6 樹搜尋法

章節大綱

6.1　何謂樹搜尋法？

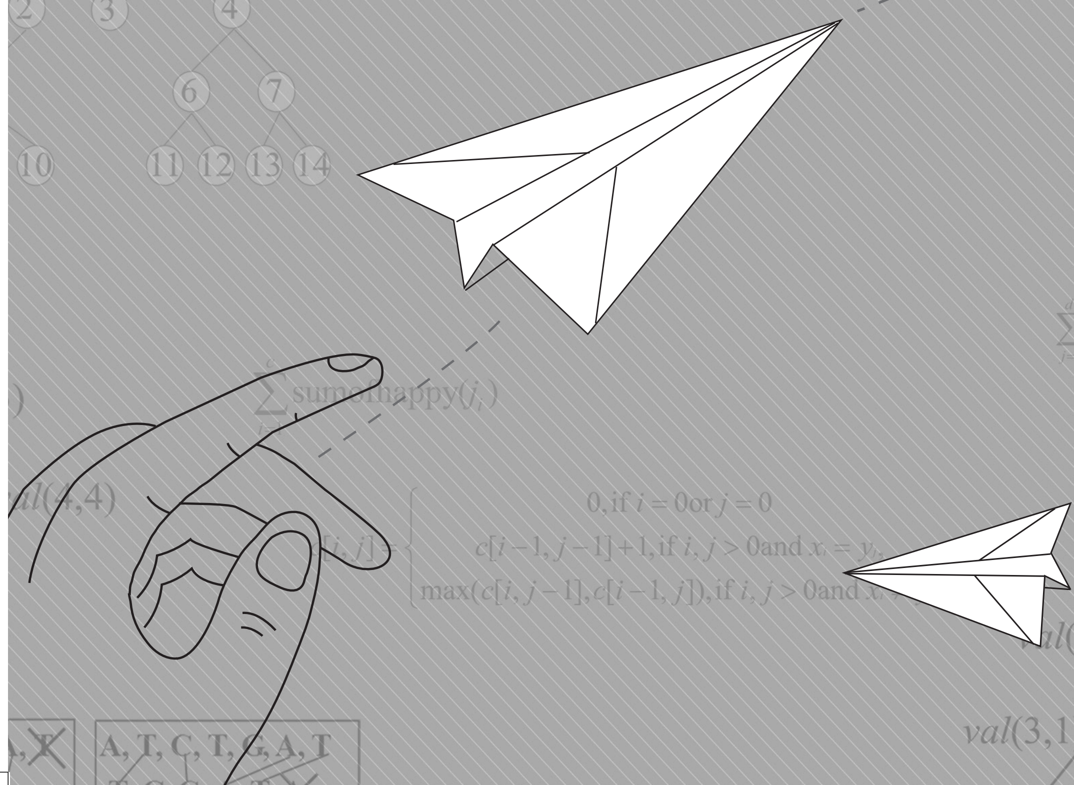
6.2　樹狀解空間：n 個皇后問題

6.3　撤退法：塗色問題

6.4　寬度優先搜尋法：八數字謎題

6.5　加速技巧：旅行推銷員問題

6.6　樹搜尋法的技巧



6.1 何謂樹搜尋法？

「什麼是樹搜尋法(tree searching)？」

簡言之：「將問題的解空間想像成一棵樹。求解的過程如同在這棵樹上搜尋答案。」

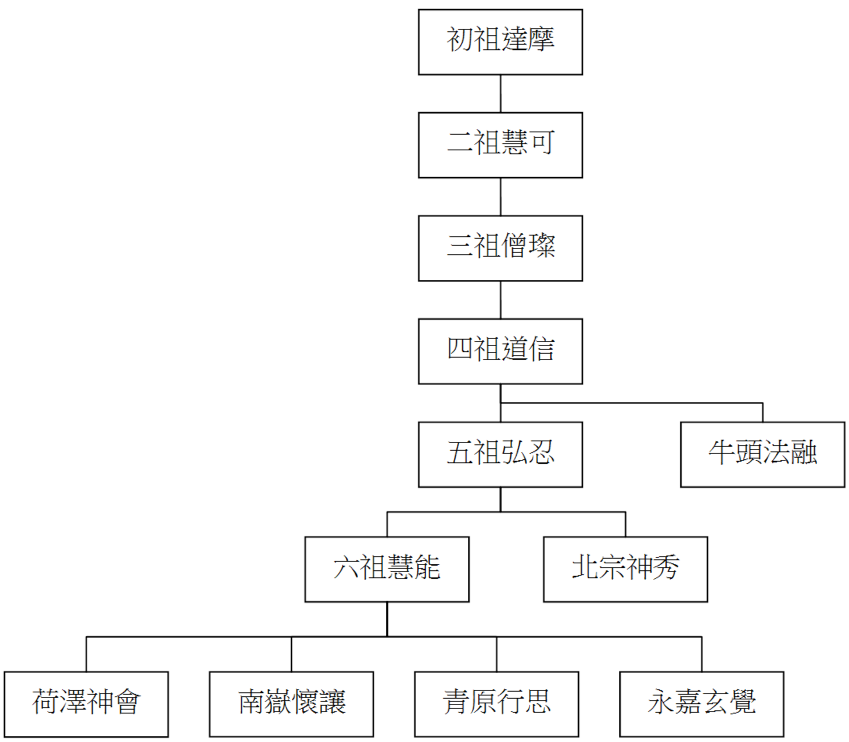


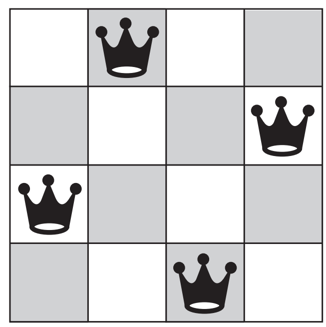
圖 6.1 禪宗法脈傳承

許多問題的解空間可以被表示成一棵樹。譬如禪宗法脈傳承、生物的演化或人類家族的繁衍都可以利用樹來展現。在樹上搜尋解的過程，如同在族譜上找尋自已的親人一般地自然。

6.2 樹狀解空間：*n* 個皇后問題

第一個例子是 n 個皇后問題。

表 6.1 *n* 皇后問題

問題	將 n 個皇后置放在一個 $n \times n$ 的棋盤上，使得彼此不會相互攻擊。也就是，沒有兩個皇后被放在同一列或同一行或同一個對角線上
輸入	皇后的個數 n n=4
輸出	輸出 n 個皇后放在 $n \times n$ 的棋盤上的位置，使得彼此不會相互攻擊 右圖是四個皇后的可能置放位置(♛ 代表皇后) 

若將所有皇后按照橫列的順序編號，此處的 X_i 代表第 i 個皇后，置於該列自左開始計算的位置(即 $1 \leqq i \leqq 4$ 且 $1 \leqq X_i \leqq 4$)。例如，四皇后問題的其中一解，可表示成$(X_1, X_2, X_3, X_4)=(2, 4, 1, 3)$(請參考表 6.1 中的輸出)。

