
貓都學得會的運算思維
作者：孔令傑

貓都學得會的運算思維

（這裡只是想要來隻貓貓）

作者：孔令傑

總編輯：陳庭姍

PLUS 推動大學{程式設計}教學
Programming Learning for University Students

作者簡介

孔令傑

國立臺灣大學資訊管理學士、碩士、美國加州大學柏克萊分校工業工程與作業研究博士，現職國立臺灣大學資訊管理學系副教授。在系上開設「資訊經濟」、「作業研究」、「程式設計」、「資管新生專題」、「資料結構與進階程式設計」等課程之餘，亦自 2016 年起與盧信銘教授一同於管理學院開設「商管程式設計」，推廣非資訊科系之程式設計與資訊科技教育，其完整課程影片自 2018 上半年起於 Coursera 上免費開放，至今有近 20000 人次修習。除此之外，也從 2016 年起經營系上「服務學習」，組織系上學生至國中小進行基礎程式設計教學，足跡遍及新北市、宜蘭縣、新竹市、彰化縣等。自 2018 年起參與教育部「推動大學程式設計教學」計畫，擔任分項共同主持人。曾獲 2019 年臺大優良服務學習導師獎、2018 年臺大校教學傑出獎、2017 年臺大優良導師獎與 2015-2017 年臺大校教學優良獎。

陳庭姍

國立臺灣大學圖書資訊學士，現就讀於國立臺灣大學資訊管理學研究所碩士班一年級。熱愛綠色，擅長手工藝，代表作有手摺雪花、手摺火箭、手做花束、手做活動式卡片。曾獲 2020 年「決策分析研討會」最佳論文獎。

序

接到李蔡彥老師邀請，在教育部「推動大學程式設計」計畫中擔任分項二「共通性需求」的共同主持人，負責「運算思維」這個項目時，心裡非常惶恐，畢竟在我當大學生的年代，「運算思維」這個詞是聽都沒聽過，回臺灣任教後雖然開始接觸到這個概念，但距離能製作教材、開班授課也是非常遙遠。所幸過去幾年來一直都有在臺大管院教授給非資訊科系學生的基礎程式設計，甚至也去國中小做資訊教育服務學習，對這整件事情也算有一點經驗和心得，於是便在許多人的幫助下編纂了教材、錄製了線上課程，最終完成了這本電子書。

在資訊管理系任教，對於資訊科技在企業管理乃至各行各業的應用，當然都抱持著熱情與信念。在現今的資訊社會中，我完全相信每個人都應該學程式設計，也都應該具有運算思維。這當然不是為了把每個人都變成軟體工程師，但如果未來各種進步都能藉助資訊科技驅動、每個組織或個人都能靠著資訊科技提升價值與競爭力，那讓自己能更好地善用資訊科技、更好地與資訊人員溝通，就是值得每個人去做的事了。

在這整個規劃和推廣的過程中，或許我才是收穫最多的人。我閱讀了許多參考資料，在自己心裡建立起關於運算思維的一套理論；我看到許多學生與社會人士建立起對運算思維的認識，以之做為在資訊科技的方法與應用上繼續深入學習的基礎；我也在過程認識了許多前輩、先進與好夥伴。感謝鄭瑋老師、奇宥、昀蓉、芸禎、佩穎在編製教材時的協助；感謝煜柔、怡瑄、楷翔、汶佑幫忙整理文稿；感謝梓妘和逸庭幫這本書畫了很漂亮的封面和插圖；感謝庭姍一口答應擔任這本書的編輯。如果這本電子書有好的地方，都是他們的功勞；如果仍有疏漏與不足之處，都是我的責任。

希望這本書能讓閱讀它的人，增加一點對運算思維的認識！

孔令傑 謹識

2020 年 6 月

目次

第一章

課程概述

1.1 計畫簡介

這門課是運算思維，我們將會簡單地說明、介紹以及指導有關於運算思維的知識。這門課有很多個模組，我們先從課程概述開始，來講解運算思維的定義，以及這門課的架構。

這門課起源於教育部的「推動大學程式設計教學計畫」，計畫辦公室中有各式各樣與程式設計相關的分項團隊，如寫網頁、寫 APP、互動設計、資料分析等，無論是哪一種程式設計教學計畫，團隊都包含了許多教授、專家、業界人士來負責教學。然而，綜合來說，無論大家要寫的是網頁、APP、還是做互動設計、資料分析，其中都有兩件事情是共通的，也是所有學習程式設計的人都必須要學習的，那就是**運算思維（Computational Thinking）**以及**軟體工程（Software Engineering）**。

稍微介紹一下軟體工程，軟體工程是寫軟體的**方法論**，以及如何透過有效率、有規模、有系統的方式將軟體的維護、開發寫得更好。對已經會寫程式、而且希望能用程式設計方法打造一個好軟體的人而言，軟體工程會是十分重要的事情；但對於初學者而言，討論軟體工程仍言之過早。也就是說，在學習軟體工程，或是在學習任何一種程式設計之前，理論上應先建立所謂的「運算思維」較為適當。

1.2 運算思維

即使是對於沒有要往資訊科技業做深造的人，學習程式依然非常有幫助，因為透過學習程式設計，能夠培養**邏輯思考**的能力。程式是由邏輯運算組成，既然如此，寫程式跟思考必然是相關的，這也是運算思維一詞的來源。

實際上，在筆者念大學的時候（約 15 年前），當時並沒有「運算思維」這個詞彙。直到 5、6 年前筆者才開始聽到，約 2、3 年前運算思維的概念才開始慢慢地成形。用最簡單的方式來解釋，運算思維是一種思考的方式，是一種看待事情的方式，是當一件事情、問題出現在你面前時，你如何看待它的方式。

維基百科上有一句關於運算思維的話講得還挺不錯的：

「從教育的角度上來講，運算思維就是一套解決問題的方法，而這一套解決問題的方法主要是表達問題的方式，還有表達解法的方式，使電腦可以理解，並且可以去執行的。」
"In education, computational thinking is a set of problem-solving methods that involve expressing problems and their solutions in ways that a computer could execute."[1]

換句話說，當我們想要解決問題，就必須提出一個解決問題的方法，而既然問題有很多種，解決問題的方法當然也有很多種。對於有運算思維概念的人而言，當這個問題出現在他面前，他會發現電腦可能對解決問題有所幫助，並判斷問題中哪一個部份適合用電腦解決、哪一個部分不適合用電腦解決，這就是運算思維的一個最直觀的定義了。

舉個例子，假設你在某間公司裡頭工作，你的前輩不知為何地非常討厭你，時常在公司開會時對你出言不遜，或是在你的辦公室的座位上撒圖釘。這種情況的確是一種問題，但是你大概不會想要去寫個程式、或是透過資訊科技去解決這種問題。不過，有很多別的問題卻可以用程式去解決，例如老闆要求你每天早上 8 點去搜尋各大新聞網站，找出跟公司有關的所有資訊，全部抓下來並做成文件寄到老闆的信箱。這件事情很像例行公事，以現在的科技來說，並不需要派人每天去這麼做。事實上，你完全可以寫一個程式，就像寫出一個軟體機器人一樣，讓它每天到了固定的時間就去固定的網站、固定地去搜尋你所設定的關鍵字，抓到了資訊之後做一下判別，整理成文件後自動寄到老闆的信箱。這個意義上，運算思維是指有沒有能力去判斷一個題目、任務適不適合讓電腦來解決。

這門課的目的在於培養運算思維，以理解程式設計的邏輯與架構。透過思維邏輯，能夠在完全不需要寫程式的前提之下，就可以去學習程式設計、演算法可以幫忙做哪些事情，也希望能夠為後續的學習奠定良好的基礎。對於一個剛學習程式語言，就直接上機打程式碼的人而言，很容易會遇到環境設定、語言選擇、語法上的困擾，這是件很麻煩的事。這門課中，我們完全不用寫程式，只要動腦思考、培養運算思維，並試著在這過程中理解，生活中原來有這麼多問題是程式設計可以解決的。

這門課的教學內容主要都是日常生活中所用得到的技巧，而這些技巧能夠幫忙我們解決問題。由於電腦能夠幫我們解決的問題，通常都會有個固

[1] 維基百科: 運算思維（英）https://en.wikipedia.org/wiki/Computational_thinking

定的類型與模式，因此，當你在日常生活中遇到一個困難時，首先必須要清楚地**定義問題**。例如老闆要求你每天早上 8 點的時將搜尋進來的文章做成文件，再寄給老闆，在這個情境中，你所要解決的問題究竟是收集文章、是判定文章符不符合老闆的需求、還是把文章做成文件時還要排版排得漂亮、又或者是以上都要。在開始找出解決問題的方法之前，你必須先試著定義現階段要解決的是什麼問題。

接下來，就要去**抽象化（abstraction）**這個問題，也就是把所定義出的這個真實世界的問題描述成電腦可以理解的樣貌，讓電腦知道目標的問題是什麼，畢竟就本質上而言，電腦不過就是一大堆 0 跟 1 去進行運算的機器。讓電腦去了解問題的方法就是進行抽象化，而這個行為也被稱為**建構模型**。在問題抽象化完成、模型已經被建構出來了之後，接著就要去建構解決方案。

在電腦的世界之中，解決方案被稱為**演算法（algorithm）**，演算法會告訴電腦如何一個步驟、一個步驟地做，直到完成目標。也就是說，解決問題過程中做事情的方法，就是所謂的演算法。在設計演算法時，經常會有一些共同性的策略，如**拆解（decomposition）**與**轉化（reduction）**。**拆解**顧名思義就是把一個大問題拆成很多個小問題，若遇到難以處理的大問題，透過解決小問題並將結果拼湊起來，就可以找到答案；**轉化**則是將一個不會處理的新問題轉成一個等價的、會處理的舊問題，藉此找到答案。

因此，抽象化、演算法、拆解、轉化將會是這門課裡最主要的 4 個單元。雖然不敢說將這 4 個單元合併起來就能涵蓋運算思維的全部，畢竟運算思維確實有很多種不同的定義，但是，這門課程基本上可以提供一個「如何看待問題的方法」，幫助大家判定這個問題電腦能不能懂、是否可交由電腦處理，若可以，則試著抽象化問題，並設計一個演算法，讓電腦能夠透過執行程式得到解答；而拆解和轉化方面的思維能幫助你進一步的思考，所面臨的問題能夠被電腦解決的可能性。若到了未來真的要學習寫程式時，心裡有一些運算思維的概念，思考起來將會順暢許多。

1.3 教材大綱

後續的課程將會分為抽象化、演算法、拆解及轉化等 4 個模組，在解決

問題的時候，抽象化、演算法和拆解這個順序是必要的，必須要先能夠理解前面的概念，後面的概念才能夠跟著理解。而拆解跟轉化 2 個模組相互獨立，其中轉化的概念相對來說較為困難，因此，這門課的核心將會是抽象化、演算法和拆解。當然，若可以的話，將轉化的概念融會貫通一定也會有所幫助。

課程解說過程中，筆者會盡可能地用例子來講解這一些概念，這些例子未必十分有趣，但是從例子出發總是比直接從概念開始要更好理解一點。舉例來說，這些例子可能包含如何在地圖上移動、達成送快遞的需求，或是如何去登山、拉自來水管、查字典、搭公車等各式各樣的問題。這些問題當然有共通的類型，不過只要能夠學習到這些問題的共通精神，舉一反三應該不會很困難，同時也能夠了解一些資訊科學中的知識點。學習程式或多或少就是想要踏進資訊科技的大門，若能對一些重要的基本知識融會貫通，如**圖（graph）**、**網路（network）**、**背包問題（Knapsack Problem）**、**旅行銷售員問題（Traveling Salesman Problem）**、**二元搜尋法（Binary Search）**等，未來在學習正式的程式設計課程時，這些知識都可以派上用場。

主題	實例	資訊科學中的知識點
抽象化 Abstraction	如何走過七座橋？	Graphs and Networks
	快遞員的煩惱	Traveling Salesperson Problem (TSP)
	帶什麼去登山？	Knapsack Problem
演算法 Algorithms	自來水管怎麼拉	Kruskal's Algorithm
	快遞員的煩惱	Greedy Algorithm
拆解 Decomposition	查字典	Binary Search
	快遞員的煩惱：何時搭公車	Modularization and Function
轉化 Reduction	博物館參觀攻略	Hamiltonian Circuit ➔ TSP
	不患寡而患不均	Partition ➔ Knapsack

表 1-1　課綱（主題、實例與知識點）

在課程的時間分配上，若純粹看影片、聆聽筆者講解課程，一個模組大約要花費 30 到 50 分鐘完成。另外，這門課也提供了大量的練習題及學習資源，有興趣的老師和同學們可多加參考。

第二章

抽象化

2.1 七橋問題

這一個單元，我們來談談運算思維裡的**抽象化**。首先，就從下面這個真實歷史事件作為例子：

*東普魯士柯尼斯堡（今日俄羅斯加里寧格勒）市區跨普列戈利亞河兩岸，河中心有兩個小島。小島與河的兩岸有七條橋連接。在**所有橋都只能走一遍**的前提下，如何才能把這個地方**所有的橋都走遍**？*[2]

