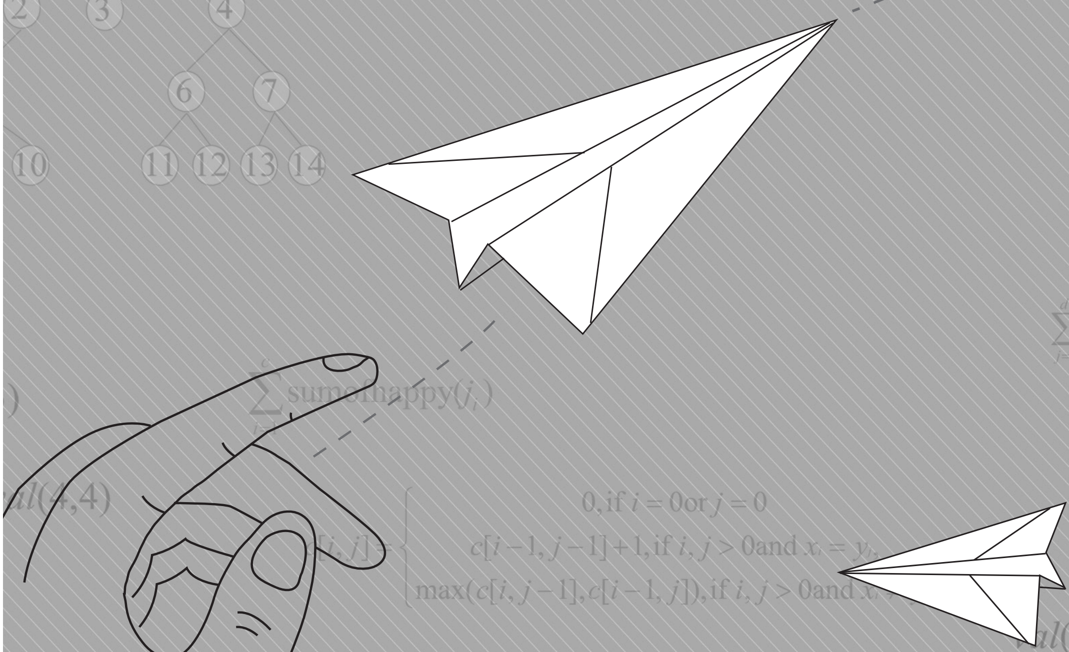
10

演算法的難題

章節大綱



10.1 何謂 NP-complete？

「何謂 NP-complete？」

簡言之：「屬於 NP-complete 的問題，目前並沒有 $O(n^k)$時間複雜度的演算法被設計出來。也並沒有被證明，這樣的演算法是不存在的。」

NP-complete 是一個集合(set)，包含許多困難的演算法問題。若將解決演算法問題，譬喻成電動遊戲中的怪獸，這一類怪獸的戰鬥力是十分龐大的，目前人類尚未完全戰勝他們。更不幸地是，屬於 *NP-complete* 的問題，幾乎遍及所有資訊應用領域。例如，***漢米爾路徑***(Hamiltonian circuit)問題(可否找到一條路徑將每一個點剛好經過一次又回到原點)，就是這樣的難題。

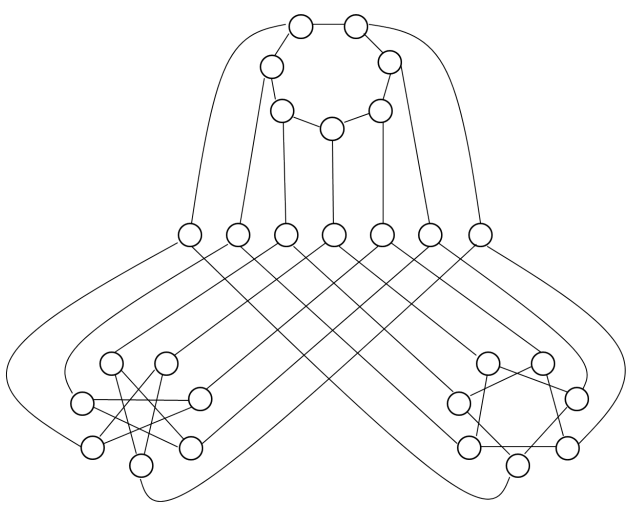


圖 10.1　您可以找到一條漢米爾路徑，將每一個點經過剛好一次又回到原點嗎？

面對 *NP-complete* 的問題時，若有人想要設計出一個有效率(即擁有 $O(n^k)$時間複雜度)的演算法，將是一個很大的挑戰。注意，本章中的 n 是指輸入的數量，而 k 是一個任意大小的常數。

「這個程式為什麼跑得這麼慢啊？」

「我很努力的試過其他演算法及資料結構，但是都只是稍微快一些。後來才發現這是一個 NP-complete 的問題。」

「蝦米! NP-complete？真的還假的？」

「不然您自己試一試⋯」

10.2 集合 P 與集合 NP

一個決策問題(decision problem)是指其輸出，只有「是」或「否」的問題。例如，搜尋問題為詢問 x 是否出現在一個集合 A 中?若有則輸出「是」，否則輸出「否」。如此，搜尋問題為一個決策問題。為了方便討論，本章僅對決策問題進行難易分類。決策問題中，有些較簡單，有快的演算法可以解決之；但是，也有一些較難的問題，目前只存在慢的演算法。