樹搜尋法的第一步驟，需將問題的解空間想像成一棵樹。下圖以一個四個皇后問題為範例(圖 6.2)。

這棵樹的第一層(level 1)，代表第一個皇后所放的行位置 X_1。這棵樹的第二層(level 2)，代表第二個皇后所放的行位置 X_2。類似地，這棵樹的第 i 層(level i)，代表第 i 個皇后所放的行位置 X_i。任意自這棵樹的樹根(root)走到樹葉(leaf)的一條路徑，就代表四個皇后可能的置放位置。因為，同一行是不能置放兩個皇后；即從樹根走到樹葉的任一條路徑上的值，不會重複出現。

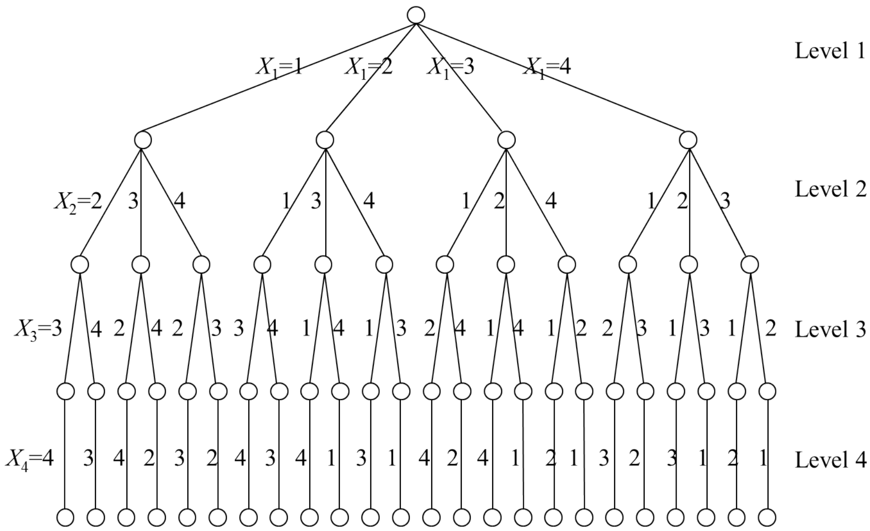


圖 6.2 *n* 皇后問題的解空間被想像成一棵樹(當 *n*=4)

如果是利用深度優先搜尋法(depth-first search)的方式在這棵樹上搜尋，就被稱為***撤退法***(backtracking)。如圖 6.3 所示，圖中樹上的數字代表此搜尋的順序。

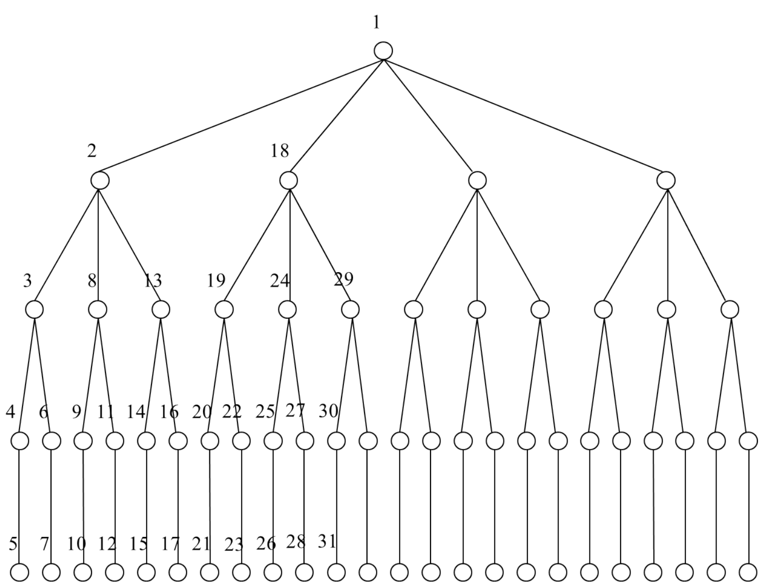


圖 6.3 在 4 個皇后問題的樹狀解空間上，利用撤退法搜尋的前 31 個步驟

在圖 6.3 上，撤退法自編號 1 節點開始，按照數字的大小順序搜尋解答。當拜訪第 31 個節點時，找到第一個四個皇后的解。此時，從樹根走到樹葉的一條路徑為 1-18-29-30-31，其對應到圖 6.2 的解為(X_1, X_2, X_3, X_4)=(2, 4, 1, 3)。此解正好是表 6.1 中四個皇后的置放位置。

