge）。

例如在圖 3.5 所示的生成樹中， 如果 v 爲樹根， 則 u 的長輩共有 u, v_1, v、 晚輩有 u, v_2, v_3, v_4、 父點爲 v_1、 子點有 v_2， 而 e_1 是一條跨越邊， e_2 是一條反向邊。

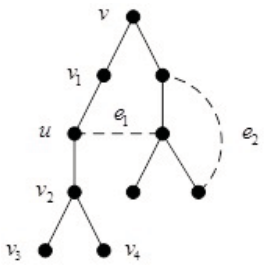


圖 3.5: 生成樹示意圖。

十九世紀時， 法國數學家 Trémaux 利用深度優先搜尋的策略解決迷宮問題， 這算是深度優先搜尋的起源。 轉換成圖論的語言， 深度優先搜尋的作法是， 先選擇一個點 a 做爲樹根， 然後前往 a 的一個鄰居 b， 並將 ab 選爲樹邊， 接著再前往 b 的一個還沒去過的鄰居 c， 將 bc 選爲樹邊， 依此類推， 每次都往一個還沒去過的鄰居前進， 並且把連接這兩點的邊選爲樹邊。 如此一來最終會來到一個點， 它所有的鄰居都已經到過了， 這個時候就返回到最後一個尚有還沒走過的鄰居的點， 並繼續行走。 這跟第 1 章中用來尋找 Euler 迴路的演算法有點像， 當時是針對邊「能走就走、不能走就回頭到一個較早的點，從未走過的邊走完之後再插入」； 而 DFS 則是針對點「能走就走、不能走就回頭到一個較早的點，從未走過的點繼續走下去」。 如果圖是連通的， 那麼這個作法將會走過每一個點。 如果圖不連通， 也可以執行 DFS， 那就在每一個連通部分做一次 DFS， 事實上應該反過來說， 是每一次 DFS 構造出一個連通部分。 DFS 在有向圖一樣適用， 差別只在每次只能沿著順向邊前往下一個鄰居。

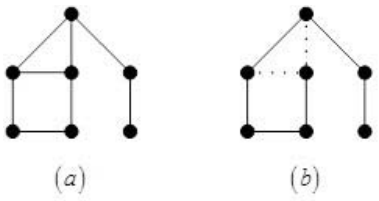


圖 3.6: (a) 原圖。(b) 以最上面的點爲樹根做 DFS 所產生的生成樹，虛線表示反向邊。

DFS 的一個特性是， 在無向圖的情況中， 邊就只有樹邊和反向邊兩種， 不會有跨越邊出現。 這並不難理解， 因爲如果第一次來到 x 之後， 發現它有一個不是長輩卻是曾經到過的鄰居 y， 那麼當時在處理 y 的時候應該就已經走過 x 並把 xy 這條邊選爲樹邊才對， 這跟假設矛盾。 因此在無向圖的情況中， 一條邊不是樹邊就是反向邊。 DFS 在有向圖就可能產生誇越邊。

如果要實作 DFS 的話， 可以採用堆疊結構來記錄走過了哪些點、 好讓走到盡頭的時候可以方便倒退。 還有一個方法是利用遞迴程序（就是一個會呼叫自己的程序）來實作。 爲了紀錄遍歷各點的次序， 每一點 v 可以用 DFS_NUM[v] 何時被搜尋到， 一開始 DFS_NUM[v]=0 表示點 v 還未被搜尋過。 具體的程式如下所示， 其中 X $\leftarrow$ ++Y 表示， 先將 Y 值增加 1 然後將此新值代入 X。

演算法 3.15.（深度優先搜尋）

輸入：連通圖 $G = (V, E)$。

輸出：樹邊之集合 T（而 $E \backslash T$ 就是反向邊之集合）。

方法：

1. 將 V 中所有點 v 設定 DFS_NUM[v] $\leftarrow 0$； CUR $\leftarrow 0$； $T \leftarrow \emptyset$；
2. 任選一個點 v 並執行 DFS(v)；

DFS(v)：

3. 設定 DFS_NUM[v] $\leftarrow$ ++CUR；
4. 每當（v 有一個滿足 DFS_NUM[u]=0 的鄰居 u ）
5. { $T \leftarrow T \cup \{uv\}$； 執行 DFS($u$)；}

對於一般可能不連通的圖 G， 將演算法第 3 行做適當修改如下， 每一次就可以得到一個連通部分。 這時候， T 是一個森林。

2'. 每當（G 有一個滿足 DFS_NUM[v]=0 的點 v） 執行 DFS(v)；

1950 年代，Moore 利用廣度優先搜尋去找走出迷宮的最短路徑， 這算是廣度優先搜尋的起源。 廣度優先搜尋的作法是先盡可能地把同一個頂點的所有鄰居都走過， 再前往下一個頂點去搜尋其鄰居。 首先選一個點 a 當樹根， 然後把 a 的每一個鄰居 $b, c, \ldots$ 走一遍， 並且將 $ab, ac, \ldots$ 這些邊都選爲樹邊。 接著， 剛才繼 a 之後第一個被走過的點是 b， 就對 b 進行同樣的操作， 做完之後再繼續對下一個最早被走過的點、 也就是對 c 繼續做同樣的操作。 有向圖的情況則是， 每次從一個頂點對所有它指向的所有鄰居都走過， 也就是， 只能挑出順向邊作爲樹邊。