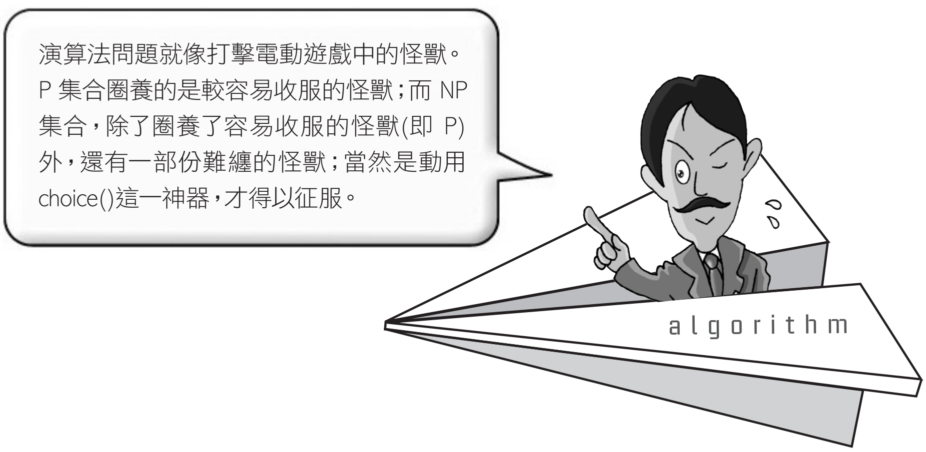
當一個決策問題存在一個 $O(n^k)$時間複雜度的演算法時，則稱此問題落在 P 的集合中。落在 P 中的決策問題，在本章中，可以視為較簡單的問題。

相對地，有一些決策問題，人類目前尚無法將他們歸入集合 P 中。為了思考這些問題，於是在一般演算法可採用的功能上，擴增以下虛構的新指令。這些新指令雖然不存在於現實中，但是對探討這些難題的性質及彼此的關係，有很大的幫助。以下是這些虛構的新指令：

1. **choice(S)**：自集合 S 中，選出會導致正確解的一個元素。當集合 S 中無此元素時，則可任意選擇一個元素。
2. **failure()**：代表失敗結束。
3. **success()**：代表成功結束。

其中 choice(S)可以解釋成，在求解的過程中，神奇地猜中集合 S 中其中一個元素，使其結果是成功的；並且這三個指令只需要 $O(1)$時間來執行。當然，choice(S) 是如何快速猜中的，在此是不需討論的，因為畢竟它只是虛構的。

在添加這些虛構功能後，所設計出的演算法，被稱為***非決定性演算法***(non-deterministic algorithm)；相較之下，原來一般的演算法，就稱為***決定性演算法***(deterministic algorithm)。利用非決定性演算法，我們定義出另一個集合 NP，來討論目前尚無法歸入集合 P 的難題。當一個決策問題，存在一個 $O(n^k)$時間複雜度的非決定性演算法時，則稱此問題落在 NP 的集合中。集合 P 和集合 NP 在本章中將扮演重要的角色。



以下，先設計一個非決定性演算法來解決搜尋問題。

表 10.1 搜尋問題的非決定性演算法

輸入	存入 $A[1:n]$的 n 個值，其中 $n \geqq 1$
輸出	x 值在 $A[1:n]$中嗎?

步驟	`Algorithm search (A, x)`
	`{`
	`Step 1: j:=choice (1, n);     /*利用 choice 直接猜中 x 的位置 j*/`
	`Step 2: if A(j)=x then {write (j); success ()};/*檢查是否 x 在 A(j)上*/`
	`Step 3: write (0); failure ();/*因 x 不在 A()上，輸出搜尋失敗訊息*/`
	`}`

上述的演算法，因為 Step 1 中使用 choice()，故為非決定性演算法。其時間複雜度顯然為 $O(1)=O(n^0)$；因此，搜尋問題落入 NP 中。相對地，設計出一個決定性演算法，透過掃描陣列來解此搜尋問題時，可能需要 $O(n)$時間；因此，搜尋問題也落入 P 中。

相較於非決定性演算法，決定性演算法常需要較多的時間，來解同一個問題。NP 中的問題，不會比 P 還要少(主要是因為在 $O(n^k)$時間內 NP 可以選擇使用擴增的功能 choice()的緣故)。簡言之，NP 包含 P (圖 10.2)。