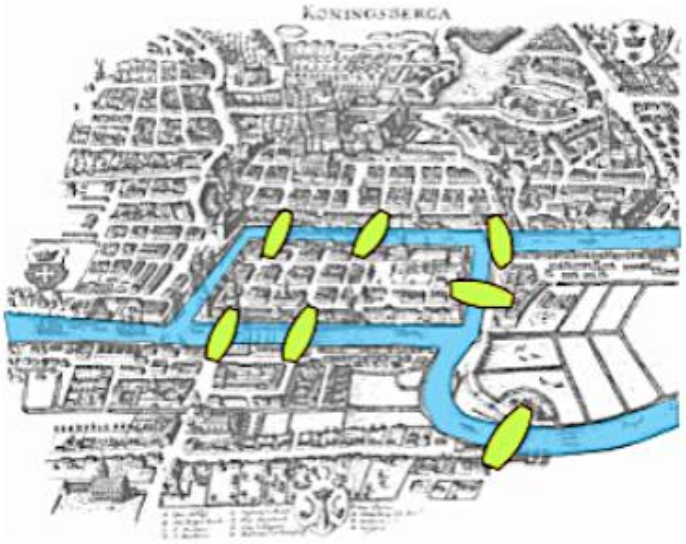


圖 2-1　柯尼斯堡七橋問題示意圖[3]

現在請大家試試看，是否能一筆畫走完所有的橋，並且回到原點？

大家嘗試完後會發現，要一筆畫走完所有的橋，恐怕有點困難。這樣的問題，在某些情況下看似普通，某些情況下又很常見。以圖 2-2 與圖 2-3 為例，請大家試試看，是否能一筆畫走完圖 2-2 中所有的橋，並且回到原點，以及是否能一筆畫走完圖 2-3 中所有的橋，並且也回到原點呢？

[2] 維基百科：七橋問題（英）https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg
七橋問題（中）https://zh.wikipedia.org/zh-tw/柯尼斯堡七橋問題

[3] "Konigsberg bridges" by bogdan is licensed under CC BY-SA 3.0

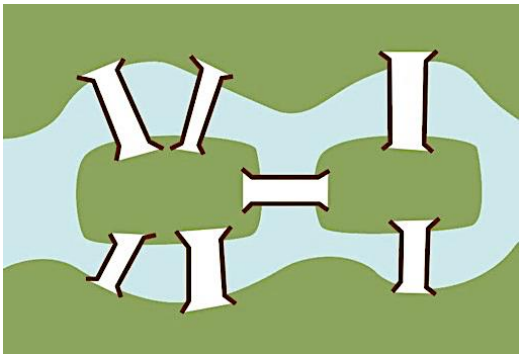


圖 2-2　七橋問題圖形版

圖 2-3　七橋問題地圖版[4]

分別走完後會發現，圖 2-2 與圖 2-3 其實是在解同一個問題，並且都無法一筆畫走完各自的 7 座橋。換句話說，七橋問題若放在不同的圖、地圖、城市，即使道路、街景不同，但本質上有可能是在解同一問題。

此外我們也知道，世界上不一定每個城市都是 7 座橋，同樣要一筆畫走完所有的橋，有可能是像圖 2-4 為六橋問題，或是圖 2-5 為三橋問題。這些不一定是 7 座橋的問題，本質上類似，但又不完全一樣。

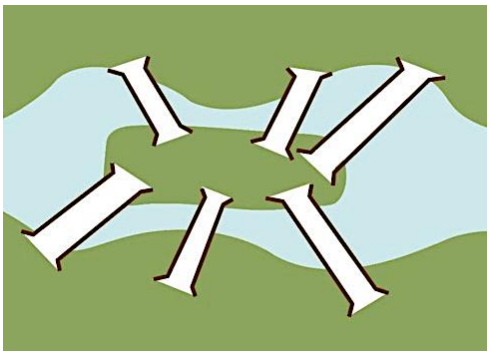


圖 2-4　六橋問題圖

圖 2-5　三橋問題圖

那麼若給定一張地圖，要怎麼解決這類型問題？你可能會像上述一樣，窮舉出所有可能性，來證明是否能一筆畫走完所有的橋。但若這張地圖有 20 座橋、30 個小島呢？或是此時有 1000 張地圖等待你解決呢？沒有找出規則的窮舉，就不是個好的解決辦法。

同樣是解決問題，每個人的做法都可能不太一樣，有的人會在每次有問題出現時，就想辦法解決眼前的問題；但有的人卻會在解決許多問題後，從中找出相關性或是共通點。若我們能針對共通的地方設計一套解法，就可以一次解決很多

[4] “Königsberg 1651” by Merian-Erbe is under Public Domain

問題。換句話說，在程式的世界、電腦科學的世界或是運算思維的世界中，我們真正想找的不是眼前問題的「解」，而是一個問題的「解法」。「解法」就是一套方法、規則，只要將解法套入類型相同的問題，就會得到合理的答案。

要找**規則（rule）**有個先決條件，必須先將實際問題**抽象化（abstraction）**。抽象化的動作是將實際問題用**模型（model）**表示，通常是**數學模型**，將與問題本質無關的細節略去，只留下關鍵元素。以七橋問題為例，在經過抽象化後會發現，真實世界中許多城市、地圖，其實都在描述同一個問題。屆時，若我們針對這些問題的本質設計一套解法，此解法就可以被套用在本質相同但其他細節不同的問題上，我們稱這樣的解法具有**一般性**，換句話說，你完成了**一般化（generalization）**的解法，同時，你也比做不到抽象化的人更會解決問題了。

2.2 七橋問題抽象化

以圖 2-3 七橋問題地圖版為例，讓我們來練習如何將問題抽象化。

進行抽象化前，我們須先找出關鍵元素與重要資訊，關鍵元素包含河與 7 座橋，其中橋的位置、河的位置、島與島之間如何連結，是此問題的重要資訊。下一步，我們過濾掉與問題本質無關的細節，例如河的寬度、橋的材質等這些不影響題目的資訊。在定義出關鍵元素與資訊後，即可進行抽象化。

首先，將河與橋從地圖中標示出來。以圖 2-6 中第 1 張圖為例，藍色標註的為河、綠色標註的為橋。

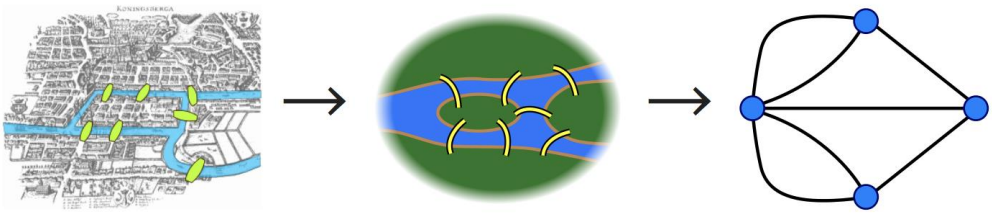


圖 2-6　抽象化過程

接著，將此地圖轉換為圖 2-6 中第二張圖，略去與問題本質無關的細節，使得地圖簡化後，本質仍不變，沒有損失任何我們解決此問題時所需要的資訊。我們將原先地圖上的城堡、農田等統一畫成綠色區塊、統一橋與河的寬度樣式，且維持橋、河、區域間結構一致，例如：被河分開的不同區域之間，橋連結的數量，必須與圖 2-6 中第 1 張圖一致。

最終，我們將圖 2-6 中第 2 張圖，簡化為平面上的點與線組合，也就是圖 2-6 中第 3 張圖。其中每一座橋簡化為一條線，橋所連接的地區簡化為點。最終的模型是一個**圖**或**網路**，這個模型是由**點（vertex、node）**和**線（edge、link、arc）**組成。

因此在解問題時，只需要對抽象化後的模型進行研究即可，從模型中研究得出的答案，即是問題在地圖上的解答。

數學家**尤拉（Euler）**在 1735 年論述，圖 2-6 **不存在**一筆劃且不重複地走完所有的邊，並回到原點的解。因爲若從某點出發後，最後再回到這點，則這一點上的**連結數（degree）**必須是偶數，這樣的點稱為偶頂點。相對的，連有奇數條線的點稱為奇頂點。

由於抽象化的特性，我們透過定義每一點上的連結數，可相較於看一般地圖更清楚地得到結論。假設我們從圖中其中一點 A 出發，一筆畫走完所有的邊，並經過所有的點後可以回到 A，則從圖中其他任一點出發，同樣可以一筆畫走完所有的邊，並回到原點。因此在七橋問題中，圖中每一點都必須為偶頂點，若有任一點是奇頂點，則無法在此圖上找到能夠一筆畫走完所有邊，並回到原點的解答。由於圖 2-6 中存在 **4 個**奇頂點，所以必然**無解**。

圖 2-7　尤拉肖像畫[5]

尤拉也宣稱所有的這種圖都可以被抽象化，請各位嘗試將圖 2-4 六橋問題與圖 2-5 三橋問題抽象化成圖，計算每一點的連結數來判斷是否能一筆畫走完所有

[5] “Königsberg graph”by Chris-martin are licensed under CC BY-SA 3.0; Leonhard Euler is under public domain

邊，並回到原點的可能。

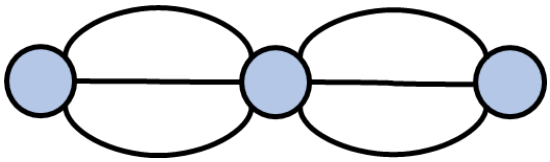


圖 2-8 抽象化後的六橋問題

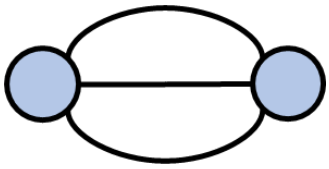


圖 2-9 抽象化後的三橋問題

圖 2-8 與圖 2-9 即為抽象化後的圖。圖 2-8 各點的連結數分別為 3、6、3，圖 2-9 各點的連結數分別為 3、3。由於在兩張圖上都有奇頂點的存在，這裡的六橋問題與三橋問題皆無法一筆畫走完全部的邊，並回到原點。

最後，我們總結上述的發現，可定義出七橋問題的一般規則（general rule）。若一個圖有 **0 個奇頂點**，則有一筆畫經過所有邊且回到原點的解，稱為**尤拉迴路（Eularian Circuit）**；若一個圖有 **2 個奇頂點**，則有一筆畫經過所有邊的解，稱為**尤拉路徑（Eularian Path）**；若為其他狀況，則無解[6]。

上述範例讓各位了解抽象化的進程與意義。**抽象化**是為了**一般化**，找出一般規則的解法。抽象化後可以讓我們更專注在題目的重點，在設計解法時，不受到非問題本質的細節影響，讓我們的解法更具一般性，可以在更多本質相同的問題中適用。

2.3 旅行銷售員問題

為了讓各位更了解抽象化，我們來看抽象化的第 2 個範例。

請大家先憑直覺想像一下這個問題，假設你是一個業務員、銷售員或是推銷員，公司駐點在某一個地方。今日你需要在外跑業務，總共有 12 個客人需要拜訪，所以你需要前往這 12 個位於同一城市的客戶所在地，最後回到公司。簡而言之，你需要在繞完一圈目的地之後回到原點。跑業務的過程中，你的目標是**總移動距離越短越好**，盡量減少通勤的時間，才能花更多時間在與客戶的業務，或是可以早點下班，這就是一個**旅行銷售員問題**。

筆者在臺灣大學服務，請大家試想，誰會想要用最短路徑，拜訪數個臺大校

[6] 現在我們已知偶數個奇頂點要如何判斷，請大家試想若有奇數個奇頂點時，是否能一筆畫走完並回到原點呢？

園內的地點，最後再回到起點呢？在筆者教過的課程裡，曾接收到各式各樣的答案。例如：在管理學院打工的同學，需要送公文到文學院、教務處、電機系等，此同學想要找到最短路徑，繞一圈回到原點；或是同學今天從宿舍出發，要去 5 個地點上課，上課完後要回到宿舍。事實上，這些人都想要用最短路徑完成他們的任務，但是任務的頻率不高，或是移動距離相對短。針對這個問題，最在意的人，其實是校內負責拖吊違規、廢棄腳踏車的職員。

圖 2-10　臺灣大學校內拖吊腳踏車[7]

在此簡單描述臺灣大學校內腳踏車拖吊事宜。腳踏車拖吊職員有據點，專門放置廢棄腳踏車或被拖吊的腳踏車，此據點位於圖 2-11 臺灣大學地圖中的左下角。而他們平日的任務即是開車前往校內各個經常違停腳踏車的地點，拖吊違規的腳踏車，最後回到據點放置被拖吊的腳踏車們。

將上述問題簡化成旅行銷售員問題，可能並未考量到現實中的其他狀況，例如：卡車上腳踏車乘載的數量有限，一趟不一定能載完所有的腳踏車。若將卡車容量一同考量的話，會變成一個更複雜的旅行銷售員問題，每一次腳踏車拖吊直員每次出勤不一定能一次跑完所有點，因此還要決定地點怎麼分著跑，以便總共的移動距離可以最少。

[7] 圖片取自：臺大資管系 103-2 學期「作業研究」期末報告：廖得凱、殷豪、李尚恩、陳昀靖、林翰伸、簡敏瑜

圖 2-11　腳踏車拖吊路線圖[8]

目前我們還沒有要教各位解這類複雜的旅行銷售員問題，我們要理解的事情是，今天如果要解這個路線選擇問題的話，會選擇圖 2-12 這張涵蓋許多資訊、但也增加許多非問題本質的細節，還是選擇僅涵蓋求解問題所需要之重要資訊的圖 2-13？

圖 2-12　國立臺灣大學地圖[9]

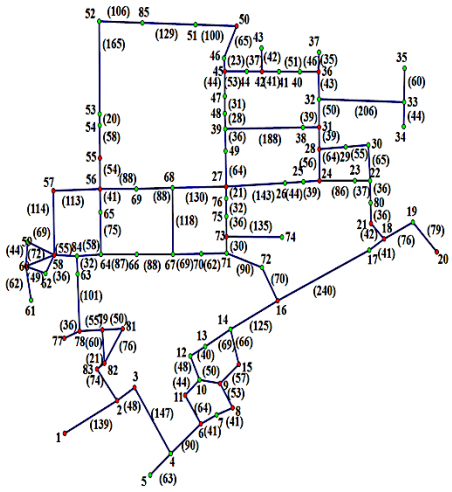


圖 2-13　抽象化後的臺灣大學地圖[10]