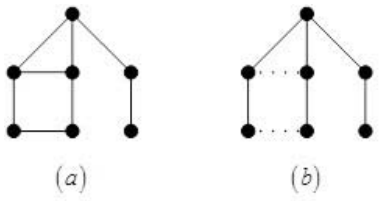


圖 3.7: (a) 原圖，(b) 以最上面的點爲樹根做 BFS 所產生的生成樹，虛線表示跨越邊。

跟 DFS 相反地， 在無向圖的情況中， 只會出現跨越邊而不會出現反向邊， 道理跟之前 DFS 的討論類似。

爲了要實作這個演算法， 可以利用佇列的資料結構， 每次抓鄰居的時候就排在佇列後面， 而處理完一個點的時候則從佇列的前端移除。 與 DFS 一樣， 爲了紀錄遍歷各點的次序， 每一點 v 可以用 BFS_NUM[v] 記錄何時被搜尋到， 一開始 BFS_NUM[v]=0 表示點 v 還未被搜尋到。 具體的程式如下所示。

演算法 3.16.（廣度優先搜尋）

輸入：連通圖 $G = (V, E)$。

輸出：樹邊之集合 T（而 $E \backslash T$ 就是誇越邊的集合）。

方法：

1. 將 V 中所有點 v 設定 BFS_NUM[v] $\leftarrow 0$；
2. CUR $\leftarrow 0$； $T \leftarrow \emptyset$； Q 爲空佇列；
3. 任選一個點 v 並執行 BFS(v)；

BFS(v)：

4. 將點 v 加入 Q； 設定 BFS_NUM[v] $\leftarrow$ ++CUR；
5. 每當（佇列 Q 非空）
6. { 將 Q 的第一個元素移出、並設其爲 v；
7. 對所有（滿足 DFS_NUM[u]=0 的 v 的鄰居 u）
8. { $T \leftarrow T \cup \{vu\}$； 將 u 加入 Q； 設定 BFS_NUM[u] $\leftarrow$ ++CUR；} }

與 DFS 一樣， 對於一般可能不連通的圖 G， 將演演算法 3 行做適當修改如下， 每一次就可以得到一個連通部分。 這時候， T 是一個森林。

3'. 每當（G 有一個滿足 BFS_NUM[v]=0 的點 v） 執行 BFS(v)；

後面的章節還會看到很多 DFS 和 BFS 在圖論演算法的應用， 例如， DFS 可用來測試圖的平面性、 決定圖的各種連通度、 找有向圖的拓撲排序、 測試有向圖的圈； 而 BFS 則可用來解決最短路徑問題、 測試圖的無弦圈、 解決網路流問題等等。

3.5. 生成樹計數

考慮所有以 $[n] = \{1, 2, \ldots, n\}$ 爲點集的圖， 其中同構但是頂點名稱不同的圖視爲相異的； 想要知道的是， 其中到底有幾種是樹？ 更一般而言， 給定一個圖、 甚至是重圖， 則它有多少生成樹？

第一個問題相當於是在後者當中取圖爲 K_n 的特例， 這個問題首先由 Cayley [4] 給出答案。 他原來的證明方式是用生成函數的技巧[4]， 這裡採用 Prüfer 的證明。

對於任一棵有 $n \geq 2$ 點的樹 $T = (V, E)$， 爲了方便可假設 $V \subseteq \mathbb{N}$， 令 $\mathcal{T}$ 表示所有這種樹構成的集合。 考慮集合 $V^{n-2} = \{(a_1, a_2, \ldots, a_{n-2}) : a_i \in V\}$， 並透過底下的演算法定義映射 $f : \mathcal{T} \to V^{n-2}$： 令 $T_1 \leftarrow T$，對 i 從 1 到 $n-1$ 執行

取 T_i 中編號最小的葉 b_i、 其唯一鄰居爲 a_i、 並令 $T_{i+1} \leftarrow T_i - b_i$。

也就是說， 每次都拔掉一片葉， 並且把跟該葉相鄰的點紀錄起來。 當 $i \leq n-2$ 時， a_i 不是 T_i 的葉， 更不會是 T 的葉。 這樣得到的序列 $f(T)$ 稱爲是樹 T 的 **Prüfer 碼**（Prüfer code）。 由於 $|V^{n-2}| = n^{n-2}$， 如果能說明 f 是一個**對射**（bijection，即一對一映成函數）， 那就可以得到：

定理 3.17.（Cayley [4]） 恰有 n^{n-2} 棵樹其點集 $V \subseteq \mathbb{N}$ 且 $|V| = n$。

證明： 定理當 $n = 1$ 時顯然成立。 對於 $n \geq 2$ 的情況， 我們準備用歸納法證明上面定義的 f 是一個對射。 這當 $n = 2$ 時成立， 因爲只有一棵兩點的樹， 而 V^0 當中也只有空序列 ( ) 而已。

接著假設定理對 $n = k$ 成立， 則當 $n = k+1$ 時， 對任何 $a = (a_1, a_2, \ldots, a_{n-2}) \in V^{n-2}$， 要求 $f(T) = a$ 的解。 注意到因爲 $n > 2$， 表示 $a_1, a_2, \ldots, a_{n-2}$ 都不會是 T 的葉。 同時， 這樣的 T 當中編號最小的葉一定是 V 當中沒有出現在 a 裡面的最小數（不妨稱爲 x）， 所以 T 一定有一個邊爲 xa_1。 考慮 $a' = (a_2, a_3, \ldots, a_{n-2}) \in (V \backslash \{x\})^{n-2}$， 由歸納假設， 恰存在一個不包含 x 的樹 T' 使得 $f(T') = a'$， 而將 T' 加回 x 與邊 xa_1 就得到 $f(T) = a$ 的唯一解。 ■

[4]用生成函數的求法也相當簡潔但須略多預備知識， 有興趣的讀者可以參考 [6]。 利用生成函數的手法， 可以有效地處理非常多不同種類關於樹的計數問題， 該書中同樣對這些技巧有詳細的討論。