表 6.2 四個皇后的撤退法

輸入	(無)
輸出	X(1:4)

步驟	Algorithm four_queen_backtrack
	{
	integer k, X(1:4) //k 代表目前考慮的皇后的編號；
	　　　　　　　　　//X(1:4)則儲存四個皇后的行數
	
	X(1)←0; k←1　　　//設定目前考慮的皇后為第 1(k=1)皇后，其行位置為 0(X(1)=0)
	while k>0 do
	{
	X(k)←X(k)+1　　　//將第 k 個皇后的位置移到下一行
	
	while X(k)≤4 而且不可安全地置放此皇后時　//當不能放置時考慮下一行
	do X(k)←X(k)+1　　　//直到找到或 X(k)=5 為止
	
	　if X(k)≤4 then　　　//當目前考慮的皇后找到位置時
	　　if k=4 then 輸出 X(1:4)　　//當找到全部四個皇后的解時，輸出其位置
	　　else {k←k+1; X(k)←0}　　//否則尋找下一個皇后的位置
	　else k←k-1　//目前考慮的皇后無法在此列找到合適的位置時，
	　　　　　　　//則撤退回去，即繼續考慮上一個皇后的下一行的位置
	}
	}

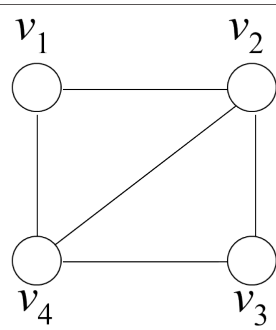
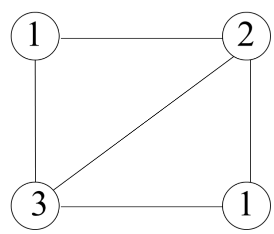
注意上述的演算法並未曾造出一棵樹後，再於這棵樹上搜索解答。也就是樹搜尋法用樹來代表解空間，可能祇是想像中的資料結構。

一般而言，一個問題的解空間，可能使用不同種樹來表示。當然，就算在同一棵樹上，也可以有不同的搜尋順序，下列各節將介紹常用的搜尋方式。

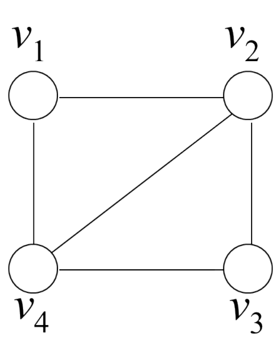
6.3 撤退法：塗色問題

第二個例子是塗色問題，也是一個可以利用撤退法來解題的例子。

表 6.3 塗色問題

問題	這是一個塗顏色的遊戲。遊戲的內容是利用 k 個顏色，將一個圖的點(vertex)塗上顏色，使得相鄰兩個點的顏色不同
輸入	一個圖 G 及顏色數 k k=3 
輸出	利用 k 個顏色，將輸入圖的點塗上顏色，使得相鄰兩個點的顏色不同 以下是當 k=3 的一種塗色 

同樣地，首先將塗色問題的解空間想像成一棵樹。下圖是以表 6.3 中的輸入圖為範例。此處的 X_i 代表點 v_i 的顏色編號(即 $1 \leqq i \leqq 4$ 且 $1 \leqq X_i \leqq 3$)。