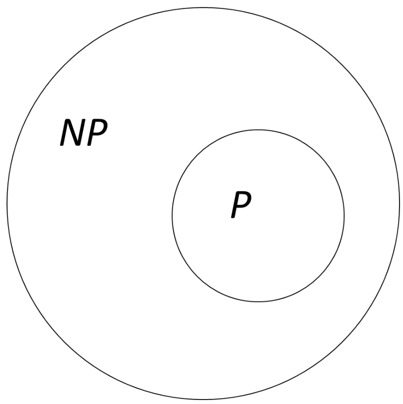


圖 10.2 *NP* 包含 *P*

「NP 內的問題都很難嗎？」

「不！NP 中也有簡單的問題，例如搜尋問題就是其一。」

「NP 比 P 大嗎？」

「NP 看起來比 P 大呀！NP 包含 P 呀！」

「有沒有可能 NP=P？」

「不知道！可否告訴我答案？」

「我也不知道！」

「蝦米！那誰會知道呢？」

「目前全世界並沒有任何一人，可以證明 NP=P 或 NP≠P。」

10.3 滿足問題

接下來介紹的***滿足問題*** (satisfiability problem，簡稱 ***SAT***)，就是一個 *NP* 中的典型難題。

表 10.2 滿足問題

問題	令 x_1，x_2，⋯，x_n 代表布林變數(boolean variables)(其值非真(true)即假(false)的變數)。令$-x_i$ 代表 x_i 的相反數(negation)。一個布林公式是將一些布林變數及其相反數利用而且(and)和或(or)所組成的表達式。滿足問題是判斷是否存在一種指定每個布林變數真假值的方式，使得一個布林公式為真
輸入	一個 n 個變數的布林公式 $(-x_1 \vee -x_2 \vee x_3) \wedge (x_1 \vee x_4) \wedge (x_2 \vee -x_1)$
輸出	是否存在一種指定每個布林變數真假值的方式，使得此公式為真？ 是(當 x_1=真，x_2=真，x_3=真，x_4=真時，此公式為真)

我們很容易地設計一個暴力法(brute force method)來解決滿足問題。此暴力法列出所有 n 個變數 2^n 不同的真假組合，並代入此式中，檢測是否為真即可。可惜，此演算法的時間複雜度為 $O(2^n)$，因此目前不能將滿足問題列入 *P* 集合中。相對地，以下將設計出一個非決定性演算法，並說明滿足問題可列入 *NP* 集合中。

表 10.3 滿足問題的非決定性演算法

輸入	一個 n 個變數的布林公式 $E(x_1, \cdots, x_n)$
輸出	是否存在一種指定每個布林變數真假值的方式，使得此公式為真

步驟	`Algorithm satisfiability (E(x1, ⋯ , xn))`
	`{`
	`Step 1: for i=1 to n do`
	`        xi←choice (true, false) /*利用 choice 直接猜中 xi 的真假值*/`
	`Step 2: if E(x1, ⋯ , xn) is true then success () /*計算此布林公式是否為真*/`
	`        else failure ();`
	`}`

上述非決定性演算法的時間複雜度為 $O(m+n)$。其中利用 $O(n)$時間來猜測 n 個變數的值，及 $O(m)$時間來計算，長度為 m 的布林公式 $E(x_1, \cdots, x_n)$是否為真。因此，滿足問題落入 NP 中。針對滿足問題而言，此非決定性演算法比上述的暴力法(決定性演算法)，減少了不少執行時間。總結，滿足問題這個難題目前不在 P 中，但是在 NP 中(圖 10.3)。

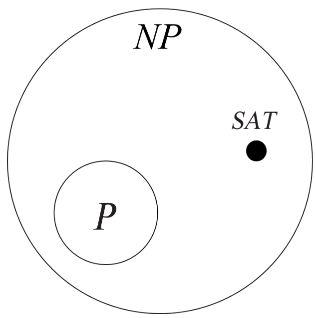


圖 10.3　滿足問題落於 *NP* 中

10.4 多項式時間轉換

面對一個演算法的題目時，可以利用問題轉換(請見第七章)將目前的問題，轉換成另一個問題。本節關心的問題轉換技巧，其所需要轉換的時間皆需在多項式時間(即 $O(n^k)$)內完成。利用此多項式時間的轉換，我們可以將 NP 中的難題建立起一些有趣的關係。

針對兩個問題 A 和 B，如果存在一個 $O(n^k)$時間的(決定性)演算法，將每一個問題 A 的輸入轉換成問題 B 的輸入，使得問題 A 有解時，若且惟若，問題 B 有解。此關係被稱為，問題 A ***轉換成***(reduce to)問題 B，可表示成 $A \propto B$。

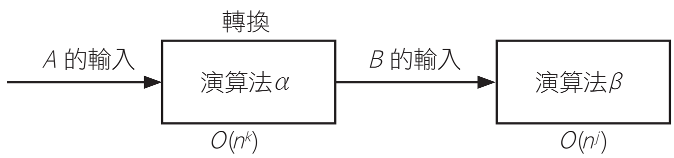


圖 10.4　多項式時間問題轉換($A \propto B$)

當 $A \propto B$ 時，若設計出一個 $O(n^j)$時間的決定性演算法β來解決問題 B 時，則立即存在一個多項式時間的決定性演算法，可解決問題 A。原因是，我們可以先利用演算法α(需時間 $O(n^k)$)將問題 A 轉換成問題 B，接著執行演算法β(需時間 $O(n^j)$)，因此共執行 $O(n^k)+ O(n^j)$的多項式時間(圖 10.4)。