一般而言， 用 $\tau(G)$ 表示近圖 G 的生成樹的個數。 Cayley 定理便是在說明 $\tau(K_n) = n^{n-2}$。 爲了計算一般的 $\tau(G)$， 可以引入「**收縮**（contraction）」的概念。 給定近圖 G 中的一個非迴邊 e， 將該邊與其兩端點「合成」爲一點， 得到的新近圖稱爲 G 的**收縮圖**（contraction）， 以 $G \cdot e$ 表示（有些書用 G/e）。 例如圖 3.8 就是一個收縮的例子； 在新圖 $G \cdot e$ 裡面， 將 u 跟 v 變成同一點， 讓 e 消失， 並且將本來其他的連邊關係繼續維持。 值得注意的是， 收縮一條邊的結果可能會產生新的重邊與迴邊。

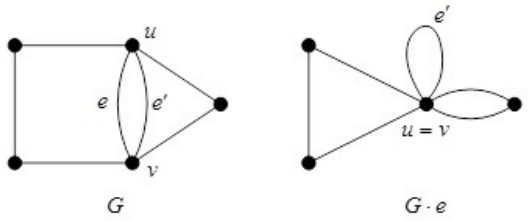


圖 3.8: 圖 G 及 $G \cdot e$。

性質 3.18. 若 e 是近圖 G 中一條不是迴邊的邊， 則 $\tau(G) = \tau(G - e) + \tau(G \cdot e)$。

證明： 任何 G 的生成樹都可以分成兩種， 一種是不包含 e 的、 一種是包含 e 的。 第一種的數目即爲 $\tau(G - e)$； 而第二種可以和 $G \cdot e$ 的生成樹一一對應。 考慮 G 的一個包含 $e = uv$ 的生成樹 T， $T \cdot e$ 就會是 $G \cdot e$ 的生成樹； 反之對於 $G \cdot e$ 的生成樹， 將 $u = v$ 那一點「擴張」成兩點並加回一條邊， 就會得到 G 的生成樹了。 ■

因爲 $G - e$ 與 $G \cdot e$ 都比原圖小， 上面給出了一個遞迴關係。 但是用這個方法計算 $\tau(G)$ 是不理想的， 甚至可以說是很糟糕的， 因爲每次做遞迴時圖的個數都變成兩倍， 而且可以預期的是， 需要經過非常多次遞迴關係才能得到便於計算的小圖， 這使得計算量跟資料儲存需求都大得驚人。 而底下應用矩陣來解決這個問題則會是一個比較好的方法。 以下的內容會用到一些線性代數的知識。

將近圖 G 當中的迴邊去掉並不會影響 $\tau(G)$ 的值， 所以只需討論重圖 G 的 $\tau(G)$ 就夠了。 對於任意的重圖 G， 設其點集爲 $\{v_1, v_2, \ldots, v_n\}$， 考慮 $n \times n$ 矩陣 $Q = (a_{i,j})$ 定義如下： 當 $i = j$ 時， $a_{i,j} = \deg_G(v_i)$， 而當 $i \neq j$ 時， v_i 和 v_j 之間恰有 $-a_{i,j}$ 條重邊。 以 K_n 爲例， 其矩陣就會型如：

$$\begin{bmatrix} n-1 & -1 & -1 & \cdots & -1 \\ -1 & n-1 & -1 & \cdots & -1 \\ -1 & -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & -1 & \cdots & n-1 \end{bmatrix}_{n\times n}$$

計算 $\tau(G)$ 的公式是透過行列式達成。

定理 3.19.（矩陣-樹定理，Matrix-Tree Theorem） 若將 Q 的第 s 列與第 t 行去掉後得到 Q^*， 則 $\tau(G) = (-1)^{s+t}\det(Q^*)$。

先來看一個例子。 仍以 K_n 爲例， 若取 $s = t = 1$， 則矩陣 Q^* 的長相跟本來的 Q 是一樣的， 只是階數降了一階而已。 利用列（行）運算操作如下： 首先將第 1 列以後的每一列減去第 1 列， 會得到：

$$\begin{bmatrix} n-1 & -1 & -1 & \cdots & -1 \\ -n & n & 0 & \cdots & 0 \\ -n & 0 & n & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -n & 0 & 0 & \cdots & n \end{bmatrix}_{(n-1)\times(n-1)}$$

然後再把除了第 1 行以外的每一行都加到第 1 行， 就會得到如下的矩陣。 其行列式值顯然爲 n^{n-2}， 因此 $\tau(K_n) = n^{n-2}$。 這樣就重新驗證了 Cayley 定理。

$$\begin{bmatrix} 1 & -1 & -1 & \cdots & -1 \\ 0 & n & 0 & \cdots & 0 \\ 0 & 0 & n & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & n \end{bmatrix}_{(n-1)\times(n-1)}$$

矩陣-樹定理的證明需要一些預備知識。 首先， 在第 2 章（以及其習題中）曾談過如何用相鄰矩陣和相連矩陣來表示一個圖。 對於重圖的情況中這兩種表示法一樣適用，只是這次相鄰矩陣的元素變成是指兩個頂點之間的邊數， 而相連矩陣可能會有重複的行而已。

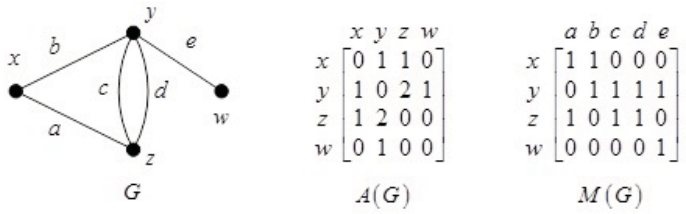


圖 3.9: 圖 G、其相鄰矩陣 $A(G)$ 與相連矩陣 $M(G)$ 。

對於一個無向圖 G， 任意指定邊的方向可以得到一個有向圖 G'， 而任何一種這樣的 G' 都稱爲是 G 上的一個**定向**（orientation）。 例如圖 3.10 中的 G' 就是圖 3.9 中的 G 的一個定向。

對於有向圖， 相鄰矩陣與相連矩陣的定義略有不同。 相鄰矩陣仍舊是以點爲行列，不過矩陣中第 (i, j) 元的數字是表示從 v_i 到 v_j 有多少條邊； 相連矩陣當中， 如果有向邊是從該點出發則記爲 1， 如果是在該點結束則記爲 -1。