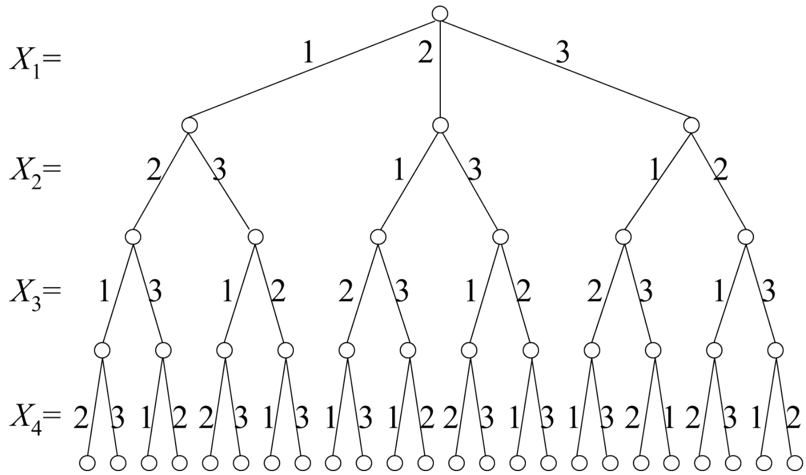


圖 6.4　四點圖(使用三個顏色)的塗色問題其解空間可以被想像成一棵樹

同樣地，在這棵樹上作深度優先搜尋所得到的解為 $(X_1, X_2, X_3, X_4)=(1, 2, 1, 3)$ (圖 6.4)。

表 6.4 塗色問題的撤退法

<table>
<tr><td>輸入</td><td>一個圖 G 及顏色數 k</td></tr>
<tr><td>輸出</td><td>利用 k 個顏色，將輸入圖上的點塗上顏色，使得相鄰兩個點的顏色不同</td></tr>
<tr><td>步驟</td><td>

```
Algorithm Graph_Coloring(i)
{
//X[i]儲存點 v_i 的顏色(1≤i≤n);起始值皆為 0 (代表未被塗上顏色)
//參數 i 是下一個被塗色的點之編號
//演算法以執行 Graph_Coloring[1]開始
{
Repeat
{
呼叫 Next_Value(i);   //呼叫 Next_Value(i)以取得 X[i] 的下一個顏色
if (X[i]=0) then return; //無顏色可用於點 v_i 時，則撤退回到上一個點 v_{i-1}
if (i=n) then write (X[1:n]); //如果 n 點都被塗上顏色，則輸出各點的
                                顏色
else Graph_Coloring(i+1);   //否則呼叫 Graph_Coloring(i+1)以塗下
                              一點 v_{i+1}
} until (false);
}
```

next</td></tr>
</table>

步驟	

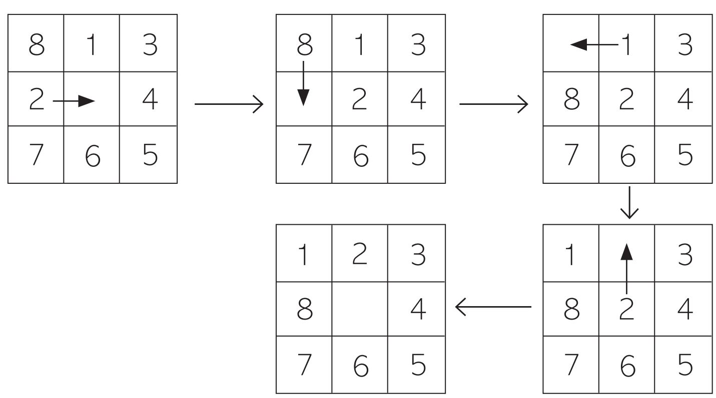
	Algorithm Next_Value(*i*)
	//回傳點 *i* 的顏色
	{
	    Repeat
	    {
	    X[*i*]=*X*[*i*]+1 mod (*k*+1); //試一試為點 v_i 塗上下一個顏色
	    if (*X*[*i*]=0) then return; //若全都試過並無顏色可用時，跳回
	    for *j*:=1 to *n* do //檢查點 *i* 的顏色是否與相鄰點 *j* 相同？
	    {
	    if((*G*[*i*,*j*]<>0)and(*X*[*i*]=*X*[*j*])) //若(點 v_i 和點 v_j 是鄰居)
	                                        且(點 v_i 和點 v_j 的顏色相同)，
	    then break; //則跳出此迴圈
	    }
	    if(*j*=*n*+1) then return; //如果確定與所有相鄰點使用不同的顏色
	                                    //則跳回
	    } until (false);
	}

注意在塗色問題的撤退法中，當無法順利為點 v_i 塗上顏色時(即所有顏色皆與其相鄰點衝突時)，則撤退回到上一個點 v_{i-1}，並繼續嘗試為點 v_{i-1} 塗上下一個顏色。注意，此演算法也並未建立一棵樹出來。

6.4 寬度優先搜尋法：八數字謎題

撤退法是在一個問題的解空間(表示成一棵樹)上作深度優先搜尋。當然在解空間中，也可採取不同的搜尋策略。以下介紹的就是，利用寬度優先搜尋法(breadth-first search)來解八數字謎題。以下先介紹八數字謎題。

表 6.5 八數字謎題

<table>
<tr><td>問題</td><td>一個 3×3 的格盤上，任意置放 1~8 的數字。其中有一個空
格未放數字。請每次移動一個數字使得最終的格盤如下：
<table>
<tr><td>1</td><td>2</td><td>3</td></tr>
<tr><td>8</td><td></td><td>4</td></tr>
<tr><td>7</td><td>6</td><td>5</td></tr>
</table>
</td></tr>
<tr><td>輸入</td><td>一個 3×3 的格盤上，任意置放 1~8 的數字
<table>
<tr><td>8</td><td>1</td><td>3</td></tr>
<tr><td>2</td><td>6</td><td>4</td></tr>
<tr><td>7</td><td></td><td>5</td></tr>
</table>
</td></tr>
<tr><td>輸出</td><td>每次移動一個數字，使得最後排出最終的格盤