學過抽象化的各位可以知道，在解腳踏車拖吊任務的旅行銷售員問題時，你真正關心的只有點與點之間的距離、每個點各自會有多少輛腳踏車會違停在那、以及哪些點之間相通。因此你會從距離或路線的角度來進行抽象化。相對的，建

[8] 圖片取自：臺大資管系 103-2 學期「作業研究」期末報告：廖得凱、殷豪、李尚恩、陳昀靖、林翰伸、簡敏瑜

[9] 圖片取自：https://www.ntu.edu.tw/about/map/C_02.jpg

[10] 圖片取自：臺大資管系 103-2 學期「作業研究」期末報告：廖得凱、殷豪、李尚恩、陳昀靖、林翰伸、簡敏瑜

築物高低、大小等資訊就屬於非問題本質的細節，在抽象化時應予以略去。

應用到真實解決問題，也是一樣的，同一時刻你可能會有非常龐雜甚至混亂的資訊，可能九成都是用不到的。當把這些資訊過濾以後，能讓你專注在真正有用的資訊上，從中再去設計解法，才是正確的做法。因此，我們寫程式時會讓大家做大量的類似練習，即使問題給你很多資訊，但你真正需要的其實只有其中一部分，經過抽象化以後，剩下的問題就會容易許多。

2.4 背包問題

抽象化的最後一個問題是**背包問題**。試想一個情境，你正要去登山，打算在背包裡裝一些有用的物品，像是指南針、柴刀、火柴等，但背包最多只能再裝 5 公斤。每一個物品都有重量，像是指南針的重量為 0.5 公斤、柴刀 1.5 公斤等；每一個物品也都有一個價值，這個價值未必是指該物品的售價或是有用的程度，而是指把該物品帶上山會對你所帶來的效益，如表 2-1 所示。舉例來說，像是拉拉熊雖然在這次的登山活動之中看起來很沒有價值，但是帶拉拉熊還是會對你帶來效益，它可以撫慰你的心靈、或是緊急時刻可以拿來燒，雖然效益很少，但是還是能夠帶來些許價值的。如表 2-1 所示。

編號	1	2	3	4	5	6	7
東西	指南針	柴刀	火柴	塑膠布	望遠鏡	鋼瓶	拉拉熊
重量	0.5	1.5	0.4	1	1.1	1.6	0.8
價值	6	5	4	4	3	4	1

表 2-1　背包問題的範例

背包問題有一個特色，就是每一個東西只能選擇帶或不帶，不能部份攜帶，因為大部分的物品，如指南針，只帶一部份是沒有意義的。那麼在上述限制之下，該選擇攜帶哪些東西，才可以在**不超過重量限制**的情況之下，**最大化背包裡面物品的總價值**？

這個類型的問題就被稱作背包問題。或許在看到這個題目的同時，你會開始去思考哪些東西應該要放：指南針感覺應該很有用、拉拉熊看起來就沒有必要……。然而，身為一個專業的程式設計師、軟體工程師，或是有運算思維的人，在思考的時候應該要多想一點、思路應該要有邏輯一些，畢竟，

你所思考的應該要是能夠解所有這類型的問題的方法，而不只是去思考解這一個題目的方法。你想出的這個**演算法**，應該要能夠用來解有 7 個物品的題目，也能夠用來解有 70 個物品的題目，甚至可以用來解有 700 個物品的題目，即使這類題目被擴充、放大、延伸了以後，也一樣能夠處理的演算法。在思考問題時，**不要解一個問題，要解一類問題；不要找一個解，要找一套解法**，如此才能夠精進自己思考、解決問題的能力。

在此先開始討論**抽象化**的部份，這個題目就資訊量而言，有 4 種資訊：編號、物品（名稱）、重量、價值，而能夠在抽象化的過程之中被過濾掉的資訊，是物品(名稱)這一項。雖然說物品是這一個題目做決策的物體本質，但是真正的模型，只注重於物品被量化出的重量與價值，並不在乎物品的本質。

首先，每一個物品都要為它決定一個**決策變數（Decision variable）**，稱為 x_i。x_i 只會是 1 或是 0，如果 x_i 為 1 的話，就表示決定要帶物品 i，如果 x_i 為 0 的話，就表示決定不要帶物品 i。在定義好決策變數的情況下，便可以做出為上述背包問題的**模型**，如圖 2-2。

$$
\begin{aligned}
\max \quad & 6x_1 + 5x_2 + 4x_3 + 4x_4 + 3x_5 + 4x_6 + x_7 \\
\text{s.t.} \quad & 0.5x_1 + 1.5x_2 + 0.4x_3 + x_4 + 1.1x_5 + 1.6x_6 + 0.8x_7 \leq 5 \\
& x_i \in \{0,1\} \quad \forall i = 1, ..., 7
\end{aligned}
$$

圖 2-2　背包問題的模型

圖 2-2 中第二行稱做**限制式（Constraint）**，表示帶或不帶物品 i 會對背包重量產生的影響。如果 x_1 為 1，其餘為 0 的話，就代表指南針要帶，而背包的重量，也就是限制式的左側就會變成 0.5；如果 x_2 也為 1 的話，就表示柴刀也要帶，而背包的重量，也就是限制式的左側就會變成 $0.5 + 1.5 = 2$，依此類推。根據每個東西要帶或不帶，而改變決策變數了之後，這個限制式可以表示出背包重量不能超過 5 公斤的這個限制。

在這個限制條件下，我們便要去決定目標。圖 2-2 中第一行稱做**目標式（objective function）**，根據每個東西要帶或不帶，而會產生出不同的**目標值（objective value）**，例如有帶指南針可以加幾分、有帶柴刀可以加幾分、帶拉拉熊可以加幾分等。第三行表示對於決策變數的限制，例如 x_i 只會是 1

或是 0 等。將這三行合在一起，就會是我們先前所提到的背包問題抽象模型，而東西的名稱在這個模型之中顯然並不重要。

對於上述問題，模型可以再進行更進一步的抽象化，希望這一個模型能夠面對被擴充、被放大、或被延伸了的背包問題。於是我們更進一步地將物品重量、物品價值、物品總數以及背包最大載重也給抽象化，將物品 i 的重量以 w_i 來取代，將物品 i 的價值以 v_i 來取代，將背包的載重以 B 來取代，物品總數 n 來取代，如圖 2-3 所示。這是一個更加抽象的模型，裡面基本上連數字都沒有，但是卻還是可以描述背包問題的一個模型。如果這類型的問題可以被設計出一個演算法而被解決的話，就表示所有的背包問題都能夠被用相同的演算法所解決。

$$\begin{aligned} \max \quad & v_1x_1 + v_2x_2 + \cdots + v_nx_n \\ \text{s.t.} \quad & w_1x_1 + w_2x_2 + \cdots + w_nx_n \leq B \\ & x_i \in \{0,1\} \quad \forall i = 1, ..., n \end{aligned}$$

圖 2-3　背包問題的通用模型

當然，上述問題並不僅能拿來描述單純地把東西塞進背包的這個問題，它也可以被拿來描述公司內部的生產抉擇。將訂單 i 的處理時間以 w_i 來取代，將訂單 i 的毛利以 v_i 來取代，將工廠的產能以 B 來取代，訂單總數 n 來取代，就可以用來模擬一個公司要在符合工廠產能的情況之下，選擇不同的訂單進行生產，以最大化生產訂單所帶來的毛利之情境。

同樣的，將新創團隊 i 的待投資金額以 w_i 來取代，將新創團隊 i 的預期報酬以 v_i 來取代，將投資的預算以 B 來取代，新創團隊總數以 n 來取代，就可以用來模擬一個公司在預算有限的情況之下，選擇不同的新創團隊去進行投資，以最大化投資新創團隊所帶來的預期獲利之情境。

其實無論是一開始的登山問題、公司的生產問題，還是新創團隊的投資問題，將這些問題全部都抽象化過後，其實都是十分類似的背包問題。而如果背包問題是能被某個演算法所解出來的，就表示上述的所有問題都是能被解出來的，前提就是要能夠看得出這些問題都是相同類型的問題。從解這一類型的問題之中，能夠學到不少技巧，可以去解決不少同類型的問題。

在抽象化單元我們所得到的結論是，無論面對任何的題目，都要試著把它修改成電腦所能理解的樣子。**抽象化是為了做一般化、濾掉不必要的資訊**，以方便進行下一步的運算。

第三章

演算法

3.1 鋪設自來水管

在前一章介紹了何謂**抽象化**後，我們對於找出問題的核心有了一定程度的瞭解，學習到如何將一個問題歸類成一類問題，並將目標從找出一個問題的解轉變為找出問題的**解法**。本章的目標即在於介紹找出解法的關鍵——**演算法**，以下將舉例介紹本章主題，以利讀者理解。

假設有一個城市，城市裡有多戶人家，各戶人家之間有許多道路連接，如圖3-1。這個城市原本是沒有自來水的，現在政府決定興建自來水管線，讓每戶人家都有水可使用。

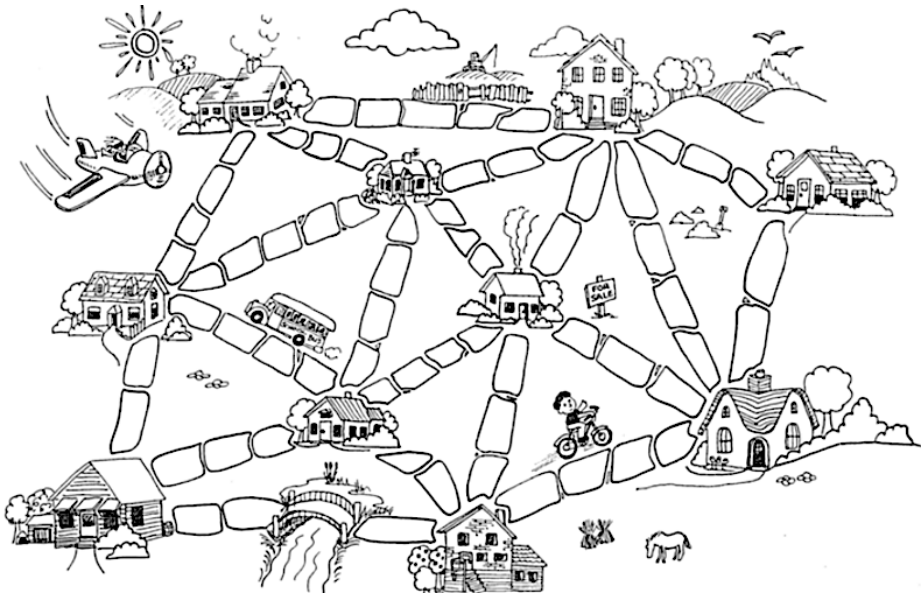


圖 3-1　城市示意圖[11]

然而，興建自來水管線所費不貲，若將每戶人家之間連接的每條道路底下都鋪設自來水管線，鋪設成本將會過高，有浪費公帑之嫌，因此政府決定從圖中挑一些路段來興建自來水管，只要能讓任意兩戶之間有被直接或間接地連接到即可，如此一來自來水廠只要連接任何一戶人家，其他戶人家都能享受到自來水資源。

如今，這座城市的政府希望能將這個任務交給你，要如何**花費最少的成本**，讓**任兩點之間**直接或間接地有被自來水管連通，使得每一戶人家都有水可以喝

[11] csunplugged.org, 中華民國軟體自由協會, 不插電的資訊科學, p.99, CC BY-NC-SA 3.0, 2016/3/8 https://classic.csunplugged.org/wp-content/uploads/2014/12/CSUnplugged-2016-03-08.pdf 2019/3/19 visited

呢？

學過抽象化的你，此時一定看著這個地圖想著：「這個地圖上哪些資訊是我不需要的，而哪些資訊才是問題的核心呢？」顯然地，圖上的飛機、狗、太陽、騎腳踏車的小孩、房子的寬敞程度以及樣式等皆不重要，我們需要關注的重點是房子與房子間是否有道路連通，以及這些道路的長短為何，做為挑選興建自來水管道路的依據。

我們可以將任何一個地圖抽象化成一個**圖**，如圖 3-2，圖上的圓圈代表**點**，圓圈中的數字代表點的編號，點和點之間連結的線稱為**邊**，每一個邊旁邊都有數字，這些數字稱作**權重(Weight)**，這樣的圖我們稱為**有權重圖(Weighted Graph)**。對應回去城市中，整個城市就是這張圖，房子是點、道路是邊，而道路的長度即是邊的權重。然而，聰明的你此時可能會想到，建造自來水管的成本高低通常不只與道路長短 (兩戶之間的距離) 有關，可能還會參雜地形、地質、興建材質等因素，但無論影響建造成本的因素有哪些，我們都能事先計算好每條路段興建管線的成本，再將成本化作邊的權重繪製成圖，以達成抽象化，讓我們更接近問題的本質，畢竟權重本是抽象的概念，是量化後的結果。