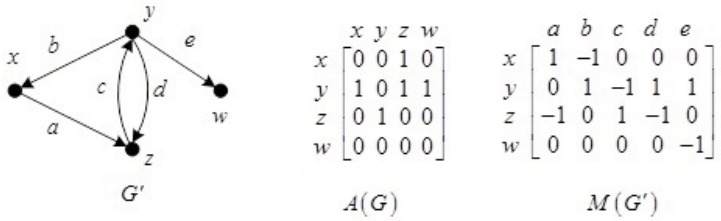


圖 3.10: 圖 G 的一個定向 G'、其相鄰矩陣 $A(G')$ 與相連矩陣 $M(G')$。

現在回到先前的重圖 G 以及 $n \times n$ 矩陣 Q。 首先需要一些引理。

引理 3.20. 若 M 是重圖 G 的一種定向 D 的相連矩陣， 則 $Q = MM^{\mathrm{T}}$。

證明： MM^{T} 的 (i, j) 元表示 M 的第 i 列和 M 的第 j 列的內積。 當 $i = j$ 時， 這個內積等於第 i 列非 0 的項數、 也就是 $\deg_G(v_i)$； 當 $i \neq j$ 時， 這個內積等於 v_i 和 v_j 之間邊數的負值。 這都跟 Q 的情況相符。 ■

引理 3.21. 設 B 爲 M 的某個 $(n-1) \times (n-1)$ 子矩陣， 則 $\det B = \pm 1$ 若 B 的行對應的 $n-1$ 條邊構成 G 的生成樹； 不然 $\det B = 0$。

證明： 如果B 的行對應的 $n-1$ 條邊構成 G 的生成樹， 對 n 做歸納法證明。 當 $n=1$ 時， 根據通例， 空矩陣的行列式值爲 1。 設引理對 $n=k$ 成立， 則當 $n=k+1$ 時， 令 T 爲B 的行對應的邊所構成的生成樹， 則因爲 T 至少有兩片葉， B 裡面就至少有一列是對應於 T 中的葉（不妨稱爲 x）。 因爲 x 是 T 的葉， 所以 B 的 x 列當中只有一個非零元素。 對 B 的 x 列做降階以計算行列式值， 則得到的矩陣 B' 就對應於 $G-x$ 的生成樹（即 $T-x$）， 於是由歸納法假設得知 $\det B'=\pm 1$， 因此 $\det B=\pm 1$。

如果 B 的行對應的 $n-1$ 條邊不構成 G 的生成樹， 則由定理 3.7 等價關係 (2)， 這些邊會包含有一個圈 C。 把那些構成 C 的邊對應的那幾行挑出來， 把順向邊乘以 1、 逆向邊乘以 -1 加起來的話就會得到零（因爲頭尾互相抵銷）， 表示這幾行是線性相關的， 於是 $\det B=0$。 ■

引理 3.22.（Binet-Cauchy 公式） 設 A 爲 $n\times m$ 矩陣， B 爲 $m\times n$ 矩陣。 給定 $S\subseteq[m]$ 使得 $|S|=n$， 令 A_S 爲那些由 S 所標示的行所構成的 $n\times n$ 矩陣， 而 B_S 爲那些由 S 所標示的列所構成的 $n\times n$ 矩陣。 若 $C=AB$， 則 $\det C=\sum_S(\det A_S)(\det B_S)$， 其中 S 過 $[m]$ 的所有 n-子集。

證明： 考慮等式

$$\begin{bmatrix} I_m & 0 \\ A & I_n \end{bmatrix}\begin{bmatrix} -I_m & B \\ A & 0 \end{bmatrix}=\begin{bmatrix} -I_m & B \\ 0 & AB \end{bmatrix},$$

注意到 $\begin{vmatrix} I_m & 0 \\ A & I_n \end{vmatrix}=1$ 且 $\begin{vmatrix} -I_m & B \\ 0 & AB \end{vmatrix}=(-1)^m\det C$， 同時不難看出

$$\begin{vmatrix} -I_m & B \\ A & 0 \end{vmatrix}=(-1)^m\sum_S(\det A_S)(\det B_S),$$

於是就得到了結論。 ■

定理 3.19 的證明： 這裡只證明 $s=t$ 的情況， 而一般的結論可以由線性代數中的定理（如果矩陣的所有列向量總和爲零向量， 則每一行當中的各個餘因子都相等）推出來。

假設 M^* 是把 M 的第 s 列去掉後得到的矩陣， 則由引理 3.20， 容易看出有 $Q^*=M^*(M^*)^{\mathsf{T}}$。 假如 $m<n-1$ 的話， 則由 Binet-Cauchy 公式知道 $\det Q^*=0$， 而另一方面， 確實 G 沒有生成樹， 因爲 G 根本就不連通。 因此我們可以假設

$m \geq n-1$。 再次根據 Binet-Cauchy 公式， 可見 $\det Q^* = \sum_B (\det B)^2$， 其中 B 過 M^* 的所有 $n-1$ 階子方陣。 由引理 3.21， $(\det B)^2 = 1$ 若且唯若對應的邊是生成樹， 而 B 又通過所有可能的組合， 因此就計算到了所有的生成樹恰一次。 ■

3.6. 最小生成樹

一個圖的生成樹無論是點數或邊數都是固定的， 所以在這個觀點下每一個生成樹都是一樣大的； 不過， 在實際的應用當中， 每條邊代表的意義可能會略有不同。 例如在本章一開始的網路架設問題當中， 每條網路線所對應的架設費用就不太一樣， 而目標是要找一個總值加起來最小的連通生成子圖。 這種時候， 很自然地會尋求生成樹， 因爲連通生成子圖當中如果有圈， 就可以去邊得到總費用更小的答案。