若以設計出一個多項式時間演算法為主要目的而言，在多項式時間內將 A 轉換成問題 B，有一個好處；即我們可以直接解決問題 A，或選擇解決問題 B，間接地解決問題 A。因此，問題 A 和 B 似乎出現了一個依賴關係；即「解決 B」→「解決 A」的關係。注意此處「解決」是指此問題存在一個 $O(n^k)$時間的決定性演算法。

10.5 NP 中的難題

「若將演算法問題，譬喻成打擊電動遊戲中的怪獸，
NP-complete 就算是 NP 中最難纏的怪獸了。」

首先，定義一個名詞，常用來代表一群目前未被有效解決(指未落入 P 中)的演算法問題。

一個問題 L 被稱為是 ***NP-hard***，若且惟若，滿足問題轉換成 L(即滿足問題$\propto L$)。

我們已經知道了滿足問題是 *NP* 中的難題，而 *NP-hard* 的問題則是滿足問題衍生(轉換)出來的。

「換句話說，滿足問題這一隻怪獸進化成另一隻 NP-hard 怪獸。」

反之，若能有效地解決 *NP-hard* 的問題，就可以有效地解決滿足問題；即如果 *NP-hard* 的問題落入 *P* 中，則滿足問題也落入 *P* 中(圖 10.5)。

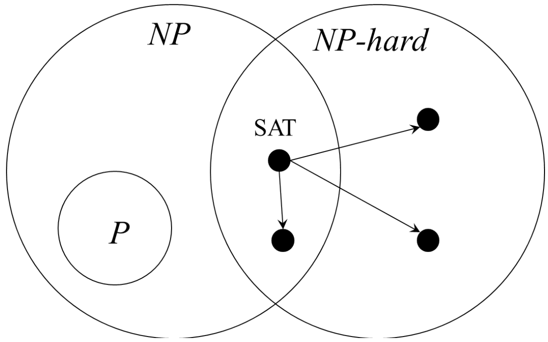


圖 10.5　*NP-hard* 的問題是自滿足問題(*SAT*)轉換過來，但不一定落於 *NP* 中

NP-hard 的問題雖然轉換自 *NP* 中的滿足問題，但不一定全部落於 *NP* 中(圖 10.5)。接下來，我們要討論的是同時落在 *NP* 和 *NP-hard* 中的問題。

一個問題 L 被稱為是 ***NP-complete***，若且惟若，$L \in NP$ 而且 $L \in NP\text{-}hard$。

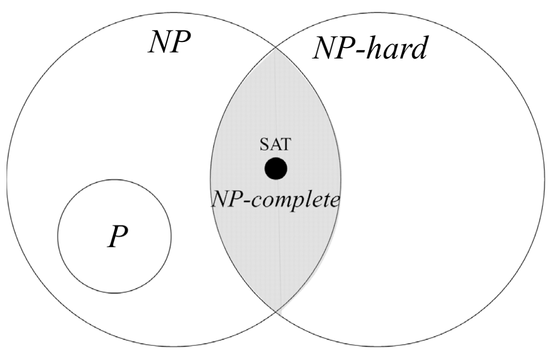


圖 10.6　*NP-complete*(兩集合交集處)是 *NP* 中的難題

NP-complete 的問題轉換自滿足問題，且落於 *NP* 中(圖 10.6)；可以想像成，在 *NP* 中，與滿足問題為同等級難度的怪獸。同樣地，若能有效地解決 *NP-complete* 的問題，就可以有效地解決滿足問題；即如果 *NP-complete* 的問題落入 *P* 中，則滿足問題也落入 *P* 中。

簡言之，*NP-complete* 的問題也是 *NP-hard* 的問題，兩者都是轉換自滿足問題。差別是 *NP-complete* 的問題必須在 *NP* 中(圖 10.6)。

「P 集合圈養的是較容易收服的怪獸，NP-complete 和 NP-hard 兩集合養的怪獸，都是滿足問題進化而成的，目前並未被人類收服；但是，若動用 choice()這一神器，NP-complete 內的怪獸立即投降。可惜的是，人類在現實生活上買不到這一神器。」

截至目前為止，人類尚未設計出一個 $O(n^k)$時間複雜度的(決定性)演算法，來解決任何一個 *NP-complete* 或 *NP-hard* 內的問題。更不幸地是 *NP-complete* 或 *NP-hard* 所涵蓋的問題，幾乎是遍佈所有資訊領域。以下介紹三個 *NP-complete* 的範例：

1. **圖塗色(graph k-colorability)問題**：此問題需回答「任意一個輸入的圖中，是否存在一種塗色方法，利用 k 個顏色，使得相鄰點上的顏色是不同的」。當輸入圖如圖 10.7 且 k=2 時，答案為「否」。然而，當輸入圖如圖 10.7 且 k=3 時，答案為「是」。

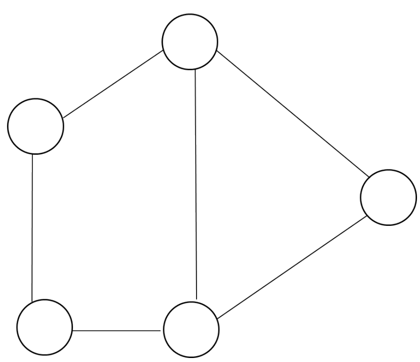


圖 10.7 圖塗色問題範例

2. **漢米爾路徑(Hamiltonian circuit)問題**：一個漢米爾路徑為圖上的一條路徑，此路徑需經過每一個點剛好一次，且形成圈圈(circuit)。漢米爾路徑問題即是回答「任意一個輸入的圖中，是否存在一條這樣的路徑」。當輸入圖為圖 10.8 時，其答案為「是」。