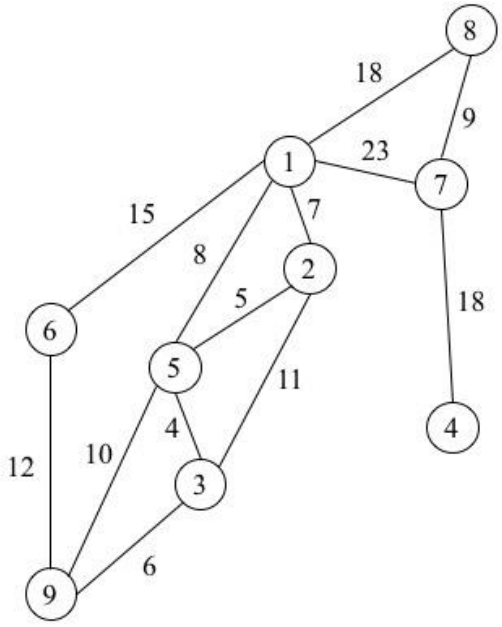


圖 3-2　有權重圖

那麼，假設我們將地圖抽象化成圖 3-2 這樣的有權重圖，應該要挑選哪些邊來鋪設管線，才能最小化總成本並將所有的點都連結起來呢？如果我們任意地畫一畫，可能可以得到以下幾種結果 (如圖 3-3、圖 3-4)：

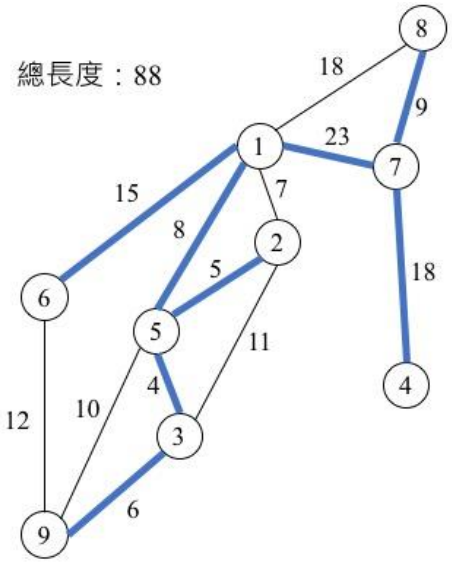


圖 3-3　管線選擇之一

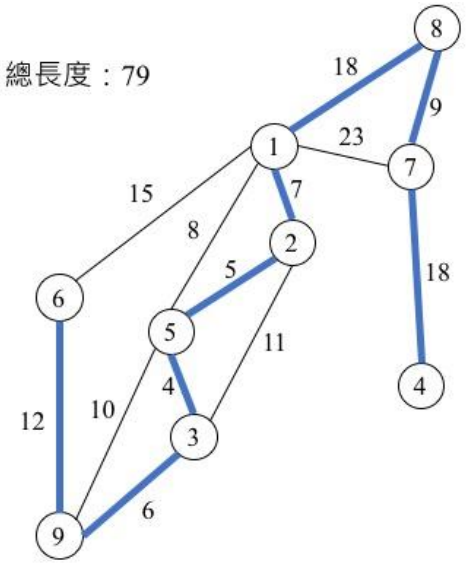


圖 3-4　管線選擇之二

由圖 3-3 與圖 3-4 可以看到，兩張圖都畫了 8 個線段，但很明顯地圖 3-4 所選擇的邊使得總成本較低，看著圖 3-3 我們也可以很明顯地知道如何改善，例如點 1 和點 7 中間的線可以拿掉，改為選擇點 1 和點 8 之間的邊。然而，若是看著圖 3-4，我們是否能就確定這樣的路段選擇就已經使總成本最小化了？有沒有能夠再改進的空間？

欲回答圖 3-4 是否為最好的路段選擇，我們可以直覺地想到一些方法，例如把所有可能的路段選擇都窮舉出來找出最小的路段總和。這樣的方法或許能解這個問題，但現實生活中的地圖中可能多達上萬個點及上萬條邊，如同上一章所學的抽象化，我們的目標並不是找到解就好，而是找到能解這類問題的解法。在程式設計的世界中，解法就是**演算法**。

演算法是一套**定義精確的一系列執行步驟**，在我們日常生活中許多活動也具有類似演算法的概念，例如「早上起床後先刷牙，接著打開冰箱，如果有食材就做早餐，如果沒有食材則穿鞋出門」，這是一系列具有先後順序且定義完整的執行步驟，在程式設計的世界中就是演算法。接下來讓我們試著運用演算法解決上述的架設自來水管線問題。

3.2 貪婪演算法

演算法的種類非常多，其中一種簡單的演算法稱作**貪婪演算法（Greedy**

Algorithm)，其演算法的主軸是：**反覆執行某個任務**，直到求得一組**解**，且每一次在執行該項任務的時候，都去找**當下最好的方案**，不考慮目前的選擇對於整個大問題或未來是否最好。

以日常生活中的活動舉例，假設我們帶著 1000 元去商店買東西，目標是最大化自己的享受程度，那麼套用貪婪演算法即是：每一次挑選一個東西，都選擇當下覺得最能使自己享受的那個商品，直到 1000 元用完為止。這樣的方法有可能得到最佳解，但有時未必，也許買完第 6 個最享受的商品時還剩一些零頭，但若捨棄第 6 個最享受的商品，也許可以買得起第 7 享受及第 8 享受的商品，且加起來的享受度比起第 6 個更高。從這個簡短的例子可知，也許再多精算一下能使這個演算法更好，但這屬於更進階的演算法課程之範疇，容許我們在此先擱置不談。

回到架設自來水管線的問題，我們的總目標是找到能連接所有點的線段，並最小化總成本。如果將貪婪演算法套用至此問題，那麼「反覆執行某個任務」就是**每次挑選一個路段**，「當下最好的方案」是還沒被挑中的路段中**最短的一條**，而「解」則是**連通所有點的一群路段**。總結來說，我們提出的貪婪演算法為「執行數輪挑選，每次都從還沒被挑中的路段中挑最短的一條，直到所有點都被連通為止」。接著就讓我們試著執行看看這個貪婪演算法，演算法流程如下圖 3-5 及圖 3-6。

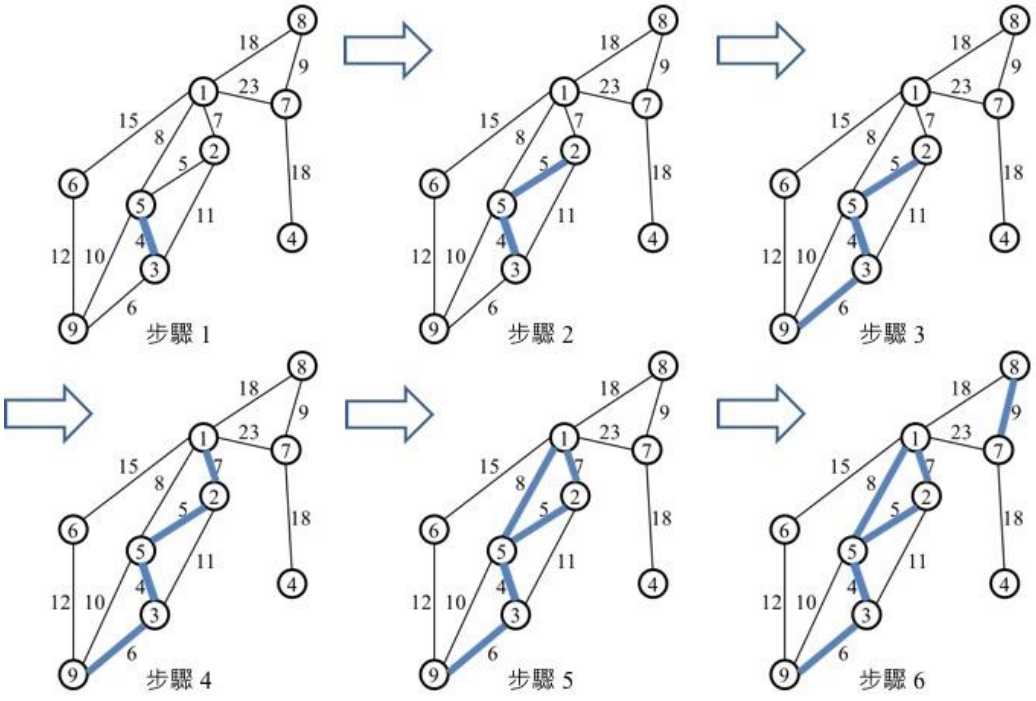


圖 3-5　貪婪演算法流程 1

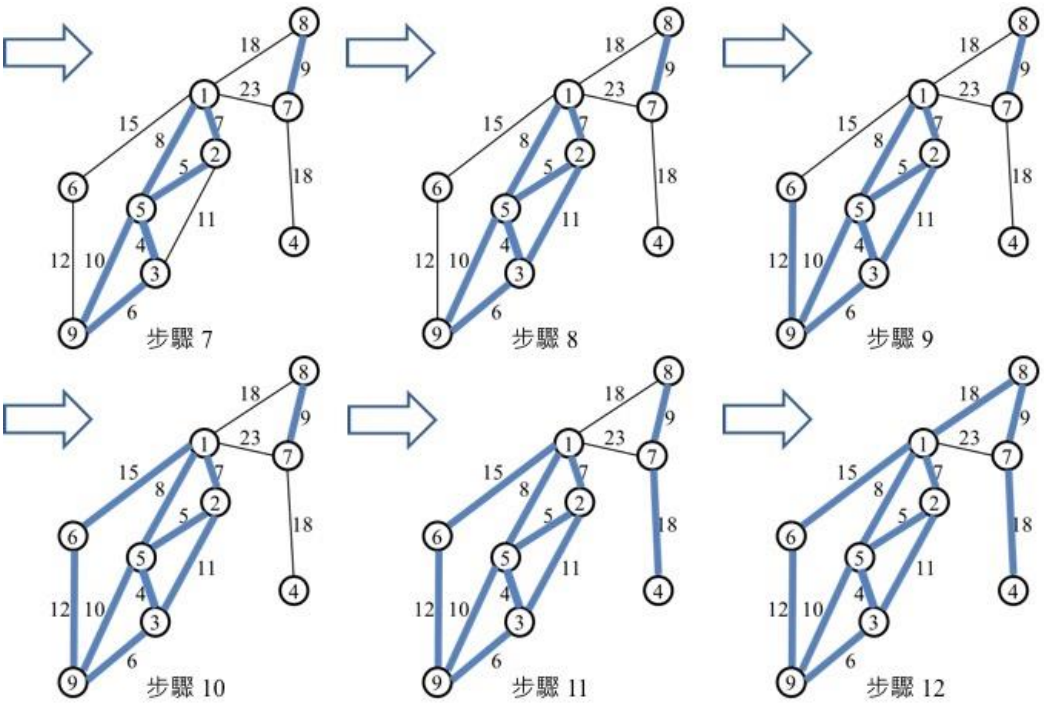


圖 3-6　貪婪演算法流程 2

執行完整個貪婪演算法流程後，結果相當不理想，幾乎所有的邊都被選擇到了，從流程中我們也可以很明顯地看出需要改進的部分，例如圖 3-5 中的步驟 5 選擇了連接點 1 和點 5 權重為 8 的邊，但顯然沒有這條邊，點 1 和點 5 還是能夠連接；步驟 7、步驟 8 以及步驟 10 同樣地選擇了不需要選擇的邊，即使少掉這些邊，這張圖中的每個點野仍然能夠連接，這是便是此演算法的第 1 個缺陷：**不夠好**。

這個演算法還存在著第 2 個缺陷：**定義不精確**。在步驟 10 結束到步驟 11 時，依照我們設定的演算法規定，必須找當下還沒被挑中的邊中最短的一條，然而在進行這個步驟時還沒被挑中的邊之中，最短的邊有兩條：連結點 1 至點 8 以及連結點 7 至點 4 的邊，權重都為 18。在先前提出的演算法中並沒有定義好如何處理最短邊有兩條以上的情況，我們也許能很輕易地隨意挑一條，但對於電腦來說，即使是希望它能「隨意地挑」也是需要我們事先定義好的，畢竟演算法設計及其效能好壞是由設計者負責的，電腦只負責執行設計者所定義好的演算法。

上述的兩個缺陷，在設計演算法時都是必須持續花時間及精力解決的，唯有設計良好的演算法才能讓找出的解接近設計者所預想的結果。在這個例子中我們也試著解決這兩個缺陷，不過在讓演算法更好之前，先讓我們看看定義不精確可能造成的後果。

3.3 演算法定義不精確的後果

為了能讓讀者更了解定義不精確的後果，容我們分享一則改編自網路的工程師笑話。楷翔跟汶佑是一對大學室友，有一天：

楷翔：「回宿舍路上幫忙買一個西瓜，如果看到蘋果，買十個」

結果，身為資工系學生的汶佑買了十個西瓜回來。

相信讀者看到這個笑話一定會想：「想也知道，應該不可能要買十個西瓜。」然而，電腦並不知道所謂的「想也知道」，電腦所知的是人為定義好的一系列規則，並如實地依照這些規則依序執行任務。笑話中只需提到「買十個」我們就能知道是買十個蘋果，是因為在生長過程中所累積的生活經驗形成了買十個蘋果比起買十個西瓜的機率高出許多的「常識」，但電腦並沒有這些「常識」，笑話中的資工系學生就如同電腦；而下指令的學生楷翔就如同演算法設計者，如果定義不夠精確的話，時常得不到我們所期待的結果。

而在企業中，受過大量運算思維及程式設計訓練的資訊工程師，因為學習背景以及工作環境使然，通常習慣將問題定義得很精確，畢竟電腦需要依循很精確的指令執行任務。這也通常是未受過運算思維的管理人員與資訊人員之間的溝通障礙，時常無法提供定義足夠精確的問題或指令給資訊人員，使得資訊人員在將任務指令轉成程式碼交由電腦執行時窒礙難行。