一般而言， 假設 G 是一個連通圖， 而且每條邊 e 上面都有賦予一個**權重**（weight）$w(e)$， 而希望求一個 G 的生成樹 T 使得總重 $w(T) = \sum_{e \in E(T)} w(e)$ 最小。 $w(e)$ 通常是一個實數， 有的時候也會加上非負的條件。 在上面的例子當中， $w(e)$ 就是在 e 上面架設一條線路的成本； 此時恆有 $w(e) \geq 0$， 有可能會等於零（例如某兩個工作站之間本來就已經有網路線了）。 假如權重有可能是負值的話， 那最小生成樹就不見得會是總重最小的連通生成子圖， 但這邊先不處理這種情況。

如何求最小生成樹的問題有很多種解法， 其中最爲人所知的就是 Kruskal 的**貪求法**[5]（greedy algorithm）。 這個方法是從 G 的無邊生成子圖開始， 將 G 中的邊依權重由小到大逐一嘗試加入子圖， 如果加入之後不會產生圈就將邊加入， 否則就跳過， 繼續看下一條邊。 寫成演算法的話， 會是下面的型式：

演算法 3.23.（Kruskal 的貪求法 [13]）

輸入：有 m 邊的連通圖 $G = (V, E)$， 每一條邊 e 有非負權重 $w(e)$。

[5]「貪求法」這個名詞並不是專指這個 Kruskal 的演算法， 這個名稱是用來稱呼一類型的演算法， 其共同特徵是這些演算法都企圖在每一個步驟皆達到對當時來說最好的結果。 舉一個例子來說， 馬拉松比賽的貪求法就相當於是打從一開始就全力衝刺、 並且在每一個時間點都一直試圖維持著自己當時能夠達到的最高跑步速度。 由這個例子就可以了解到， 貪求法不見得能夠很有效地解決問題， 因爲任何稍微有點常識的人都知道在長跑比賽當中應該要適度保留體力， 才能讓整體的成績最好。 雖然貪求法具有這種會因爲著眼於局部而忽略全域性質的弱點， 但是它卻是最直觀的一類演算法， 而且有許多問題都是只要用貪求法就可以簡單地解決（包括這邊的最小生成樹問題在內）。 在本書後面的章節也會有其他應用在別的問題上的貪求法的例子。

輸出：一個最小生成樹 T。

方法：

1. 將邊排序 $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m)$；
2. $T \leftarrow \emptyset$；
3. 讓 i 從 1 到 m { 若 $T + e_i$ 沒有圈則 $T \leftarrow T + e_i$； }

上面的演算法輸出的只有邊， 不過我們可以想像輸出的圖中也包含了所有的點。

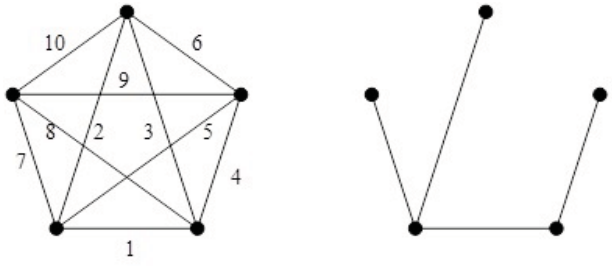


圖 3.11: 五個工作站的網路架設示意圖。

圖 3.11 慮一個有五個工作站的狀況， 在每條邊上架設網路線所需要的經費分別如其左圖所示。 其中最左邊的工作站因爲要穿越河流， 因此通往該工作站架設成本都比較高一些。 依照貪求法， 先將最底下那條成本爲 1 的邊加入， 然後是 2； 但是 3 就不能加入， 因爲加入 3 的話會構成圈， 所以跳過， 接著加入 4； 隨後的 5 跟 6 也都不能加入， 所以加入 7， 在那之後都沒有其他邊可以加入了， 所以演算法最後會輸出右邊的圖。 這確實是生成樹， 而且如果讀者願意， 可以驗證這確實是最小的生成樹。

對於任何演算法， 一方面需要確定其正確性， 另一方面也需要探究其時間複雜度， 這裡先針對第一點討論。 要證明這個演算法是正確的， 需要用到下面這個跟生成樹有關的性質：

性質 3.24. 若 T 跟 T' 都是 G 的生成樹， 則對於任一條邊 $e \in E(T) \backslash E(T')$， 恆存在 $e' \in E(T') \backslash E(T)$ 使得 $T' + e - e'$ 是 G 的生成樹。

證明： 由於 T' 是生成樹， 根據定理 3.7 的等價關係 (5)， $T' + e$ 一定有一個圈 C。 但是 T 沒有圈， 因此 C 中一定有一條邊 e' 是 T 沒有的。 考慮 $T'' = T' + e - e'$ 這個

圖， 根據性質 3.4， T'' 是連通的。 又因爲 T'' 恰有 $n-1$ 條邊， 因此根據定理 3.7 的等價關係 (3)， T'' 是樹， 而且用到了 G 的每一點， 因此是 G 的生成樹。 ■

定理 3.25. *Kruskal* 的貪求法可以找到賦權連通圖的一個最小生成樹。

證明： 首先要確定這個演算法輸出的 T 是一個生成樹。 演算法從頭到尾都沒有加入任何一條使得 T 會有圈的邊， 因此 T 是無圈的。 如果 T 不是 G 的生成樹， 由定理 3.7 的等價關係 (5) 可知， 可以在 T 中加入一條邊 e_i 使得它仍然無圈； 然而， 在貪求法的過程中因此會將 e_i 加入 T 中， 矛盾。 故 T 是 G 的生成樹。