</td></tr>
</table>

這個問題的解空間可表達成下面的一棵樹。

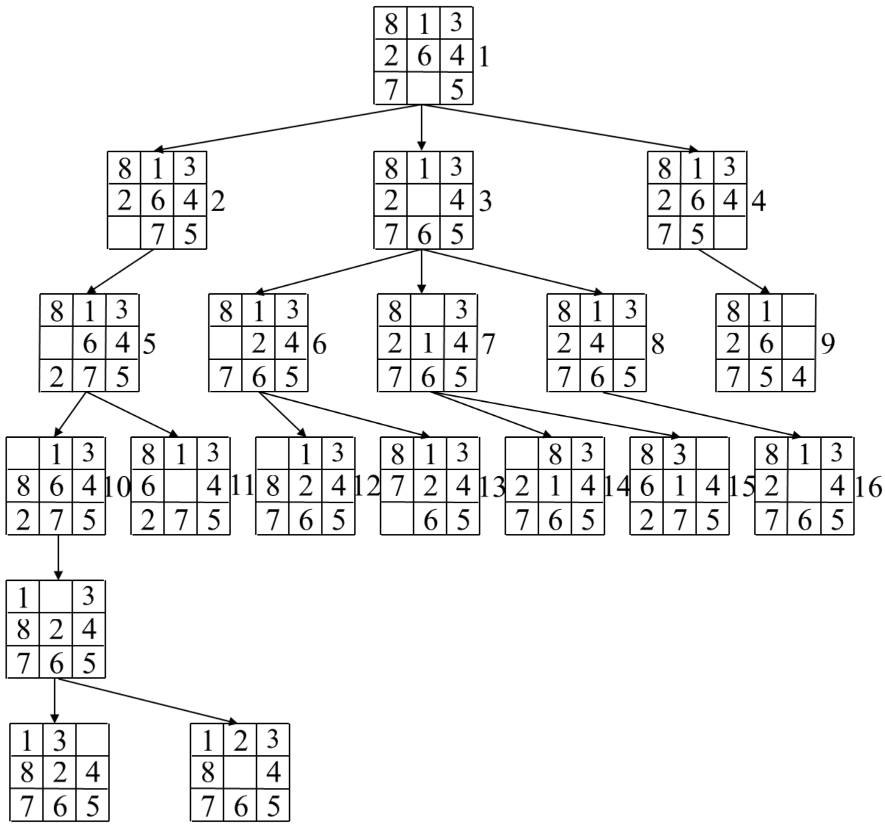


圖 6.5 八數字謎題的解空間被表示成一棵樹(格盤上的數字代表搜尋的順序)

當在這棵樹上作寬度優先搜尋時，其順序如上圖所示。注意寬度優先搜尋法規則為:在下一層的節點開始被搜尋前，目前這一層的節點，需先被搜尋完畢。圖 6.5 顯示出八數字謎題前 16 步驟的搜尋。

表 6.6 八數字謎題的的寬度優先搜尋法

輸入	一個 3×3 的格盤上，任意置放 1~8 的數字
輸出	每次移動一個數字，使得最後排出最終的格盤

步驟	Algorithm Eight_Puzzle
	//每個節點代表一個格盤，並將解空間想像成一棵樹。在此樹中，父格盤可移動一個數字以轉換成子格盤，但此數字必需迴避上次移動的數字，以加速找尋。//
	{
	Step 1: 將輸入的格盤加入 queue 中。
	Step 2: 檢查在 queue 中的第一份資料是否為最終的格盤。如果是，則停止。
	Step 3: 移除 queue 中的第一份資料。並將此資料的兒子加入 queue 的尾部。
	Step 4: 如果 queue 是空的，則輸出找尋失敗的訊息，並停止。否則，執行 Step 2。
	}

6.5 加速技巧：旅行推銷員問題

若想要在樹狀的解空間上作有效率地搜尋，其中的一個策略，就是儘量省略不必要的搜尋。我們將利用旅行推銷員問題(traveling salesperson problem)來說明這個技巧。

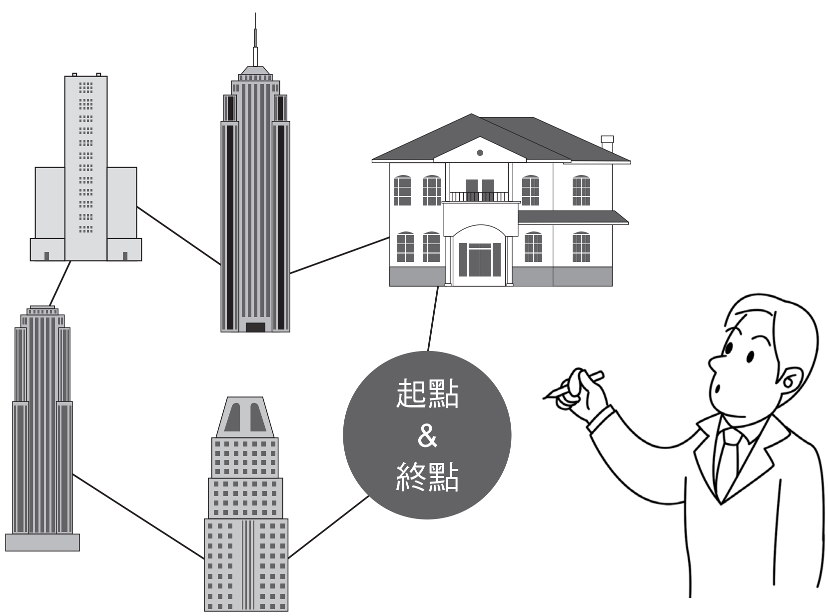
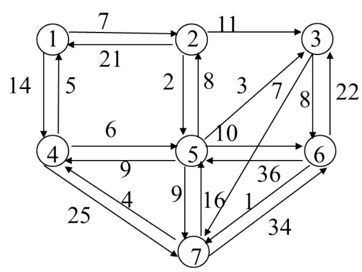


表 6.7 旅行推銷員問題

<table>
<tr><td>問題</td><td>有一位超級推銷員擁有許多客戶，而客戶分散於各地。在年終的時候，這位推銷員想要送禮給每一位客戶。在得知客戶之間所需的交通時間後，請幫這位推銷員找到一個花費最少時間的旅行路徑，使得每一個客戶剛好被拜訪一次，且最後需回到原出發客戶處</td></tr>
<tr><td>輸入</td><td>矩陣 $A[I, J]$儲存由客戶 I 直接到客戶 J 所需要的交通時間。矩陣 A 也可以被表示一個方向圖 G。此圖的兩點 I , J 代表兩位客戶的位置，而方向線$[I, J]$上面的成本(非負數)，代表由客戶 I 到客戶 J 所需要的交通時間(若客戶 I 不能直接到客戶 J 則 $A[I , J]=\infty$)