因而我們需要學習運算思維，讓我們能夠更加精確地定義問題。就讓我們試著將笑話定義的更精確一點吧：

楷翔：「回宿舍路上幫忙買一個西瓜，如果看到蘋果，買十個**蘋果**」

或者將標點符號定義精確，使得敘述很明顯地分成兩句話：

楷翔：「回宿舍路上幫忙買一個西瓜。如果看到蘋果，買十個。」

如此一來，這個指令變得清楚多了，身為資工系學生的汶佑終於能夠買到正確的水果，可喜可賀。

3.4 改進演算法效能

瞭解如何更精確地定義演算法後，讓我們回到原本的自來水管線問題，試著

解決演算法不夠好的缺陷。

在原本的演算法設計中，每一次挑選都選擇目前尚未選擇路段中最短的一條，造成了最短路段有兩條時的定義不精確問題，因此我們將它改為「每次都從還沒被挑中的路段中挑最短的一條來『**考慮**』，萬一平手，就**任意挑一條**」。此外，在原本的演算法中，只挑最短的一條路段還有可能會造成**迴路 (Cycle)**，也就是這圖上若干的點中，兩點之間有兩條路可以連通。這對我們的目標來說便是加入了多餘的邊，徒增成本，且並未連接到目前尚未連接的點。因此我們可以將它修改為「如果會造成『迴路』就**不要蓋**，並且**不再考慮這一條**，直到所有的點都被連通為止」。

修正了貪婪演算法後，我們將改進後的演算法套用回原本的問題，其挑選路段的流程如圖 3-7 及圖 3-8。

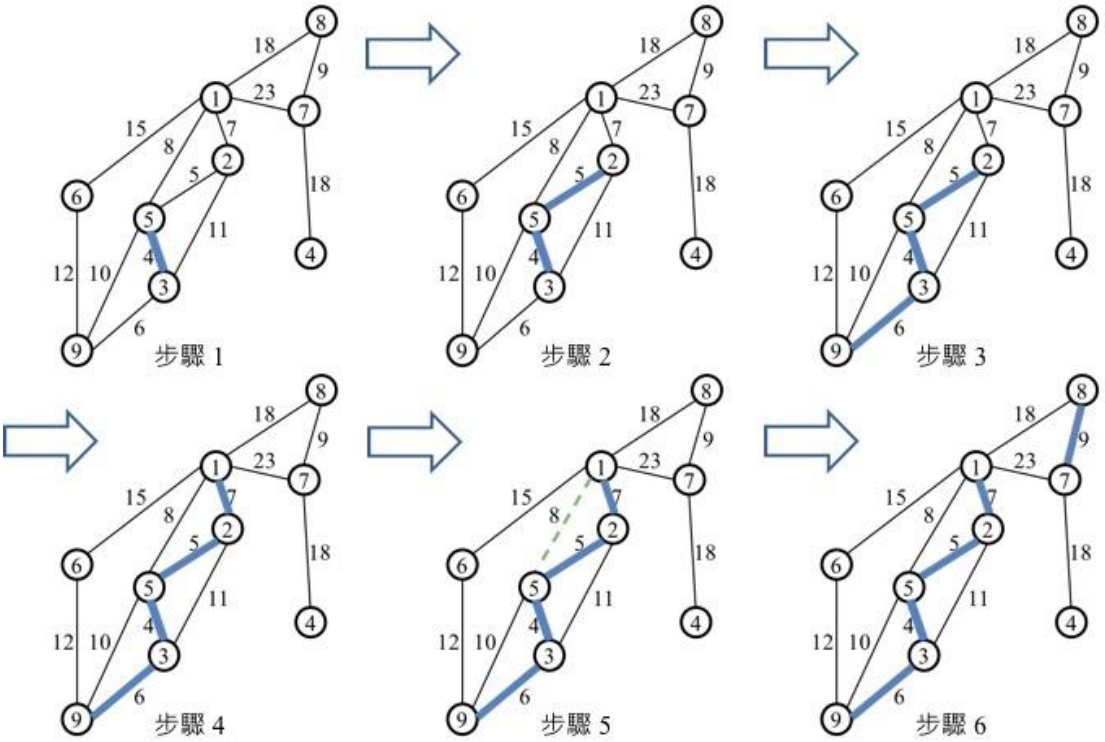


圖 3-7　修正後的貪婪演算法流程 1

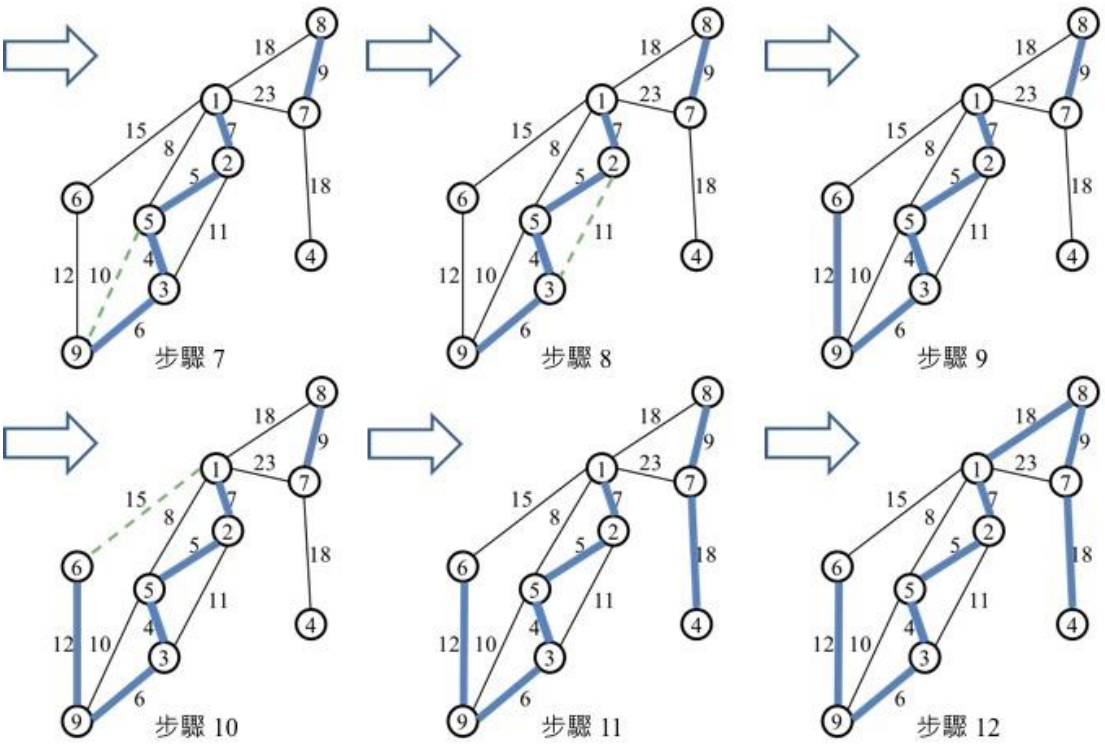


圖 3-8　修正後的貪婪演算法流程 2

在步驟 5 時，權重最小的是連接點 1 及點 5 的邊，權重為 8，但修正過後我們考慮了選擇後是否會造成迴路的因素，因此這個步驟選擇跳過這條邊，並用虛線表示，步驟 7、步驟 8 以及步驟 10 也因同樣的原因跳過該條邊不選擇。此外，在步驟 11 時我們也早已定義了「隨意」挑一條邊，比起先前我們並未定義當兩條最短邊平手時要怎麼做時，即使只多定義了「隨意挑」，仍然能讓電腦順利地將我們的演算法繼續執行下去。很明顯地，修正後的貪婪演算法比起先前提出的貪婪演算法效能好多了，指令也精確了許多。

3.5 虛擬碼

這裡要先暫時跳出剛才的演算法，來說明一下關於描述演算法的方式，也就是**虛擬碼（pseudocode）**。從未學過程式設計的同學可以跳過此章節，若有學過一點程式設計的基礎概念，則可以繼續聽下去。

虛擬碼是一種結構化的演算法描述，我們在描述演算法時，心裡會有一個模糊的概念，要先以結構化的段落式文字敘述，再進行撰寫程式。比喻來說，段落式的文字敘述就是在蓋房子前的概念，諸如蓋多高、幾層樓等，而

虛擬碼則是房子的藍圖，最後會按照藍圖去施工。寫程式也是這個概念，先以虛擬碼描述程式的藍圖，再按此結構去寫程式，通常會用到程式語言中常用到的 if-else、for、while 控制。

舉例來說，如果程式文字化的敘述如下所述：

一、執行數輪挑選，每次都從還沒被挑到的路段中挑一條最短的一條來「考慮」。

二、萬一平手，就任意挑一條。

三、如果會造成「**迴路**」，就不要蓋，並且不再考慮這一條，直到所有點都被連通為止。

則該程式的虛擬碼可以表示為：

while 還有點沒被連通：
- 從還沒被挑中的路段中挑最短的一條來「考慮」；萬一平手，就任意挑一條
- **if** 會造成迴路：
  - 不要蓋，並且不再考慮這一條

  else：
  - 蓋這一條

while 表示當情境滿足後面敘述，還有點沒被連通時，就重複執行下面描述動作的迴圈，挑選最短的一條來考慮；if 表示如果滿足後面描述，會造成迴路，則執行下面描述動作，不要蓋；else 表示如果不滿足上面描述，不會造成迴路，則執行下面描述動作，蓋這一條。可以看出，方框內的虛擬碼相較於原先的文字敘述更接近程式碼，也更能夠在實作程式碼時比照此結構去設計。我們通常會將虛擬碼修得更具體、更接近程式，如下所示，再開始寫程式。

while 挑到的邊數 < n - 1：
- 從還沒被挑中的路段中挑最短的一條來「考慮」；萬一平手，**挑編號最小的**
- **if** 會造成迴路：
  - 不要蓋，並且不再考慮這一條

  else：
  - 蓋這一條

可以發現，挑到的邊數小於 $n-1$ 相較於還有點沒被連通更具體，挑編號最小的也比任意挑一條還要明確。總而言之，在演算法由概念到實作為程式的過程中，會遇到兩種情況：一種是演算法設計上的問題，另一種則是虛擬碼實作的問題。我們會先以虛擬碼進行設計，不考慮實作上的語法或指令問題，將程式的邏輯與結構完善，如此一來，在進行大規模程式時，就能減少在開始撰寫程式時因為邏輯而造成的程式錯誤。

3.6 最小成本擴張樹

上一章節所示範的演算法其實相當有名，稱為**克魯斯克爾演算法（Kruskal's Algorithm）**[12]，該演算法已經被證明可以重複執行直到產出最佳解。這個問題抽象化後，其實可以在程式設計中被歸類為**最小成本擴張樹（Minimum Cost Spanning Tree, MCST 或 MCT）**。Minimum cost 表示要找成本最小，也就是總長度最短，Spanning 代表可以連通所有點，並且要是沒有形成迴圈的**樹（tree）**。這個問題在現實中有許多應用，諸如拉電線、拉電話線、拉網路線、鋪柏油路、蓋高速高路、蓋鐵路，即使在實務上最理想的方案未必真的是一棵**擴張樹（spanning tree）**，先找出一個最小成本擴張樹後再多挑幾條線段，也是一個有效的作法。

[12] 維基百科：Kruskal's algorithm (英) https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

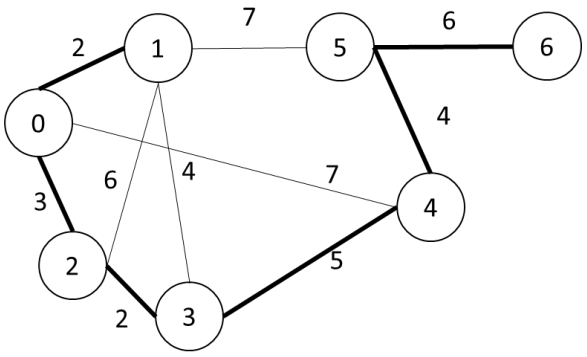


圖 3-6　最小成本擴張樹示意圖

3.7 旅行銷售員的貪婪演算法

現在來用之前提過的旅行銷售員作為第 2 個演算法範例。有一個業務員要從公司出發去拜訪 n 個顧客，我們把該公司編號為 0，n 個顧客的家則依序編號為 1、2 到 n，並以 (i,j) 來表示地點 i 和 j 之間的路段，在路段 (i,j) 上移動要花的最短時間為 d_{ij} 分鐘。現在我們要幫此業務員決定路線，既能最小化這趟拜訪要花的總時間，並且能符合以下要求：每個顧客都被拜訪，且每個客戶家只能經過一次，並從最後一個拜訪的客戶家回到公司。下圖 3-7 表示抽象化後的結果。可以發現，在只有四個點時，雖然可以列出所有可能路徑再找到最短時間路徑，但在點增加時，問題就無法輕易解決了，因此我們下面使用貪婪演算法來找出更好的解法。