令 T^* 是 G 的某個最小生成樹， 不失一般性可以假設 T^* 在 G 的最小生成樹當中與 T 有最多的共用邊。 假如 $T = T^*$ 則已經證明完畢； 不然， 選取一條下標最小的邊 $e_i \in E(T)\backslash E(T^*)$， 於是根據性質 3.24， 可以選取 $e_j \in E(T^*)\backslash E(T)$ 使得 $T' = T^* + e_i - e_j$ 是 G 的生成樹。 如果 $j < i$， 由於 e_j 以及 T 當中所有下標比 e_i 小的邊都在 T^* 中、 它們不含圈， 所以演算法進行到 e_j 的時候 e_j 會被選到 T 內， 矛盾。 因此 $i < j$ 而有 $w(e_i) \leq w(e_j)$， 於是 $w(T') \leq w(T^*)$， 但這表示 T' 也是一個最小生成樹； 然而 T' 與 T 的共同邊比 T^* 與 T 的共同邊多， 這與 T^* 的選取法矛盾。 ■

接下來要談這個演算法的時間複雜度。 跟一般在討論時間複雜度問題時相同， 想問的是整個演算法的需時跟輸入資料大小之間的關連如何； 而在這裡用的是邊數（即 m）當輸入大小。 基本上， 整個演算法分成兩個部分， 第一個部分是排序， 第二個部分則是不斷地判斷每次加入邊之後有沒有圈產生。 排序的部分在演算法學當中另有深入的探討， 這邊不多說； 例如如果採用**堆積排序法**（heap sort）， 則排序需要的時間會是 $O(m \log m)$。

至於第二個部分， 這邊遇到最主要的問題是， 怎麼判斷一個圖有沒有圈？ 如果有了這樣的判斷方法， 那麼只要連續執行 m 次， 就可以完成第二階段的工作了。 不過， 其實並不需要這樣做。 相當有趣地，「判斷一個圖有沒有圈」這個問題本身其實跟「每次加入一條邊看看有沒有圈出現」是一樣的， 換句話說， 思考方式應該是環繞著後者， 而不是前者。 具體來說， 可以這樣做。 一開始， 在還沒有加入任何邊之前， 整個圖是完全不連通的 n 個點， 於是就先建立 n 個集合來記錄這些連通部分（即每個集合裡各放一個點）。 每當加入一條邊之後， 就把這條邊兩端連接的連通部分之點集做聯集， 以維持記錄連通部分的長相。 在這個想法之下， 當準備要加入一條邊時， 要是發現它的

兩端點是屬於同一個連通部分， 那就意味著這條邊的加入必然會造成圈了。 於是， 在這種概念之下， 第 2 章提到的聯集尋找問題恰好可以派上用場。 也就是， 演算法的第 3 行可以換成：

3'. 一開始 V 的每一點 v 形成一個集合 $\{v\}$；

4'. 讓 i 從 1 到 m

5'. { 令 $e_i = uv$； $s \leftarrow \text{find}(u)$； $t \leftarrow \text{find}(v)$；

6'. 若 $(s \neq t)$ 則 $T \leftarrow T + e_i$ 並執行 $\text{union}(s,t)$； }

在執行 find 和 union 的操作時， 比較簡單的方法需時 $O(m \log m)$， 高速互斥集聯集法就更快了、 幾乎接近線性時間。 這樣就得到：

定理 3.26. *Kruskal* 的貪求法所需時間爲 $O(m \log m)$。

最小生成樹問題另一個有名的演算法是 Prim 演算法， 它以下面方法產生一邊賦權連通圖 G 的最小生成樹： 最初先選定一點 u_0， 令 $S = \{u_0\}$、 $T = \emptyset$ 並逐漸擴大直到 $S = V(G)$ 爲止； 擴大方法是， 每次由 S 和 $\overline{S}$ 之間找出一條權重最小的邊 uv、 其中 $u \in S$ 而 $v \in \overline{S}$， 將 v 加入 S 並將 uv 加入 T。

演算法 3.27. （Prim）

輸入：有 n 點的連通圖 $G = (V, E)$， 每一條邊 e 有非負權重 $w(e)$。

輸出：一個最小生成樹 T。

方法：

1. 任取一點 u_0 並令 $S \leftarrow \{u_0\}$；

2. $T \leftarrow \emptyset$；

3. 讓 i 從 1 到 $n-1$ { 選取 $u \in S$ 而 $v \in \overline{S}$ 使 $w(uv) = \min_{x \in S, y \in \overline{S}} w(xy)$；

4. $S \leftarrow S \cup \{v\}$； $T \leftarrow T + uv$； }

這個演算法的正確性請見習題 3.22。 至於其時間複雜度， 在第 3 行中如果求 $\min_{x \in S, y \in \overline{S}} w(xy)$ 以 $O(n^2)$ 估計， 就會得到總時間爲 $O(n^3)$。 其實計算 $\min_{x \in S, y \in \overline{S}} w(xy)$ 可以有如下 $O(n)$ 的方法， 這樣就會得到總時間爲 $O(n^2)$。 爲達到這個目的， 對每一

點 $y \in \overline{S}$ 要保存一個量 $\mathrm{M}[y]$ 表示 $\min_{x\in S} w(xy)$、 其中 $w(xx) = 0$、而當 $x \neq y$ 且邊 xy 不存在時設 $w(xy) = \infty$， 同時用 $\mathrm{P}[y]$ 表是達到最小值的 x。 此時第 3、4 兩行可以修改爲：

3'. 對所有 V 的點 y { $\mathrm{M}[y] \leftarrow w(u_0y)$； $\mathrm{P}[y] \leftarrow u_0$； }

4'. 讓 i 從 1 到 $n-1$

5'. { 選取 $v \in \overline{S}$ 使 $\mathrm{M}[v] = \min_{y\in\overline{S}} \mathrm{M}[y]$； $u \leftarrow \mathrm{P}[v]$；

6'. $S \leftarrow S \cup \{v\}$； $T \leftarrow T + uv$；

7'. 對所有 $y \in \overline{S}$ { 若 $w(uy) < \mathrm{M}[y]$ 則令 $\mathrm{M}[y] \leftarrow w(uy)$ 及 $\mathrm{P}[y] \leftarrow u$； }