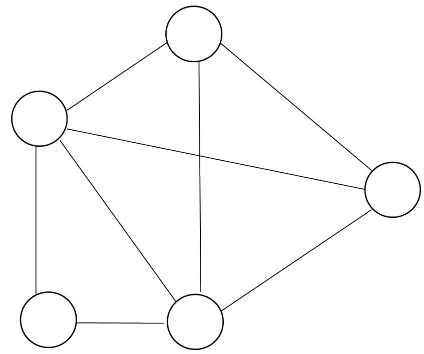
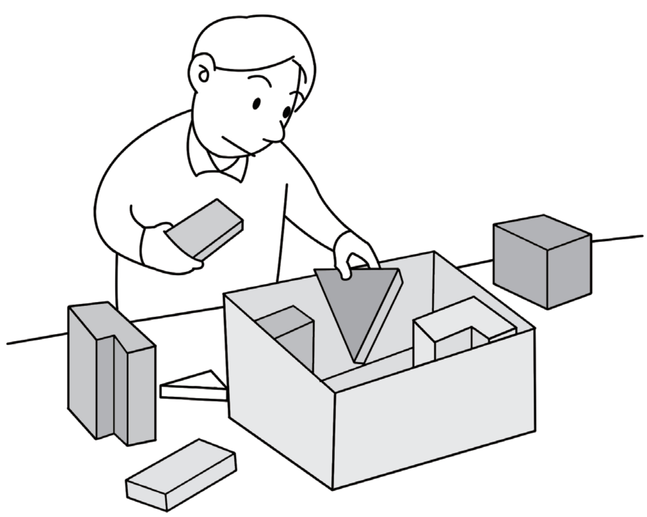


圖 10.8 漢米爾路徑範例

3. **切割(partition)問題**：輸入一個正整數的集合。切割問題即是回答「是否可以將集合切割成兩個小集合，使得每個小集合的總和是相同的」。當輸入的集合為{2, 3, 4, 6, 9}時，其答案為「是」；因為{2, 4, 6}的和與{3, 9}的和是相同的。



如何將一些東西裝到固定的箱子中, 也是一個 NP-complete 的難題之一

10.6 NP-complete 的性質

根據之前的討論，我們應有以下的認識：

1. 想要有效率地(指擁有多項式時間 $O(n^k)$時間複雜度的決定性演算法)解決一些 *NP-complete* 的難題，目前無法做到。

2. 因為利用 choice()，集合 *NP* 可以包含一部份這樣的難題(如滿足問題)。

3. 多項式時間轉換的關係，可以被用來討論這些難題之間的關係。

4. 滿足問題可以，在多項式時間內，轉換成 *NP-hard* 和 *NP-complete* 中的任一難題(圖 10.9)。

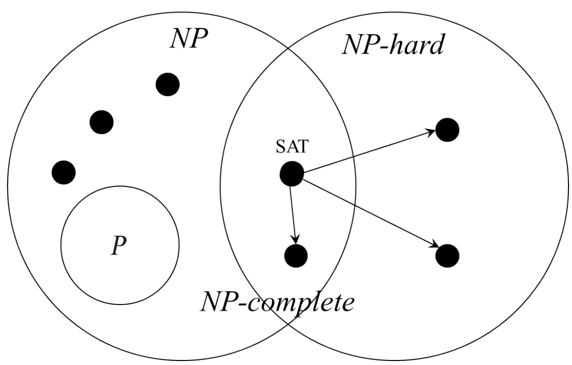


圖 10.9 滿足問題可以，在多項式時間內，轉換成 *NP-hard* 和 *NP-complete* 中的任一難題

在本節中，更多的 *NP-complete* 問題的性質將會被提及。史蒂芬庫克(Stephen Cook)證明了一個十分重要的性質：

性質(A)：「任一個 *NP* 內的問題都可以，在多項式時間內，被轉換成滿足問題。」

因此，滿足問題可以說 *NP* 內最難的問題之一(圖 10.10)。

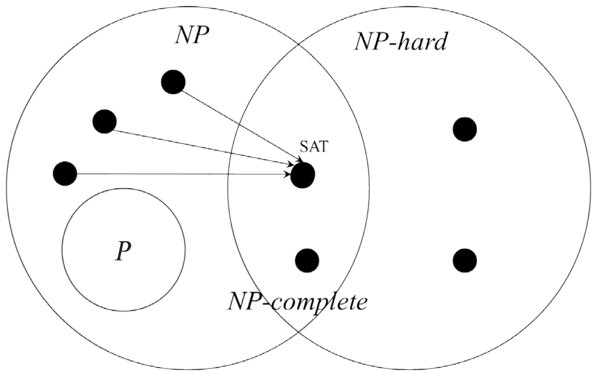


圖 10.10 任一個 *NP* 內的問題都可以，在多項式時間內，被轉換成滿足問題 *SAT*

因為滿足問題可以轉換成 *NP-hard* 和 *NP-complete* 中的任一問題，利用性質(A)我們知道，任一個 *NP* 內的問題可以，在多項式時間內，轉換成任一 *NP-hard* 或 *NP-complete* 中的問題。因此，我們可以得到性質(B)及性質(C)。