$A[7, 7]$
<table>
<tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>1</td><td>∞</td><td>7</td><td>∞</td><td>14</td><td>∞</td><td>∞</td><td>∞</td></tr>
<tr><td>2</td><td>21</td><td>∞</td><td>11</td><td>∞</td><td>2</td><td>∞</td><td>∞</td></tr>
<tr><td>3</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>8</td><td>7</td></tr>
<tr><td>4</td><td>5</td><td>∞</td><td>∞</td><td>∞</td><td>6</td><td>∞</td><td>25</td></tr>
<tr><td>5</td><td>∞</td><td>8</td><td>3</td><td>9</td><td>∞</td><td>10</td><td>9</td></tr>
<tr><td>6</td><td>∞</td><td>∞</td><td>22</td><td>∞</td><td>36</td><td>∞</td><td>1</td></tr>
<tr><td>7</td><td>∞</td><td>∞</td><td>∞</td><td>4</td><td>16</td><td>34</td><td>∞</td></tr>
</table></td></tr>
<tr><td>輸出</td><td>最少時間的旅行路徑
1 → 2 → 5 → 3 → 6 → 7 → 4 → 1
此路徑所需時間為 30</td></tr>
</table>

在一個樹狀解空間中搜尋一個最小成本的解時，如果已經推算出繼續搜尋某子樹所得到解的成本，一定會高於最佳解時，此刻顯然是可以停止搜尋此子樹。

「但是，尚未找到最佳解前，怎麼知道某些解的成本是高於最佳解的成本呢？」

「好像是喲！」

「一定要知道最佳解的值後，才能判斷一個可行解的成本是高於最佳解的成本嗎？」

「不知道耶！」

「譬如說，我們要找出全班同學中最矮的人(假設是 a)，在我們不知道 a 是誰之前，如何判斷 b 不是最矮的人？」

「只要 b 比任何一個人高，就一定不是最矮的人！」

「又譬如說，全班同學排成幾排，如果甲排最矮的都比乙排最高的人都還要高，請問那一排的人不會有最矮的人？」

「當然是甲排中不會有最矮的人！因為甲排最矮的都比乙排最高的高。」

換句話說，如果我們可以預估某一群解的成本下限(lower bound)為 x；相對地，我們也可以預估最佳解成本的上限(upper bound)為 y 時；當 $x>y$ 時，這一群解中必不存在最佳解(如此可省略不必搜尋這一群)，因為最佳解是可行解中最小成本的那一個。

從上面討論得知「預估某一群解的成本下限」及「最佳解成本的上限」，有助於在樹狀解空間中，有效率地搜尋。

我們將用以上的方法，來加速尋找旅行推銷員問題的最佳解。注意最佳解必須將每個點都剛好拜訪一次；也就是此旅行路徑需要進入且離開每個點一次。如表 6.7 的最佳路徑為 1 → 2 → 5 → 3 → 6 → 7 → 4 → 1。注意，此最佳路徑的成本，剛好每一行及每一列都出現一個數字(表 6.8)。

表 6.8 在方向矩陣 *A* 中，最佳路徑的成本，剛好每一行及每一列都出現一個數字

	1	2	3	4	5	6	7
1	∞	7	∞	14	∞	∞	∞
2	21	∞	11	∞	2	∞	∞
3	∞	∞	∞	∞	∞	8	7
4	5	∞	∞	∞	6	∞	25
5	∞	8	3	9	∞	10	9
6	∞	∞	22	∞	36	∞	1
7	∞	∞	∞	4	16	34	∞

如果我們將任一行或任一列都同樣減去一個常數(使其剩餘值不為負值)，則此最佳旅行路徑並沒有任何改變。例如，將表 6.8 中的第一列都減去一個常數 7，此舉導致點 1 直接走到其他點的成本都同時下降 7。因為路徑的其他部分成本都不變，故每一條旅遊路徑的相對成本差還是一樣(表 6.9)，所以新的方向圖的最佳路徑還是原來的那一條(1 → 2 → 5 → 3 → 6 → 7 → 4 → 1)。注意這個被減去的常數，是所有路徑在這一步最少需要付出的成本。例如，自表 6.9 中得知，自點 1 直接走到其他點，最少需付出的成本為 7。

依據同樣的道理，只要任一行(或任一列)減去某一個常數，而不會出現負數，則上述的動作可以重覆執行，而最佳路徑依舊存在於最後的那一個方向圖中。將所有減去的常數進行加總，便成為所有路徑成本的一個下限。

表 6.9 第一列都減去一個常數 7 後的方向圖矩陣 *A*

	1	2	3	4	5	6	7
1	∞	0	∞	7	∞	∞	∞
2	21	∞	11	∞	2	∞	∞
3	∞	∞	∞	∞	∞	8	7
4	5	∞	∞	∞	6	∞	25
5	∞	8	3	9	∞	10	9
6	∞	∞	22	∞	36	∞	1
7	∞	∞	∞	4	16	34	∞

每一列都減去一個最大的常數，使得該列不會出現負數，的最後結果如表 6.10。目前累積的成本為 7+2+7+5+3+1+4=29。

表 6.10 第 1 列都減少 7、第 2 列都減少 2、第 3 列都減少 7、第 4 列都減少 5、第 5 列都減少 3、第 6 列都減少 1、第 7 列都減少 4 後的方向圖矩 *A*