將 Kruskal 和 Prim 的方法綜合起來， 可以推廣成如下的演算法： 從 $\emptyset$ 開始逐一加一條邊到 T 中， 加邊的方法是， 隨便找 T 的兩個連通部分、並找它們之間權重最小的一條邊、加入 T 中。 這個方法的正確性請參見習題 3.24。

Prim 方法跟**最短路徑問題**（the shortest path problem）的解答很像。 在這個問題裡， 一樣有一個連通圖 $G = (V, E)$、並且每一邊 e 都有一個非負實數 $w(e)$。 圖中任兩點 u 和 v 的 **w-距離**（w-distance）$d_w(u, v)$ 是

$$w(Q) := \sum_{e\in E(Q)} w(e)$$

對所有可能的 u-v 路徑 Q 的最小值。 當所有邊權重 $w(e)$ 都是正的時候， d_w 是一個度量。 當所有邊權重 $w(e)$ 都是 1 的時候， d_w 就是 d。

下面是著名的 Dijkstra 演算法。 Dijkstra 的方法和 Prim 方法除了 $\mathrm{M}[y]$ 之外，其他都一樣。 在 Prim 方法裡， $\overline{S}$ 中一點 y 的 $\mathrm{M}[y]$ 表示 y 與 S 中一點 x 的最小權重 $w(xy)$； 而在 Kruskal 方法裡， $\overline{S}$ 中一點 y 的 $\mathrm{M}[y]$ 表示 u_0-y 路徑 Q、 滿足 $V(Q) - \{y\} \subseteq S$、的最小權重總和 $\sum_{e\in E(Q)} w(e)$。

演算法 3.28.（Dijkstra）

輸入：有 n 點的連通圖 $G = (V, E)$ 及其中一點 u_0， 每一條邊 e 有非負權重 $w(e)$。

輸出：點 u_0 到每一點 y 的 w-距離 $d_w(u_0, y)$。

方法：

1. 令 $S \leftarrow \{u_0\}$；
2. 對所有 V 的點 y { $\mathrm{M}[y] \leftarrow w(u_0y)$； $\mathrm{P}[y] \leftarrow u_0$； }
3. 讓 i 從 1 到 $n-1$
4. { 選取 $v \in \overline{S}$ 使 $\mathrm{M}[v] = \min_{y\in\overline{S}} \mathrm{M}[y]$； $u \leftarrow \mathrm{P}[v]$；
5. $S \leftarrow S \cup \{v\}$；
6. 對所有 $y \in \overline{S}$ { 若 $\mathrm{M}[u] + w(uy) < \mathrm{M}[y]$ 則
7. 令 $\mathrm{M}[y] \leftarrow \mathrm{M}[u] + w(uy)$ 及 $\mathrm{P}[y] \leftarrow u$； }
   }
8. 對所有 V 的點 y 令 $d_w(u_0, y) \leftarrow \mathrm{M}[y]$；

Dijkstra 的方法和 Prim 方法除了 $\mathrm{M}[y]$ 的改變不同之外， 其他都一樣， 所以時間複雜度也是 (n^2)。 其正確性請見習題 3.26。

3.7. 習題

3.1. 在已知定理 3.7 的 (1) 至 (4) 這 4 個敍樹等價的條件下， 試證明 (5) 與 (6) 和其他 4 個敍述等價。

3.2. 證明每一顆樹 T 至少有 $\Delta(T)$ 片葉。 什麼樣的樹會恰有 $\Delta(T)$ 片葉？

3.3. 若圖 G 有 $n \geq 3$ 點， 且從 G 中去掉任一點均成樹， 試求 G 的邊數， 並藉此求 G。

3.4. 若樹 T 中任一與葉相鄰的點其度至少爲 3， 證明 T 中至少存在兩片葉有共同鄰居。

3.5. 試證當 $n \geq 2$ 時正整數序列 $d_1, d_2, \ldots, d_n$ 是某棵樹的度序列之充分必要條件爲 $\sum_{i=1}^{n} d_i = 2n-2$。

3.6. 證明推論 3.12。

3.7. (a) 證明對於任意連通圖 G 恆有 $\mathrm{rad}(G) \leq \mathrm{diam}(G) \leq 2\mathrm{rad}(G)$。

(b) 若非負整數 r 和 d 滿足 $r \leq d \leq 2r$， 試求一圖使得其半徑爲 r、直徑爲 d。

3.8. 證明若 $\mathrm{rad}(G) \geq 3$， 則 $\mathrm{rad}(\overline{G}) \leq 2$。

3.9. 證明定理 3.13。

3.10. 證明定理 3.14。

3.11. 設 G 是有 n 點的連通圖， 定義一個新圖 G'， 其點集爲 G 的所有生成樹所成的集合， 而兩生成樹相鄰若且唯若它們在 G 中有 $n-2$ 條共用邊。 證明 G' 是連通圖， 並決定 G' 的直徑。

3.12. 在 DFS 或 BFS 產生的生成樹中， 對每個點 v 定義其**層次**（level）爲它到樹根的距離。 試證明， 在 BFS 當中， 無論是樹邊還是跨越邊， 它所連結的兩點之層次至多差 1。

3.13. 設計出演算法求出一個連通圖中兩點 x 和 y 的距離。

3.14. 設計出演算法判斷一個圖是否爲二分圖。

3.15. 求圖 3.12 中有 16 點 16 邊的圖的生成樹個數。

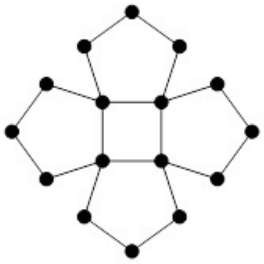


圖 3.12: 有 16 點 16 邊的圖。

3.16. 求 $K_n - e$ 的生成樹個數， 其中 e 是 K_n 的任意一條邊。

3.17. 令 G_n 是如圖 3.13 所示具有 $2n$ 點和 $3n-2$ 邊的圖。 證明當 $n \geq 3$ 時， $\tau(G_n) = 4\tau(G_{n-1}) - \tau(G_{n-2})$。 當 $n \geq 1$ 時，求 $\tau(G_n)$。