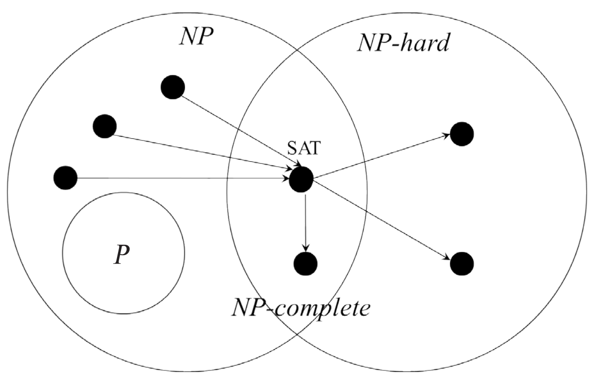


圖 10.11 任一個 *NP* 內的問題可以被轉換成任一個 *NP-complete* (或 *NP-hard*)問題

性質 (B)：「任一個 *NP* 內的問題都可以，在多項式時間內，被轉換成任一個 *NP-complete* 問題。」
性質 (C)：「任一個 *NP* 內的問題都可以，在多項式時間內，被轉換成任一個 *NP-hard* 問題。」

根據以上性質，若是 *NP-complete* (或 *NP-hard*)的其中一個問題被有效率地解決了(指設計出一個多項式時間 $O(n^k)$時間複雜度的決定性演算法)，則 *NP* 中的所有問題都將有一個有效率的解。換句話說，當 *NP-complete* 或 *NP-hard* 的其中一個問題落入 *P* 中，則 *NP* 中的所有問題都落入 *P* 中。

「再次將演算法問題，譬喻成打擊電動遊戲中的怪獸。若能解決 NP-complete 和 NP-hard 兩集合中的其中一隻怪獸，則 NP（含 NP-complete）中的全部怪獸則被消滅殆盡。」

因為滿足問題是 *NP-complete*，故可推論性質(D)是正確的。

性質(D)：「滿足問題在集合 *P* 中，若且唯若，$P=NP$。」

相反地，若存在一個 *NP-complete* 問題被證明不會落入 *P*，則所有 *NP-complete* 問題都不會落入 *P* 中(此時 $P \neq NP$)。

「NP-complete 中的怪獸具備有同生共死的特性；即 NP-complete 中的怪獸全部被消滅殆盡（指落入 P 中），或者 NP-complete 中的怪獸全部存活（指落入 P 之外）。」

「到底 NP 等不等於 P？」

「不知道！聽說目前世界上目前沒有人知道答案呢！」

「如果要挑戰此題目，您會怎麼做？」

「為其中一個 NP-complete 問題，設計出一個 $O(n^k)$ 時間複雜度的演算法；或者，證明其中一個 NP-complete 的問題絕對不存在此演算法。」

「為何只需考慮一個問題就可以了？」

「因為 NP-complete 的問題彼此是好朋友，同生共死，絕不苟活。」

「問題好像變簡單了。你的敵人只剩下一個了？」

「不過，這個敵人代表著千千萬萬個敵人呢！」

10.7 NP-complete 的證明技巧

「這個程式超難寫的，怎麼寫都跑不快！」

「會不會是 NP-complete？」

「蝦米！不會吧？」

「怎麼證明一個問題是 NP-complete？」

「不知道！」

「NP-complete 的定義是甚麼？」

「落在 NP 而且落在 NP-hard 中。」

「怎麼證明一個問題落在 NP 中？」

「設計一個花費 $O(n^k)$ 時間的非決定性演算法來解決此問題。」

「怎麼證明一個問題落在 NP-hard 中？」

「將滿足問題轉換成此問題！」

「轉得過去嗎？」

「這個…」

其實，證明一個決策問題是 *NP-hard*，不一定需自滿足問題轉換。較可行的方式是，從任何一個已經被證明為 *NP-hard* 的問題下手即可。

理由是當您將一個 *NP-hard* 的問題β，轉換成您關心的問題α後，我們可以推論問題α也是 *NP-hard*。因為問題β 是 *NP-hard*，故滿足問題必可轉換成問題β，再加上之前發現的轉換，滿足問題就可以輾轉轉換成問題α，因此問題α也成為 *NP-hard*。如圖 10.12 所示。