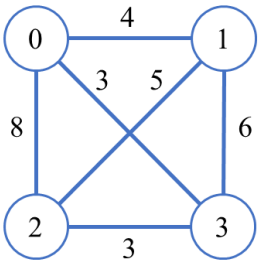


圖 3-7　旅行銷售員問題抽象化示意圖

我們以貪婪演算法來求解這個問題，步驟如下所述，參考圖 3-8：

一、一開始先從公司（位置 0）出發。

二、每當要決定下一個要拜訪的對象時，就從所有還沒被拜訪的顧客

中，挑選移動時間最短的顧客去拜訪，平手則挑編號小的。相較於顧客 1 距離 4、顧客 2 距離 8，顧客 3 距離 3 是最短的，於是移動到顧客 3。

三、如此持續下去直到拜訪完所有顧客。接著依序拜訪到顧客 2、1。

四、再從最後一個拜訪的顧客那邊回到公司 (位置 0)。

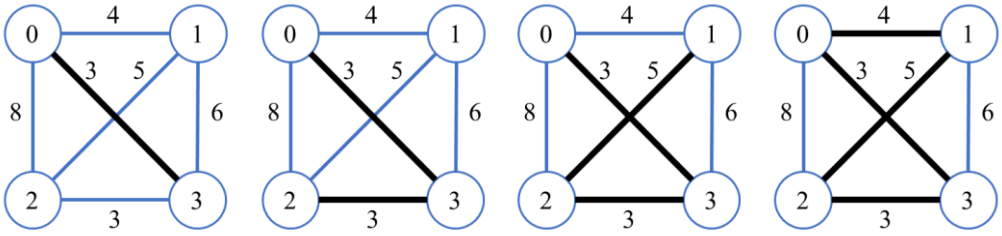


圖 3-8　旅行銷售員演算法過程示意圖

上述的演算法其實無法保證會找到最佳解，因此在演算法設計的課堂中，會進一步探討結果精準度差異如何、何種情況可以得到最佳解。在現實生活中，我們的目的是運用演算法的邏輯思維，更方便與大家溝通，也更能夠判斷哪些事情是電腦可以幫我們完成的。

3.8 旅行銷售員演算法的虛擬碼

在這節中，我們將呈現旅行銷售員的演算法及虛擬碼，沒有學過程式設計的同學可以選擇跳過。

從公司出發
for 第 **i** 次路段選擇：
- 從所有還沒被拜訪的顧客中，找出移動時間最短的顧客；平手則挑編號小的。
- 移過去

從最後一個客戶移回公司

```
cur = origin
for i in range(numLoc - 1):
  # find the next location to visit
  next = -1
  minDst = 999
  for j in unvisited:
    if dst[cur][j] < minDst:
      next = j
      minDst = dst[cur][j]

  # move "next" from unvisited to tour
  unvisited.remove(next)
  tour.append(next)
  tourLen += minDst

  # run the next iteration from the next location
  cur = next

# complete the tour
tour.append(origin)
tourLen += dst[cur][origin]
```

左邊是我們的虛擬碼，右邊則是以 Python 實作的程式碼。一開始銷售員要從公司出發，執行若干次路段選擇，因此用 for 迴圈表示。每次都會在還沒拜訪的顧客中，挑出移動時間最短的點移過去，直到走過所有點後，再從最後一個點回到公司。

以程式碼呈現後，可以看見第一行程式將 `cur` 初始位置設為 origin，也就是公司。第二行程式表示接下來的動作要執行若干次迴圈，會依照要去的地點數目 `numLoc` 有幾個就要跑幾次。每次在執行迴圈時，就挑出下一個去的點，在還沒拜訪的點中，挑出移動時間最短的，將它從尚未拜訪的點 `unvisited` 中移除，再加入我要移動的路徑 `tour`，然後將移動距離加入總移動長度 `tourLen` 中，並把現在的位置 `cur` 設為該點。最後，我們將公司原點 origin 加入移動的路徑 `tour` 中，即可表示最後回到公司。

最後簡單地做個總結，本章的重點即為了解什麼是演算法。演算法是電腦世界中寫程式建構出來的解法，現實生活中任何問題本質上都會有一套解法，而你要做的是要告訴電腦一套精確、且電腦能看懂的解法。在管理中，最重要的就是要將問題用演算法描述，至於演算法的好或壞，則由資工、電機等其他工程師去煩惱吧。

第四章

拆解

4.1 查字典

進入第四個單元**拆解**，在之前的單元中已有說明抽象化與演算法，以上兩者可簡單總結為：抽象化是如何描述問題給電腦聽；演算法是如何描述解法給電腦聽。而兩者有一共同結論：精確、簡要。

現在我們已經會描述演算法，而設計演算法時，還需要進一步思考什麼樣的問題適合用電腦來解決。這個問題要一定程度上容易設計演算法，而演算法設計中，有一些精神能夠幫助我們更容易地分析問題的解決方法，如拆解就是其中的重要概念之一。在此我們以查字典作為範例。

雖然現代大部分人都直接上網查詢，但還是有很多人小時候查過紙本字典，即使沒有查過字典，也會有翻閱書籍查找資料的經驗。以英文字典為例，紙本字典的內頁如圖 4-1，假設我們查找了「dictionary」這個字，就能找到它的音標、意涵等資訊。

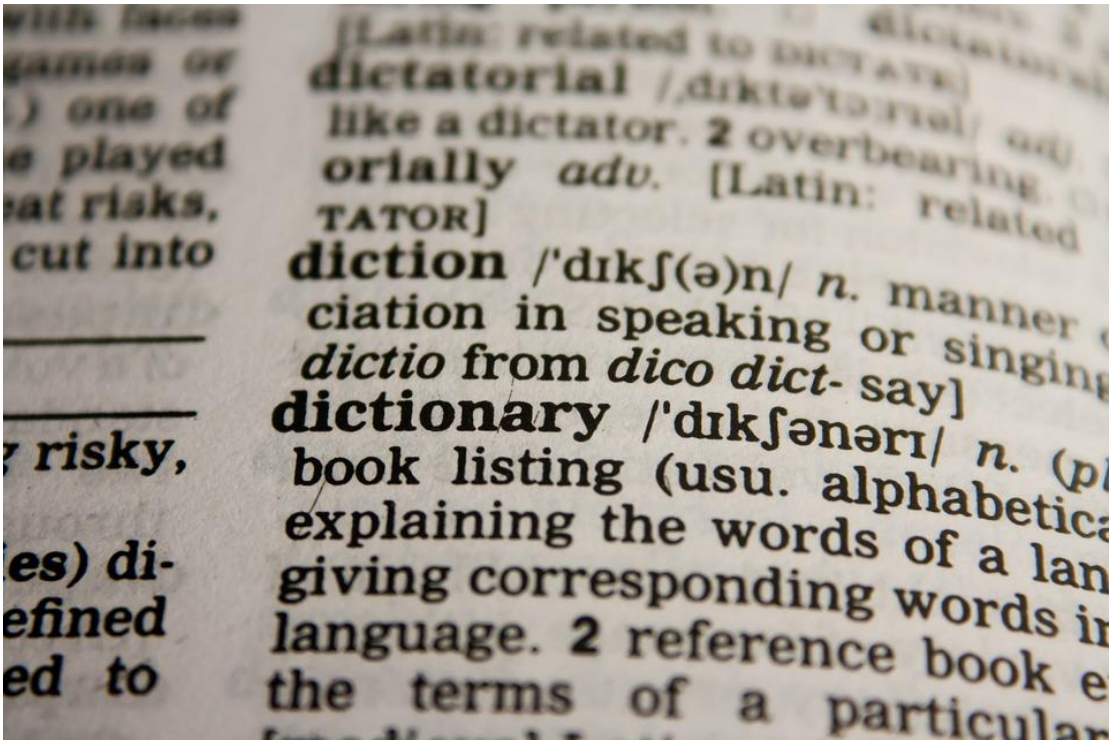


圖 4-1　Oxford Dictionary of English 內頁示意圖

查字典這件事其實頗為困難，在紙本字典時代極為不便，經常要左翻右找比誰的動作比較快。那麼，查字典要怎麼查會比較快呢？比如說要查「醬」油的「醬」，可以從筆畫出發，在 18、19 畫左右便能找到這個字，但這個方法可能不夠快；

若從注音出發，「ㄐㄧㄤ」也可以；查部首，「酉」部。那「叟」要怎麼查？不僅部首不太容易辨識，也不太了解拼音、讀法，甚至也不是很確定有幾畫。這真的蠻困難的，讓我們參考前人的智慧。

圖 4-2　華文課本內文示意圖[13]

在課文中，小男孩昨天從叔叔那獲得一本書，叔叔告訴他「字典就像你的老師，他可以教你認識很多的字。」叔叔教他查字典的方法有很多種，可以找部首、注音、筆畫數。

隔天，妹妹跑來跟哥哥說「我學會查字典了」，奇妙的是明明叔叔是教哥哥查字典，妹妹竟然也學會了。於是哥哥就問她「是嗎？你怎麼查呢？」「老師教我們用注音和部首來查」「那你當我查『淺』這個字。」「它是水部就是水很少的意思。」「果然很方便。」「有了字典就多一位老師了！」

透過以上的例子，希望可以幫助同學更具象化地了解查字典的過程。

從上述可以發現查中文字典相當困難、複雜，接下來，我們從英文字典的例子來了解查字典這個問題。

[13] 僑委會全球華文網華文泰國版課本第四冊 p.51、p.55, 民 102.12, CC BY-NC-SA 3.0, https://www.huayuworld.org/eBookstore/PDF/%E5%9C%8B%E5%88%A5%E5%8C%96%E6%95%99%E6%9D%90/064.%E8%8F%AF%E6%96%87(%E6%B3%B0%E5%9C%8B%E7%89%88)%E8%AA%B2%E6%9C%AC04.pdf

假設有一本英文字典，要如何知道「dictionary」這個字是否存在，若存在，它代表什麼意思？或者若要查「supermodularization」這個字存不存在，若存在的話又是什麼意思？

這個問題確實很困難，比如我們不知道「supermodularization」是否存在，但即使它存在，就保證它一定會出現在這本字典裡嗎？也不一定，因為這個字看起來相當罕用。而這本質上是一個**搜尋問題（searching）**，搜尋問題在程式設計或是資工、資管的領域相當常見。比如有一個戶政系統，從系統中任意輸入一串身分證字號，查找此身分證字號是否存在於系統中、有沒有相對應的人，這便是一個搜尋問題；在手機聯絡人中輸入一串電話號碼，搜尋是否有這樣一個聯絡人；在 LINE 的通訊群組輸入暱稱或是用帳號找一個人、輸入一個 ID，比對在全世界安裝 LINE 的人之中是否有人的 ID 相符，以上這些皆是搜尋問題。搜尋問題必然要有辦法處理，否則這些應用將無法成立。

搜尋可以如何搜尋？以查字典來說，有一演算法稱為**線性搜尋（linear search）**，在字典上的應用就是一頁一頁翻，因為字典的頁數有極限，如果有找到，便可以確認這個字在這本字典中存在；如果翻到盡頭都沒有就是這個字在這本字典中不存在。

這個方法很慢但是很有效，把整本字典翻完終究會得到答案，所以線性搜尋演算法終究可以得到正確解答。但一本字典逐一查詢要找到什麼時候？使用 LINE 的人有幾千萬甚至上億，輸入任一 ID 要判斷是否存在又要耗費多少時間呢？接下來，我們基於拆解來介紹一個更好的演算法。

4.2 二分搜尋法

本節將介紹更好的演算法，也就是**二分搜尋法（binary search）**，這個方法事實上很普遍，平常經常會使用到。以查字典為例，查字典會一頁一頁翻嗎？不會，一定從正中間那頁開始，如果剛好有包含要查的字即可確定要查的目標字是存在的；如果沒有，便進一步查找前半部或者後半部，那要如何決定前後順序？假設現在查的是英文字典，英文字母皆有排序，假設翻到「M」在正中間，要查的字開頭是「D」，而「D」在「M」前面這是已知的，此時就不需要查後半部的字典了。這時把字典一撕，後半部丟進垃圾桶，來查前半部，並把前半想成一本新的字典再做一次，持續重複直到只剩下一頁的時候就停下來結束這個工作，這

就叫二分搜尋法[14]。

這樣的求解策略就稱為拆解，也就是把大問題拆成數個小問題或者模組，大問題不會解就解小問題，小問題都解出來了以後組合起來，希望就是大問題的解。字典就像是一個大問題，要找的字是否在大字典裡面？不知道，就把其拆成兩半，分別解前後兩段，若兩半有其一含有目標字便是有，如果兩半都沒有就是沒有。只不過字典這個例子蠻特別的，前半部如果不會解就再切一半，持續重複切分與查找的過程，這就是所謂的拆解。

如果要查「dictionary」這個字，就翻到字典大約 50%的位置，大概會是什麼呢？就要看英文字的分佈，如下表 4-1，A 開頭的字大約有 11.6%，B 開頭的字大約有 4.7%，而 T 開頭的字偏多，Z 開頭的字較少。所以分佈於中間的字應該是 M 開頭，而目標字開頭是 D，就如同上段所述，後半部便不需要，僅需查前半部的字典。

字母	字首比率	累積比率	字母	字首比率	累積比率
A	11.60%	11.60%	N	2.37%	54.37%
B	4.70%	16.30%	O	6.26%	60.63%
C	3.51%	19.82%	P	2.55%	63.18%
D	2.67%	22.49%	Q	0.17%	63.35%
E	2.01%	24.49%	R	1.65%	65.01%
F	3.78%	28.27%	S	7.75%	72.75%
G	1.95%	30.22%	T	16.67%	89.42%
H	7.23%	37.45%	U	1.49%	90.91%
I	6.29%	43.74%	V	0.65%	91.56%
J	0.60%	44.34%	W	6.75%	98.31%
K	0.59%	44.93%	X	0.04%	98.35%
L	2.71%	47.63%	Y	1.62%	99.97%
M	**4.37%**	**52.01%**	Z	0.03%	100.00%

表 4-1　英文字首比率表[15]

前半部再翻一半，大概會在 25%的位置，字首為 F，而 D 依然在 F 前面，所以刪去後半，留下前半繼續查詢。第 3 次翻頁約會翻在 12.5%的位置，字首為 B，要找的字會落在後半部，刪去前半部，查著後半部。第 4 次翻頁約會落在 18.75%的位置，會是 C 開頭的字，繼續往後半查詢。第 5 次翻頁會落在 21.88%