圖 3.13: 圖 G_n。

3.18. 令 H_n 表示加一點連到 P_n 的所有點所成的圖， 如圖 3.14 所示。 試證明當 $n \geq 3$ 時， $\tau(H_n) = 3\tau(H_{n-1}) - \tau(H_{n-2})$。 當 $n \geq 1$ 時，求 $\tau(H_n)$。

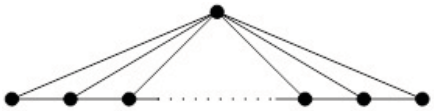


圖 3.14: 圖 H_n。

3.19. 試求 $\tau(K_{m,n})$。

3.20. 若 T 和 T' 是連通圖 G 的兩生成樹， 對任一 $e \in E(T)\backslash E(T')$， 證明存在 $e' \in E(T')\backslash E(T)$ 使得 $T - e + e'$ 和 $T' + e - e'$ 都是 G 的生成樹。

3.21. 假如 G 是一個邊賦權連通圖， 其各邊的權重都相異。 證明 G 恰有一最小生成樹。

3.22. 試證明當 Prim 的演算法結束時， T 是 G 的最小生成樹。

3.23. 對一給定的邊賦權連通圖 G 中的一個生成樹 T， 令 $m(T)$ 表示 T 中的最大權重。 若 x 表示所有生成樹中 $m(T)$ 的最小值， 且 T^* 是一最小生成樹， 試證明 $m(T^*) = x$。 這種滿足 $m(T) = x$ 的生成樹 T 稱爲是 G 的**瓶頸**（bottleneck）生成樹。

3.24. 試證明下面的演算法可以得到 G 的一個最小生成樹： 從 $\emptyset$ 開始逐一加一條邊到 T 中， 加邊的方法是， 隨便找 T 的兩個連通部分、並找它們之間權重最小的一條邊、加入 T 中。

3.25. 試證明，當 w 是正權重時 d_w 是一個度量。

3.26. 試證明 Dijkstra 方法的正確性。

3.8. 參考文獻

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Series in Computer Science and Information Processing, Addison-Wesley, Reading, Mass., 1975.

[2] B. Bollobás, Almost every graph has reconstruction number three, *J. Graph Theory*, vol. 14 (1990), pp. 1–4.

[3] J. A. Bondy, On Ulam's conjecture for separable graphs, *Pacific J. Math.*, vol. 31 (1969), pp. 281–288.

[4] A. Cayley, A theorem on trees, *Quart. J. Math.*, vol. 23 (1889), pp. 376-378.

[5] P. Erdős, Extremal problems in graph theory, in: *Theory of Graphs and its Applications*, (M. Fiedler, eds.), Academic Press, 1965, pp. 29-36.

[6] I. P. Goulden and D. M. Jackson, *Combinatorial Enumeration*, Dover Publications, Inc., Mineola, NY, 2004. pp. 174.

[7] F. Harary, On the reconstruction of a graph from a collection of subgraphs, in: *Theory of Graphs and its Applications*, (*Proc. Sympos. Smolenice*, 1963). Publ. House Czechoslovak Acad. Sci., Prague, 1964, pp. 47–52.

[8] F. Harary, A survey of the reconstruction conjecture, *Graphs and Combinatorics*, *Lecture Notes in Mathematics* 406, Springer, 1974, pp. 18-28.

[9] F. Harary and E. Palmer, On the problem of reconstructing a tournament from sub-tournaments, *Monatsh. Math.*, vol. 71 (1967), pp. 14–23.

[10] C. Jordan, Sur les assemblages de lignes, *J. Reine Angew. Math.*, vol. 70 (1869), pp. 185-190.

[11] P. J. Kelly, A congruence theorem for trees, *Pacific J. Math.*, vol. 7 (1957), pp. 961-968.

[12] W. L. Kocay, A family of nonreconstructible hypergraphs, *J. Combin. Theory Ser. B*, vol. 42 (1987), pp. 46–63.

[13] J. B. Kruskal, Jr., On the shortest spanning subtree of a graph and the traveling salesman problem, *Proc. Amer. Math. Soc.*, vol. 7 (1956), pp. 48-50.

[14] B. D. McKay, Small graphs are reconstructible, *Australian J. Combin.*, vol. 15 (1997), pp. 123–126.

[15] C. St. J. A. Nash-Williams, The Reconstruction Problem, in: *Selected Topics in Graph Theory*, 1978, pp. 205–236.

[16] P. V. O'Neil, Ulam's conjecture and graph reconstructions, *Amer. Math. Monthly*, vol. 77 (1970), pp. 35-43.

[17] H. Prüfer, Neuer beweis eines satzes über permutationen, *Arch. Math. Plys.*, vol. 27 (1918), pp. 742-744.

[18] P. K. Stockmeyer, The falsity of the reconstruction conjecture for tournaments, *J. Graph Theory*, vol. 1 (1977), pp. 19–25.

[19] P. K. Stockmeyer, A census of non-reconstructable digraphs, I: six related families, *J. Combin. Theory Ser. B*, vol. 31 (1981), pp. 232–239.

[20] S. M. Ulam, *A collection of Mathematical Problems*, Wiley, New York, 1960, pp. 29.

[21] M. von Rimscha, Reconstructibility and perfect graphs, *Discrete Math.*, vol. 47 (1983), pp. 283–291.

[22] Y. Yang, The reconstruction conjecture is true if all 2-connected graphs are reconstructible, *J. Graph Theory*, vol. 12 (1988), pp. 237–243.