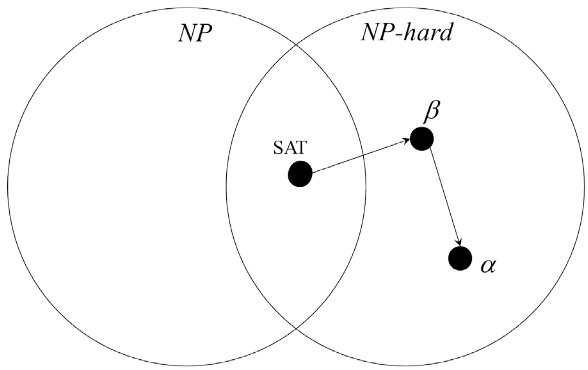
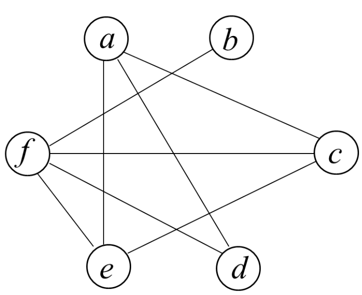


圖 10.12 從任何一個 NP-hard 問題*(β)*來證明另一個新問題*(α)*為 *NP-hard*。

以下，我們將以點覆蓋問題(vertex cover problem)(表 10.4)來說明此證明技巧。

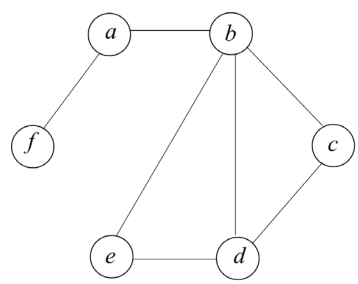
表 10.4 點覆蓋問題

問題	網路攻擊事件最近發生頻仍，造成網路管理員極大的困擾。網路管理員欲在一些網路路由器(router)上佈署追蹤器(tracker)，來記錄追蹤不正常的封包，希望找到攻擊者的網路所在。這些追蹤器需能夠直接監控到每一條網路線
輸入	一個圖 $G=(V, E)$代表網路路由器及其連接，k 則代表欲佈署追蹤器的個數 $G=(V=\{a, b, c, d, e, f\}, E=\{(a, c), (a, d), (a, e), (b, f), (c, e), (c, f), (d, f), (e, f)\})$，$k=3$  next

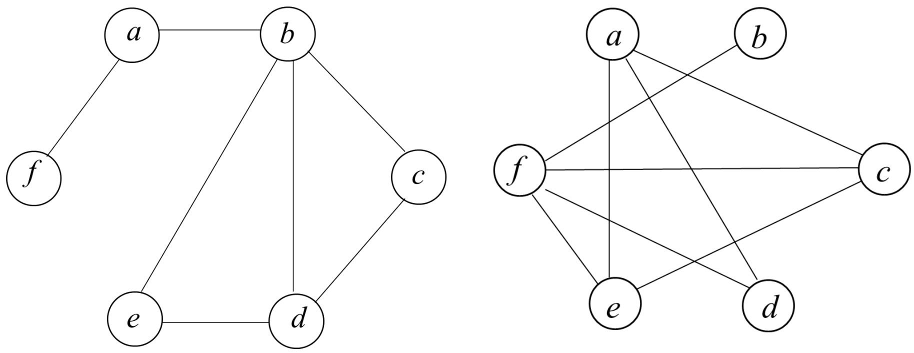
輸出	是否存在一個 V 的子集合 S 其｜ S ｜$=k$，使得每一條 E 中的線的其中一個端點 (endpoint)，需落在 S 中？ 是 ($S=\{a, f, c\}$)

已知 clique 問題(表 10.5)是 *NP-hard*。接下來，我們準備將 clique 問題轉換成點覆蓋問題，藉以證明點覆蓋問題也是 *NP-hard*。

表 10.5 clique 問題

問題	部分網路連接的線路斷裂，有時會造成通訊無法連接。我們想要在一個網路中，找出一個大小為 k 的子網路，並期望此子網路緊密連接不易分裂。因此，希望其中的任兩個節點，都有網路線直接連接。請設計一個程式協助找出這樣的子網路
輸入	一個圖 $G=(V, E)$，k 為一正整數 $G=(V=\{a, b, c, d, e, f\}, E=\{(a, b), (a, f), (b, c), (b, d), (b, e), (c, d), (d, e)\})$，$k=3$ 
輸出	在 V 的子集合上，是否存在一個大小為 k 的 clique？此處 V 的子集合 S 被稱為 clique 如果 S 中的任兩點在 G 上都有連線 是 ($S=\{b, e, d\}$)

此轉換的重點在於，將 clique 問題的輸入$\{G=(V, E), k\}$，轉換成點覆蓋問題的輸入$\{G'=(V', E'), k'\}$。轉換的方法十分簡單，新圖的點集合和原來的相同；不同的是當原來的圖上兩點有線，則新的圖不需連線；反之，當原來的圖上兩點無線，則新的圖需連線(圖 10.13)。也就是兩個圖在點集合上是相同的，但在線集合上是互補有無的；故稱 G' 為 G 的***互補圖***(complement graph)。最後，令 $k'=|V|-k$，則轉換完成。注意此轉換最多需要 $O(n^2)$ 的時間。這裡的 n 是點集合 V 的個數。因此，是一個多項式時間的轉換(圖 10.13)。