[14] 在此為強調資料的切分以及選擇，以撕字典並捨棄作為表達，建議不要實際應用此方式查找資料。

[15] 維基百科：Letter_frequency Letter frequency, CC BY-NC-SA 3.0, https://en.wikipedia.org/wiki/Letter_frequency

的位置，已經找到 D 為字首的字了，接下來看第 2 個字母或是前後瀏覽就可以找到目標字。

演算法有很多種，為什麼資工、資管科系需要修 4 年這麼長的時間，有這麼多東西可以修嗎？相關系所的學生並不是持續在學各種程式設計、程式語言，他們事實上花很多時間在學習如何把演算法寫得更好，演算法要做效率分析等。這些就是所謂專業人士或專業的程式設計師在煩惱的事情。

所以接下來就試著來分析上述所提兩種演算法的效率。假設有本英英字典，大約有 400 頁，查任一個字，因為有些字在前半部，有些在後半部，平均起來大概要翻 200 次左右；二分搜尋法最多翻 $log_2 400$ ，大約為 8 次，400 頁翻 1 次為 200 頁，翻 2 次為 100 頁依此類推，翻 8 次左右這本書就只剩下一頁，這一頁就可以得到答案。

而截至 2005 年為止，牛津字典裡約有 600,000 個字詞，假設一頁中有 20 個字，則會有 30,000 頁。隨便查找一個字，線性搜尋需要 15,000 次；用二分搜尋法 $log_2 30000$ 則約 15 次，兩種方法就有 1000 倍的差異，表示同時可以做 1000 個倍率，以線性搜尋法查 1 個字，用二分搜尋法可以查 1000 個字。假設是戶政系統，台灣有 2,300 萬人，除以 2 為 1,150 萬，log 以 2 為底運算大概是 25 左右，這之間相差非常多；如果是 Google，在搜尋引擎上有十幾億、幾十億個詞條，用線性搜尋會耗時過長，Google 也無法同時讓這麼多人搜尋，所以 Google 肯定要用二分搜尋法，甚至是比二分搜尋法更好的演算法，才能同時讓全世界的人上去搜尋如此多元的資訊，而這當然是更複雜的事情了。

從這些例子裡可以看出，好與不好的演算法，在解相同問題時或許都會得到答案，但在效率上可以有天壤之別。有興趣的同學若之後學習程式設計，當然可以設計各式各樣的解法，有些解法好有些不好，但當設計出好的解法時是非常令人興奮的，這也是專業的程式設計師或是軟體工程師要進一步精進自己的地方。

4.3 二分搜尋法虛擬碼

以下為二分搜尋法的虛擬碼，基本上的概念是要持續的切分，直到得到結論為止。

```
while 還沒結論：
• if 只剩一頁：
    • if 要查的字在這一頁：
        • 查到啦！
    • else：
        • 不存在！
• else：
    • 翻到中間那一頁
    • if 要查的字在這一頁：
        • 查到啦！
    • else if 要查的字在這一頁前面
        • 撕掉後半本，查前半本
    • else：
        • 撕掉前半本，查後半本
```

如果現在只剩下一頁，如果在這頁中包含要查的字，便找到目標字；如果不在，那這個字就不存在於這本字典中，結束整個迴圈。要是現在字典不只一頁，就翻到中間那頁，如果目標字在翻到的那頁，便找到目標字；如果不在，那就判斷目標字在此頁的前面還是後面，如果是前面就撕掉後半部、查找前半部；不然就撕掉前半部，查找後半部。

上述的所有步驟是放在 while 迴圈當中，所以通常會撕掉後半或前半再回到迴圈開頭。迴圈每跑一圈，東西就會少一半，直到得到結論為止。

透過這個虛擬碼可以發現，第 4.2 節所提及的二分搜尋法僅是粗淺的概念而已。如果有這個虛擬碼，有樣結構性的描述以後，如果真的要寫程式，把這個演算法實作出來，照著這個結構去寫相對方便且容易許多。

4.4 旅行銷售員問題公車版

在第 4.1、4.2 章中拆解有了關於二分搜尋法的例子，第 2 個例子讓我們再來解一次旅行銷售員問題，這個問題已經在抽象化模組提及過，一個業務員要從公司到顧客那邊拜訪，並將公司編號為 0，n 個顧客的家便依序由 1 到 n 編號，並以 d_{ij} 表示 i 和 j 之間有多久的路程。

現在問題變得稍微複雜，老闆提供經費可以挑一個路段搭公車，圖 4-3 和演算法模組中的圖相同，4 個點之間的 6 條邊各有各的步行時間；圖 4-4 則有搭車所需花費的時間，可以看到搭車的部分每段時間會縮短，但縮短程度不一。

而現在的任務便是幫業務員決定路線，最小化這趟拜訪所耗費的總時間。同樣是找一條路線經過所有的點，並加上搭公車方案，這就讓問題變得更複雜。例如在兩圖中 $(0,2)$ 路段搭公車時間與步行時間相差最多，那就業務員就一定會在這個路段搭公車嗎？不一定，因為極有可能會繞過，就不需要把公車錢浪費在這裡。基本上，要如何利用公車並不是這麼容易可以決定的。

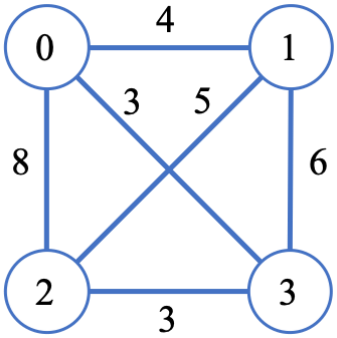


圖 4-3　旅行銷售員問題步行時間示意圖

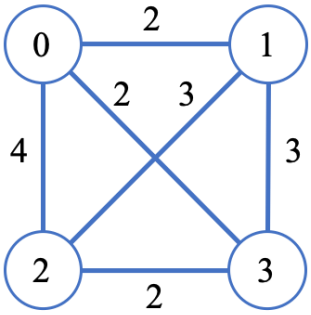


圖 4-4　旅行銷售員問題乘車時間示意圖

現在需要一個演算法來解決上述的問題，可以嘗試以下的解法。把所有可能搭公車的路段都嘗試一次，每次解一個子問題，那個子問題就是一個「彷彿」沒有搭公車議題的旅行銷售員問題。上述的意思也就是去看看如果在某一路段上搭公車，該路段時間變短，也就得到了一個單純的旅行銷售員問題，並求解。

舉例來說如果在 $(0,1)$ 搭公車，業務員從公司到顧客 1 的家所需要的移動時間就會由 4 變成 2，就得到一個旅行銷售員問題，求解一次；如果在 $(0,2)$ 搭公車，所需要的時間就會由 8 變成 4，就再得到一個旅行銷售員問題，再解一次，依此類推。總共會得到 6 個旅行銷售員問題，把所有解完後，相互比較得到最短的花費時間，便選擇那條路線。

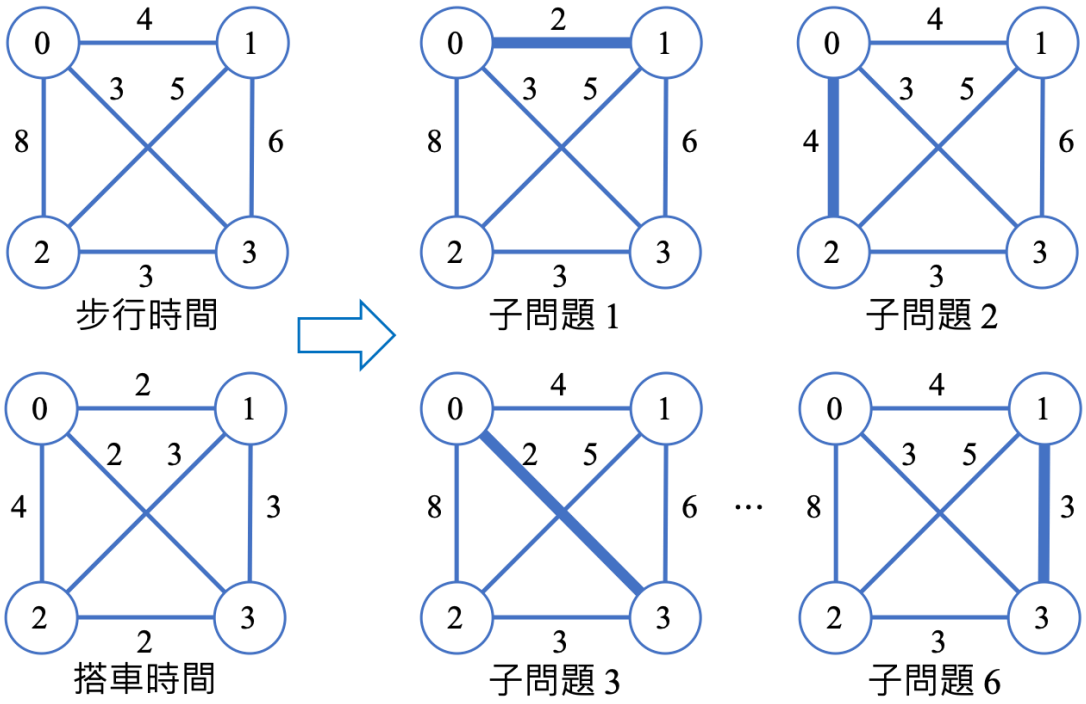


圖 4-5　旅行銷售員問題公車版拆解示意圖

由上圖 4-5 可以看到，子問題 1 便是把 $(0,1)$ 換成距離為 2；子問題 2 把 $(0,2)$ 換成距離為 4，依此類推，就能將 6 個子問題都分別建構出來，每個子問題都可以用前面模組所提及的貪婪演算法求解，最終哪個子問題給出最理想的答案，便用那個方向執行。

這個演算法不一定好，甚至可以明顯地看出許多改進的空間，不過本節不討論這個議題，在此無非就是想讓大家看到，這是一個拆解的方式。本來的問題很複雜，把問題拆成若干個較容易解的問題後，再將這些問題的答案統合，變成原本問題所要的答案。這是一個粗淺的例子，是為拆解的另一個應用。

透過本章可以看出在寫程式時經常做拆解，把大問題拆成數個小問題，將小問題個別解出後來建構原本大問題的答案。而在日常生活中也經常應用，比如要辦營隊給高中生參加，不可能全部的工作都由總召負責，會選副召、活動部長、財務部長……，所有人各司其職，總和起來就成為一個成功的專案、成功的營隊。總召的任務是組合每個人完成的成果，但個別的任務就個別解決。如果常寫程式，相對會具備拆解的運算思維，就會習慣將問題拆解成小問題，當這樣的習慣被帶進生活，在處理問題時，更有可能理性地分析問題，將問題變成更精確的模組後再一一解決，這確實是會發生的。

第五章

轉化

5.1 博物館攻略

最後一章來談談**轉化**，轉化與拆解一樣，都是一種演算法的解題思維。解決問題的方法有很多種，拆解是將問題變小，而轉化則是**將不會的問題變成會解的問題**。我們從博物館的例子開始。

有的博物館很大，如圖 5-1，我們能否找到一個方法，一次將所有博物館都繞完。像是從某個入口進入博物館，一次逛完所有房間，且不能進入同一個房間兩次，最後回到原本的入口。值得注意的是，部分房間有多扇門，但此問題並非是要經過每一扇門，而是要經過每一個廳。此類問題有些很簡單，如圖 5-2，而有些則很困難，如圖 5-3，我們可以應用一些技巧去處理這種問題，

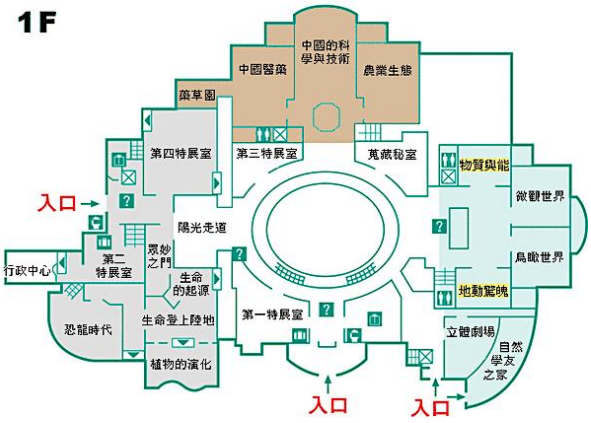


圖 5-1　國立自然科學博物館[16]

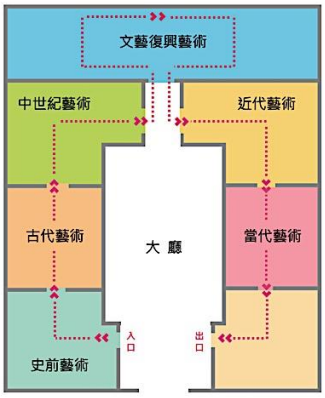


圖 5-2　博物館展廳動線

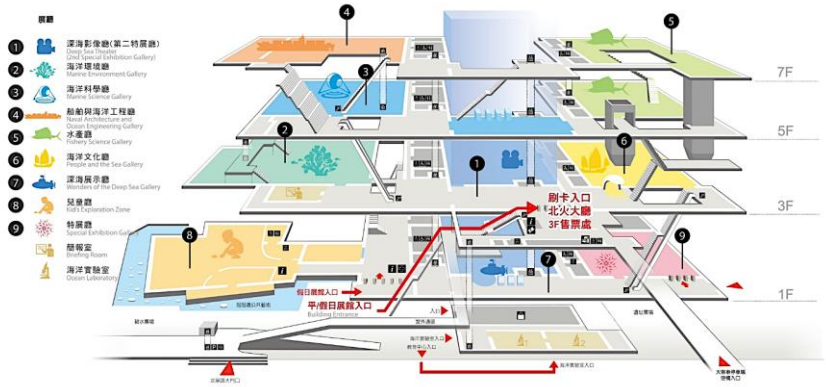


圖 5-3　國立海洋科技博物館[17]