	1	2	3	4	5	6	7
1	∞	0	∞	7	∞	∞	∞
2	19	∞	9	∞	0	∞	∞
3	∞	∞	∞	∞	∞	1	0
4	0	∞	∞	∞	1	∞	20
5	∞	5	0	6	∞	7	6
6	∞	∞	21	∞	35	∞	0
7	∞	∞	∞	0	12	30	∞

檢查表 6.10 矩陣 A 的每一行，發現第 6 行並沒有 0 的值。因此，第 6 行可以都減去 1，此旅行推銷員問題所有解的一個下限是 29+1=30。經調整後的新方向圖矩陣如表 6.11。

表 6.11 將表 6.10 之第 6 行都減去 1 後的方向圖矩陣 *A*

	1	2	3	4	5	6	7
1	∞	0	∞	7	∞	∞	∞
2	19	∞	9	∞	0	∞	∞
3	∞	∞	∞	∞	∞	0	0
4	0	∞	∞	∞	1	∞	20
5	∞	5	0	6	∞	6	6
6	∞	∞	21	∞	35	∞	0
7	∞	∞	∞	0	12	29	∞

以下我們將思考如何將旅行推銷員問題的解空間表示成一顆樹。

假設此旅行路徑選擇經過[2, 5]這條線，則此路徑的最小成本是 30。因為此類的旅行路徑皆經過[2, 5]，因為每一點僅能通過一次，故此類的旅行路徑不會在剩餘的旅程中，再也不會自點 2 出發到其他點，或自其他點抵達點 5。因此，我們可刪除第 2 列及第 5 行。而且[5, 2]這條線也不會再出現此路徑中(否則會造成迴圈)，因此可將 A[5, 2]設定為∞。經調整後的新方向圖矩陣如表 6.12 所示。

表 6.12 所有路徑經過[2, 5]的方向圖矩陣 *A*(刪除第 2 列及第 5 行之後並將 *A*[5, 2]設定為∞)

	1	2	3	4	6	7
1	∞	0	∞	7	∞	∞
3	∞	∞	∞	∞	0	0
4	0	∞	∞	∞	∞	20
5	∞	∞	0	6	6	6
6	∞	∞	21	∞	∞	0
7	∞	∞	∞	0	29	∞

相反地，當旅行路徑不經過[2, 5]，因此可將 *A*[2, 5]設定為∞(表 6.13)。

表 6.13 將 A[2, 5]設定為∞後的方向圖矩陣 *A*

	1	2	3	4	5	6	7
1	∞	0	∞	7	∞	∞	∞
2	19	∞	9	∞	∞	∞	∞
3	∞	∞	∞	∞	∞	0	0
4	0	∞	∞	∞	1	∞	20
5	∞	5	0	6	∞	6	6
6	∞	∞	21	∞	35	∞	0
7	∞	∞	∞	0	12	29	∞

雖然這些路徑不經過[2, 5]，但所有旅行路徑又必須經過 2，5 這兩點。所以此路徑一定自點 2 直接走到點 5 以外的點(即點 1 或點 3)，而且自點 2 以外的點(即點 4、點 6 或點 7)直接走進點 5。自點 2 直接走到點 1 或點 3 的最小成本為 9 (即 19 和 9 中的小值)，而自點 4、點 6 或點 7 直接走進點 5 的最小成本為 1 (即 1、35 和 12 中的最小值)。因此，不經過[2, 5]這條線的所有路徑的成本自 30 增加了 10(=9+1)，即為 40。

同樣地，第 2 列都減少 9 且第 5 行都減少 1，經調整後的新方向圖矩陣如表 6.14 所示。

表 6.14 所有路徑不經過[2, 5]的方向圖矩陣 *A*

	1	2	3	4	5	6	7
1	∞	0	∞	7	∞	∞	∞
2	10	∞	0	∞	∞	∞	∞
3	∞	∞	∞	∞	∞	0	0
4	0	∞	∞	∞	0	∞	20
5	∞	5	0	6	∞	6	6
6	∞	∞	21	∞	34	∞	0
7	∞	∞	∞	0	11	29	∞

以上利用[2, 5]將解空間分成兩個子樹，並且找到相對應的成本下限，如圖 6.6 所示。

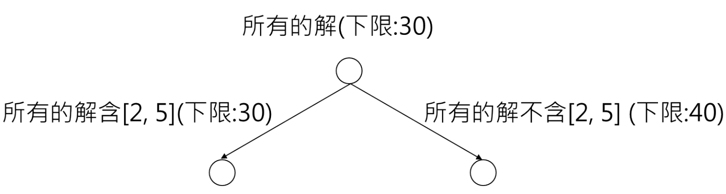


圖 6.6 利用[2, 5]來切割解空間

以上的作法可以持續進行，最後可以得到圖 6.7 之樹狀解空間。

自樹根(root)到最左下的樹葉(leaf)所形成的路徑包含[2, 5], [7, 4], [3, 7], [6, 3], [1, 2], [4, 1], [5, 6]，剛好可形成一個旅遊路徑 1 → 2 → 5 → 6 → 3 → 7 → 4 → 1 其成本為 57，可為目前「最佳解成本的上限」。