G=(*V*, *E*)，*k*=3　　*G'*=(*V*, *E*)，*k'*=3

圖 10.13 將 clique 問題轉換成點覆蓋問題

「在 *G* 中找得到大小為 3 的 clique 嗎？」

「不難吧？{b, d, e}就是。」

「所有點在刪除{b, d, e}後的點集合，在 *G'* 中有何特性？」

「刪除{b, d, e}後的點集合是{a, c, f}。{a, c, f}是啥？」

「在 *G'* 中{a, c, f}是點覆蓋嗎？」

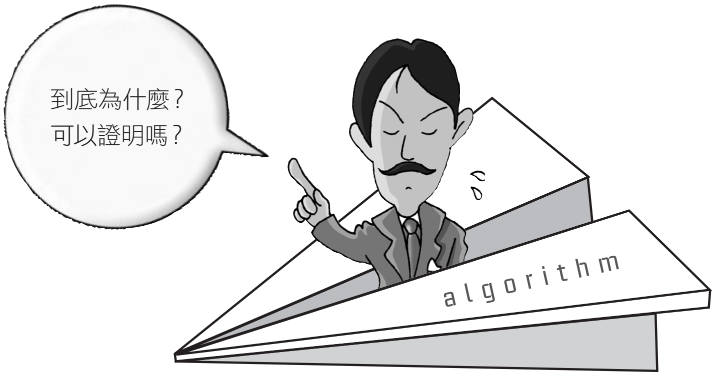
「所有的線都黏到 a 或 c 或 f，真是點覆蓋！」

「為什麼？」

「太神奇了。不知道耶！」

「試試看其他的 clique 也對嗎？」

「刪除 *G* 中的 clique {b, c, d}後的點集合是{a, e, f}，所有 *G'* 的線都黏到 a 或 e 或 f，也是點覆蓋！」



此轉換的正確性可參考表 10.6。最後，我們也藉由 clique 問題，證明了點覆蓋問題也是 *NP-hard*。

表 10.6 clique 問題轉換成點覆蓋問題之正確性論述

圖 $G=(V, E)$存有一個 clique 其大小為 k，若且唯若，圖 $G'=(V', E')$ 存有一個點覆蓋其大小為 $\|V\|-k$。
證明： (1) 當圖 $G=(V,E)$ 存有一個 clique $V^* \subseteq V$其大小為 k 時，則 $V-V^*$在互補圖 G' 中是一個點覆蓋。 利用矛盾證明法，假設 $V-V^*$ 在互補圖 G' 中不是一個點覆蓋，則在 G'上必有一條線，其兩端點 v_1, v_2 必落在 V^* 中。因為 G' 為 G 的互補圖，故在圖 G 中 v_1 和 v_2 必不相連，然在圖 G 中 v_1, v_2 都是 clique V^* 中的兩點，故在圖 G 中 v_1 和 v_2 必相連，如此產生矛盾。因此，先前的假設錯誤。因此 $V-V^*$ 在互補圖 G' 中是一個點覆蓋。 (2) 當圖 $G'=(V', E')$ 存有一個點覆蓋 $V^* \subseteq V'$ 其大小為 $\|V\|-k$，則 $V-V^*$ 在圖 G 中是一個大小為 k 的 clique。 因為 V^* 在 G' 中是一個點覆蓋，因此 E'所有的線其兩端點必定有其一落在 V^* 中。換句話說，$V-V^*$中的任兩點必在 G 中是不連接的。又因 G' 為 G 的互補圖，故在圖 G 中 $V-V^*$ 中的任兩點必是連接的，最後可知 $V-V^*$ 是一個 clique 且大小為$\|V-V^*\|=\|V\|-\|V^*\|=\|V\|-(\|V\|-k)=k$。

學習評量

1. **排列**：一字串"abc"的所有排列是{"abc", "acb", "bac", "bca", "cab", "cba"}，按照字母順序排列並給予編號為

```
0 "abc"
1 "acb"
2 "bac"
3 "bca"
4 "cab"
5 "cba"

輸入
abc            (排序後一字串)
3              (一整數 n)

輸出
bca            (編號為 n 的字串)
```

2. **尤拉 Totient 函數**：尤拉 Totient 函數$\varphi(m)$代表{1, …, m}中有多少的數和 m 互質?例如，$\varphi(1)=1$，$\varphi(2)=1$，$\varphi(3)=2$，$\varphi(4)=2$，$\varphi(5)=4$，$\varphi(6)=2$，$\varphi(7)=6$。請寫一個程式，當輸入 m 時，輸出其尤拉 Totient 函數$\varphi(m)$。

```
輸入
13
8

輸出
12
4
```

3. 是非題：請填入答案 ○ 或 × (若答案不對，請說明之)。

① (　　) NP 的問題，無法寫出程式來解決之。

② (　　) 所有 P 的問題，不見得可寫程式找到解。

③ (　　) NP 不等於 P。

④ (　　) NP=P。

⑤ (　　) NP-complete 不是 NP 的問題。

⑥ (　　) NP-complete 不是 NP-hard 的問題。

⑦ (　　) NP-hard 的問題是 NP 中特別難的問題。

⑧ (　　) NP-hard 的問題都可轉換成 SAT 問題。

⑨ (　　) 無法寫程式有效解決的問題就是 NP-hard 或 NP-complete。

⑩ (　　) 若一個 NP 的問題被有效地解決，則 NP=P。

Memo