[16] 本作品來自國立自然科學博物館網站 http://www.nmns.edu.tw，2019/3/25 取得

[17] 本作品來自國立海洋科技博物館 https://www.nmmst.gov.tw/chhtml/content/369，2019/3/25 取得

其中一個技巧是將此類問題，透過將任意一個房間連通關係抽象化成**圖**，像是房間對應於點、門對應的是邊。圖 5-4 為圖 5-2 轉化後的圖，其中圖 5-2 的古代藝術廳對應圖 5-4 的點 3，而史前藝術廳轉換成點 2，而廳中間有一扇門，對應的是圖 5-4 中點 2 和點 3 的連線。值得注意的是，**這個問題並不是七橋問題**，博物館問題是一筆畫走完所有的點，點不能重複；七橋問題則是一筆畫走完所有的邊，邊不能重複！

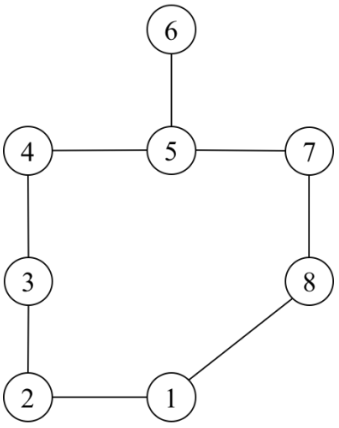
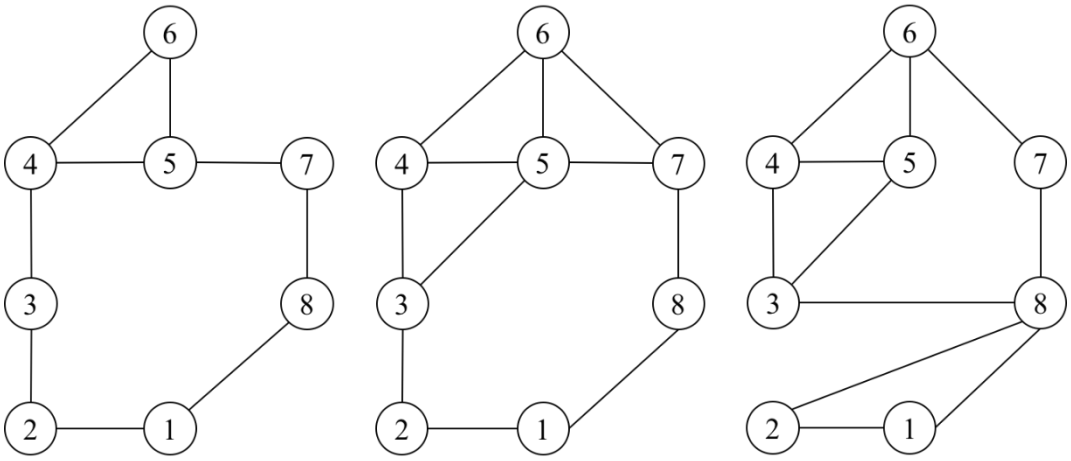


圖 5-4　將圖 5-2 轉化後的圖

有些圖可以一筆畫走完所有點，如圖 5-5 裡面的 (1)、(2)，有些則不行，像是圖 5-5 裡面的 (3)，由於 (3) 裡的點 8 不可能只通過一次，因此 (3) 這張圖無法一筆畫走完所有的點。透過這三個例子，你是否能看出一些規則？



(1)　　(2)　　(3)

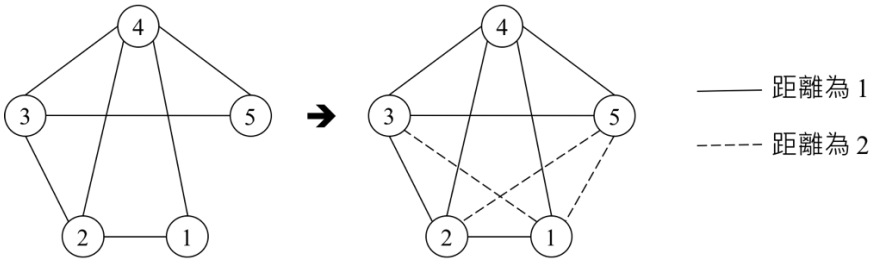
圖 5-5　一般性規則

5.2 博物館攻略轉化為旅行銷售員

一筆畫走完所有的點並且回到原點的路線，稱為一個**漢米爾頓迴路（Hamiltonian Circuit）**，而之前提過的七橋問題，稱為**尤拉迴路**，是一筆畫走完所有的邊並且回到原點的路線。漢米爾頓迴路問題是很困難的，目前還沒有人找到簡單的一般性規則，如果有很多點和邊，問題便會變得非常複雜。

對於此類問題，可以利用**「轉化」**，也就是將問題進行簡化，把一個問題 A 轉化成另一個已知的問題 B，透過解決 B 來解決 A。在前面章節我們提過非常多種問題，像是七橋、背包、旅行銷售員等問題，漢米爾頓迴路問題可以被轉化成哪一種？

仔細地回想後，我們可以發現漢米爾頓迴路問題（A）可以被轉化成旅行銷售員問題（B）。旅行銷售員問題和漢米爾頓問題一樣都需要點和點與點之間的關係，旅行問題只比漢米爾頓問題多需要點與點之間的距離。所以如果 (i,j) 存在於 A 的圖上，就在 B 中把 d_{ij} 設成 1；如果 (i,j) 不存在於 A 的圖上，就在 B 中把 d_{ij} 設成 2，如圖 5-6。



漢米爾頓迴路 A　　旅行銷售員問題 B

圖 5-6　漢米爾頓迴路轉化為旅行銷售員問題

將問題 A 轉化為問題 B 後，試著找出問題 B 的解，如果問題 B 中最短路線的總距離等於 n，A 中就有漢米爾頓路徑，如圖 5-7。反之，如果 B 中最短路線的總距離超過 n，代表最佳路徑有走過虛線，原本的題目必然要經過不存在的路線，對應問題 A 中就沒有漢米爾頓路徑，如圖 5-8。

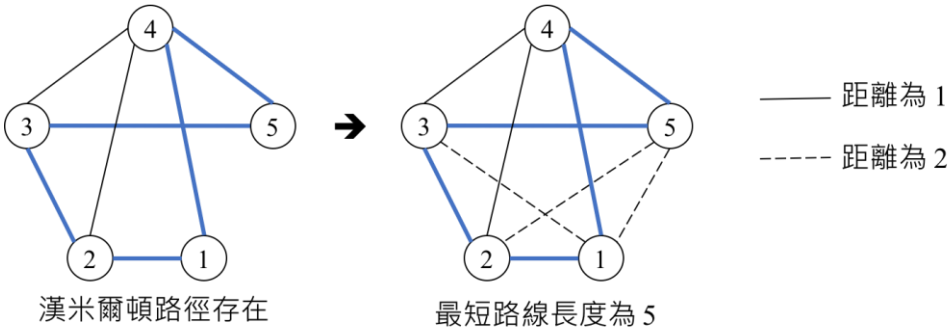


圖 5-7　存在漢米爾頓路徑

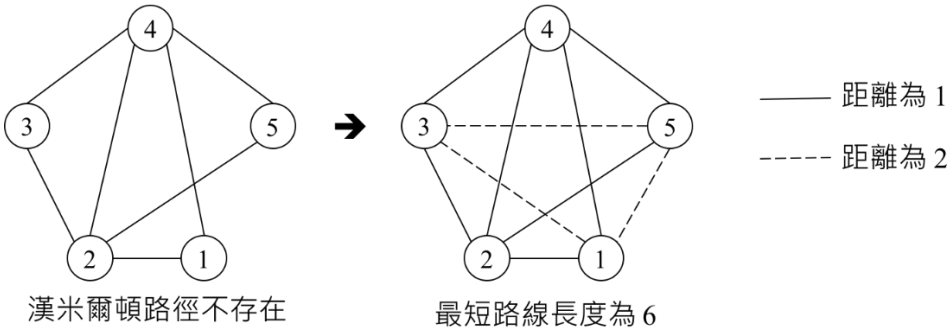


圖 5-8　不存在漢米爾頓路徑

5.3 分割問題

子曰：「不患寡而患不均」，有資源的地方就需要考慮如何分配。舉例來說，現在兩人一起準備出發去擺攤，發現還有下列工作必須完成，如表 5-1，若現在一個工作只能一個人做，一個人一次只能做一個工作，該如何分配工作，才能最快出發前往擺攤？

編號	1	2	3	4	5
工作	煮珍珠	泡奶茶	烤餅乾	做海報	借推車
所需時間（分）	20	30	60	15	25

表 5-1　擺攤前工作所需要的時間

看一下題目，馬上可以知道在一個人做工作 1、2、5，另一個人做工作 3、4，兩個人一起做 75 分鐘後一起出發時會是最佳解。

另一個例子：兩個海盜要分一堆寶箱，如表 5-2，一個寶箱只能給一個人，該如何分配寶箱，才能讓兩人的金幣數差距最小？

編號	1	2	3	4	5
金幣數	20	30	60	15	25

表 5-2　海盜的寶藏

看一下也可以發現，海盜的例子與擺攤的例子會有一樣的最佳解，當一個人拿 1、2、5，一個人拿 3、4，兩人都會拿到 75 個金幣，金幣數差距為 0。上述的兩個問題稱為**分割問題（partition problem）**，抽象化地來說，是給定 n 個正整數，將正整數分成兩組，並使兩組內的數字和差距最小，因為只有總和為偶數才有可能「完美分割」，此問題的 n 個正整數的總和通常為偶數。

「完美分割」是給定 n 個正整數，將正整數分成兩組並使兩組內的數字和恰好相等。如果給定 n 個正整數，我們如何判斷是否能將這些正整數進行「完美分割」？這問題很難，現行沒有簡單的一般性規則，如果 n 很大，幾乎找不到最佳解。但分割問題可以透過轉化去處理，哪一個問題和分割問題最相似呢？

5.4 分割問題轉化成背包問題

分割問題（A）可以被轉化成背包問題（B），A 裡有 n 個整數為 a_1、a_2 直到 a_n，而對應的背包問題需要物品重量和物品價值等資訊，可以透過令 B 中有 n 個物品，將第 i 個物品的重量與價值設定為一樣的 a_i，並令背包容量上限為所有整數加總的一半 $\sum_{i=1}^{n} a_n /2$，將問題 A 轉換為問題 B。

以表 5-2 的題目海盜問題為例，將其轉換為背包問題，數學式在圖 5-9。如果 B 的最佳解總價值恰好是 $\sum_{i=1}^{n} a_n /2$，A 就有完美分割，如圖 5-9，B 的最佳總價值為 75，等於整數加總的一半 75。如果 B 的最佳解總價值低於 $\sum_{i=1}^{n} a_n /2$，A 就沒有完美分割，如圖 5-10，B 的最佳解總價值只有 80，小於整數加總的一半 85。

$$
\begin{aligned}
\max \quad & 20x_1 + 30x_2 + 60x_3 + 15x_4 + 25x_5 \\
\text{s.t.} \quad & 20x_1 + 30x_2 + 60x_3 + 15x_4 + 25x_5 \leq 75 \\
& x_i \in \{0,1\} \quad \forall i = 1, \ldots, 5
\end{aligned}
$$

圖 5-9　A 有完美分割

$$\begin{aligned} \max \quad & 20x_1 + 30x_2 + 60x_3 + 15x_4 + 45x_5 \\ \text{s.t.} \quad & 20x_1 + 30x_2 + 60x_3 + 15x_4 + 45x_5 \leq 85 \\ & x_i \in \{0,1\} \quad \forall i = 1, ..., 5 \end{aligned}$$

圖 5-10　A 沒有完美分割

在轉化這章節，我們只提到漢米爾頓問題和分割這兩個例子，實際上轉化有非常多的應用，有些類似我們的例子。軟體工程師通常利用轉化的技巧，將不會的問題，轉化成會解的問題。而這種問題在往下進階是計算機科學領域的理論，超出這門課的範圍[18]。

運算思維是一門入門課，如果你有興趣，歡迎學習程式設計等相關課程，關於此領域後面還有很多東西可以學，可以了解更多運算思維和資訊科技的應用。

[18] 在這門課我們介紹漢米爾頓迴路、旅行銷售員、分割、背包等問題，這些問題在學術上統稱為 NP-hard 問題，這些都是非常困難的問題。在學術上我們相信在合理的時間內是找不出最佳解。雖然在第五章，我們可以將漢米爾頓轉化成銷售員問題，但事實上旅行銷售員問題也是非常困難，所以其實是不能用轉化解決漢米爾頓問題的。透過這樣的轉化，我們能因為漢米爾頓問題可以被簡化成旅行銷售員問題，而漢米爾頓很難，進而推估旅行銷售員問題是解不出來的。分割問題和背包問題也是一樣的，由分割問題的困難可以推估背包問題也是解不出來的。上述部分是為了釐清轉化並非代表有解，如果未來想要了解該如何處理沒有解的問題，後續有相關的課程可以學習。

索引

書　　名 / 貓都學得會的運算思維
編 著 者 / 孔令傑、陳庭姍
地　　址 / 臺北市羅斯福路四段1號　國立臺灣大學資訊管理學系
電　　話 / (02) 3366-1200
印刷日期 / 民國109年7月(第一版)