在搜尋整個樹狀解空間時，當某一子樹「預估的成本下限」大於此「最佳解成本的上限」時，此子樹就可以省略不必繼續搜尋。

例如，圖 6.7 中含[2, 5]但不含[7, 4]的所有解(被打×的節點)其下限為 65，已經超過 57(目前最佳解成本的上限)，可不必繼續搜尋。

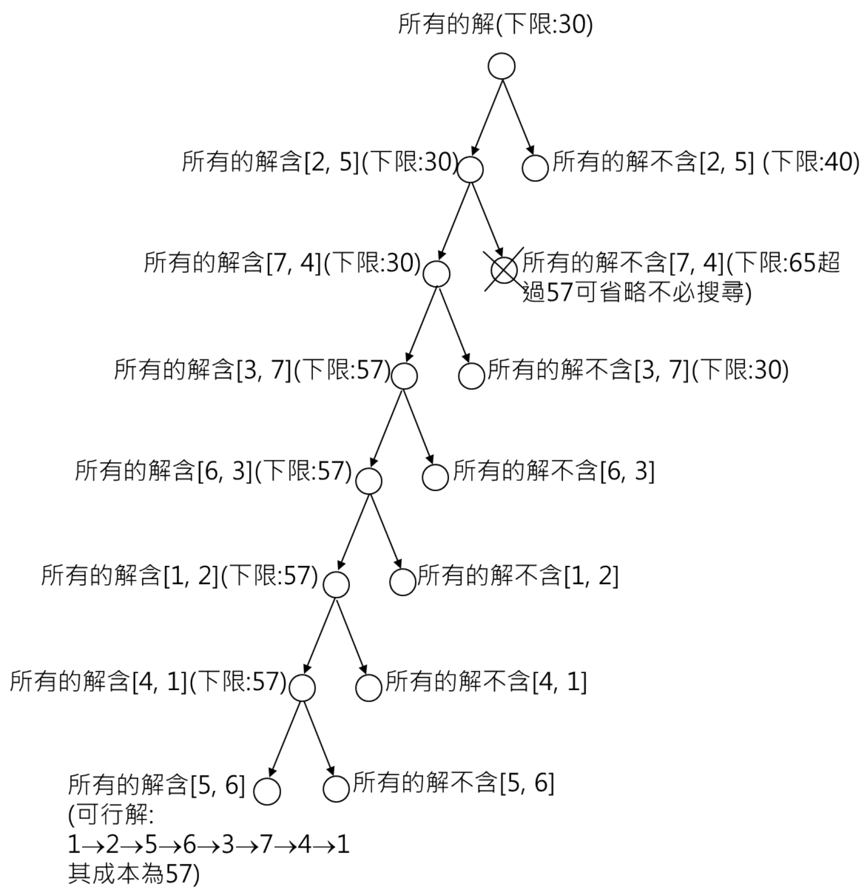


圖 6.7 找到一個旅遊路徑 1 → 2 → 5 → 6 → 3 → 7 → 4 → 1 其成本為 57。當搜尋到被打×的節點，因為其「預估的成本下限」大於目前「最佳解成本的上限」，可不必繼續搜尋此子樹

尋找旅行推銷員問題的最佳解的過程中，發現在一個子樹下一定找不到可行解時(如已經被選擇的線構成迴圈，但尚有其他點未被拜訪)，當然也可以立刻停止這棵子樹的搜尋。

6.6 樹搜尋法的技巧

「撤退法和寬度優先搜尋法，那一個方法會最快找到答案？」

「很難說耶！也許要看最佳解在樹中的位置吧！」

「有可能猜測出最佳解在樹中的位置？」

「也許需要評估每一個搜尋方向是否存有最佳解的可能！」

綜合以上的範例及對話，我們可以將使用樹搜尋法的技巧整理成表 6.15。

表 6.15 使用樹搜尋法解題的步驟

技巧	樹搜尋法
使用步驟	1. 問題的解空間可被表示成一棵樹。 2. 選擇合適的搜尋方式進行最佳解的搜索。若有必要，可以評估那一棵子樹較有可能存有最佳解，並優先進行搜索。 3. 搜索的過程中發現，某子樹內的解絕不存有最佳解時，則立即放棄此子樹的搜索。 4. 假設最佳解是可行解中最小成本時，當預估某一子樹內「可行解的成本下限」大於「最佳解成本的上限」時，可省略此子樹的搜索。

學習評量

1. 輸入一個圖及顏色數 k，請寫一個程式將此圖的點塗上顏色，使得相鄰兩個點的顏色不同。

```
輸入：

3       (顏色數 k)
4       (輸入圖點的個數，並且以 1, 2, 3, …編號)
5       (輸入圖線的個數)
1 2     (以下為線的資料)
1 4
2 3
2 4
3 4

輸出：

1 1    (點 1 被塗成顏色 1)
2 2    (點 2 被塗成顏色 2)
3 1    (點 3 被塗成顏色 1)
4 3    (點 4 被塗成顏色 3)
```

2. 輸入一個整數集合 S 及數字 m，請寫一個程式找出集合中的元素，使得其加總剛好為 m。

```
輸入：

3 1 5 8 13 6 7    (集合 S)
18                (數字 m)

輸出：

5 13
```

3. 漢米爾頓路徑：輸入一個圖，請寫一個程式找到一條路徑，將每一個點剛好經過一次，並且回到原點。

輸入：

```
5        (輸入圖點的個數，並且以 1, 2, 3, …編號)
7        (輸入圖線的個數)
1 2      (以下為線的資料)
1 4
2 3
2 4
3 4
3 5
4 5
```

輸出：

```
1 → 2 → 3 → 5 → 4 → 1
```