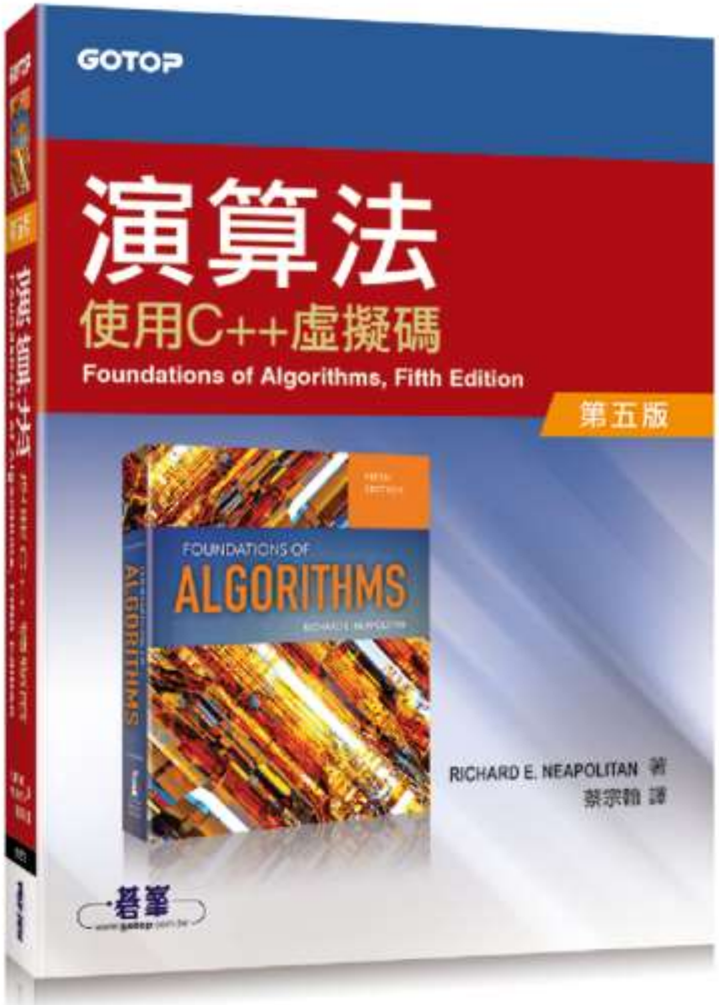


第五章 回溯

演算法

使用C++虛擬碼

Foundations of Algorithms, Fifth Edition

第五版

第五章 回溯(backtracking)

5.1 回溯(backtracking)技巧

- 回溯技巧通常被用來解決
  - 從一個物件集合中選出一序列的物件，並且這序列要滿足一些指定條件
    - *n*-Queen問題
      - 要將*n*個皇后放在一個的西洋棋盤上而彼此不互相攻擊。也就是說，沒有兩個皇后會位於同一列，同一排或同一對角線上
        - » 序列就是能安全地擺皇后的*n*個位置
        - » 物件集合就是西洋棋盤上所有可能位置，總共有n^2
        - » 問題的指定條件：任何兩個皇后都不會彼此互相攻擊

depth-first search (DFS) 演算法

- 回溯就是DFS的變形
- 樹狀結構的前序追蹤(**preorder** traversal)就是樹狀結構**depth-first search(DFS)**的結果
  - depth-first search會先走訪根節點(root node)，然後再依序走訪掛在根節點下面所有的子節點
- 簡單的遞迴演算法

```
void depth_first_tree_search (node v)
{
  node u;
  visit v;
  for (v的每個子節點 u)
      depth_first_tree_search (u);
}
```

depth-first search (DFS) 演算法

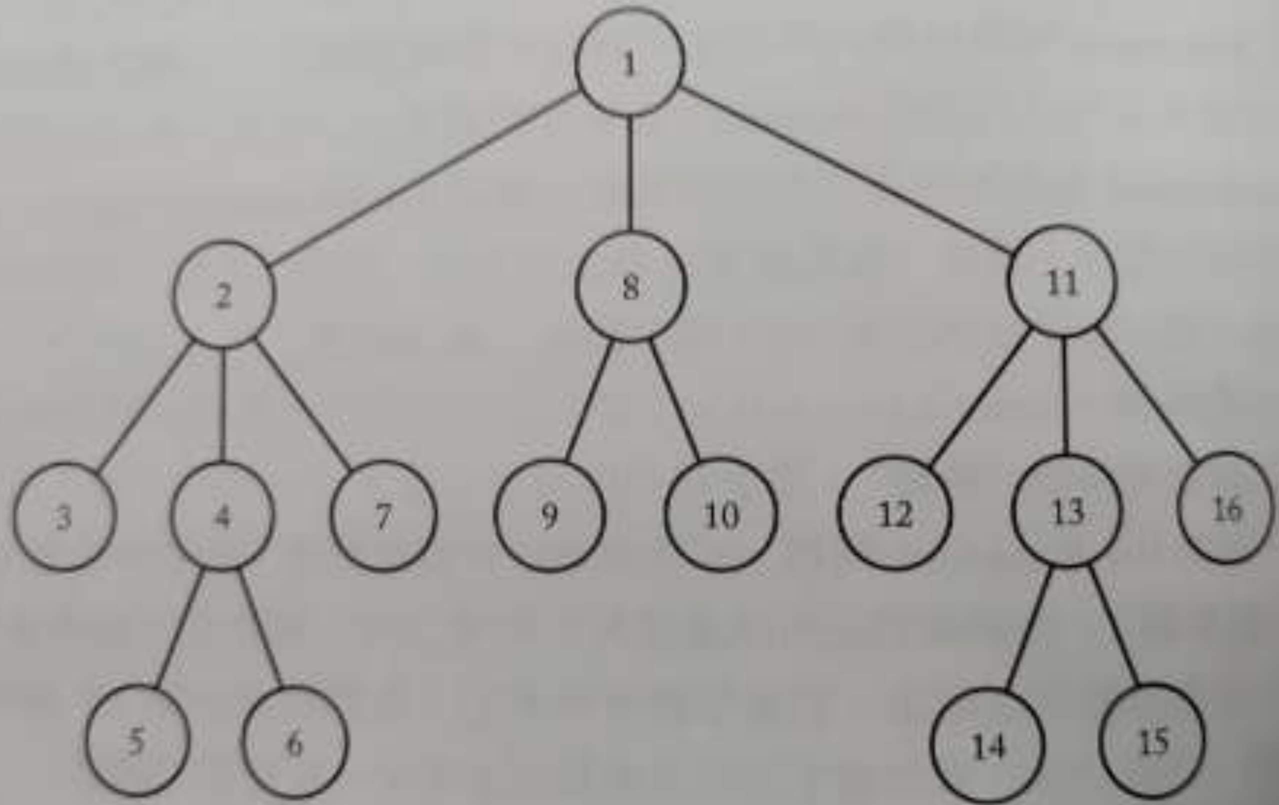


圖 5.1 一個樹狀圖。每個節點上的數字代表的是 depth-first search 的順序。

回溯演算法來解決4-Queen問題

- 每個皇后可以放在棋盤上這四個行的任何一個，所以總共只有$4 \times 4 \times 4 \times 4 = 256$種組合
- 產生樹狀結構來列舉所有可能的答案的組合

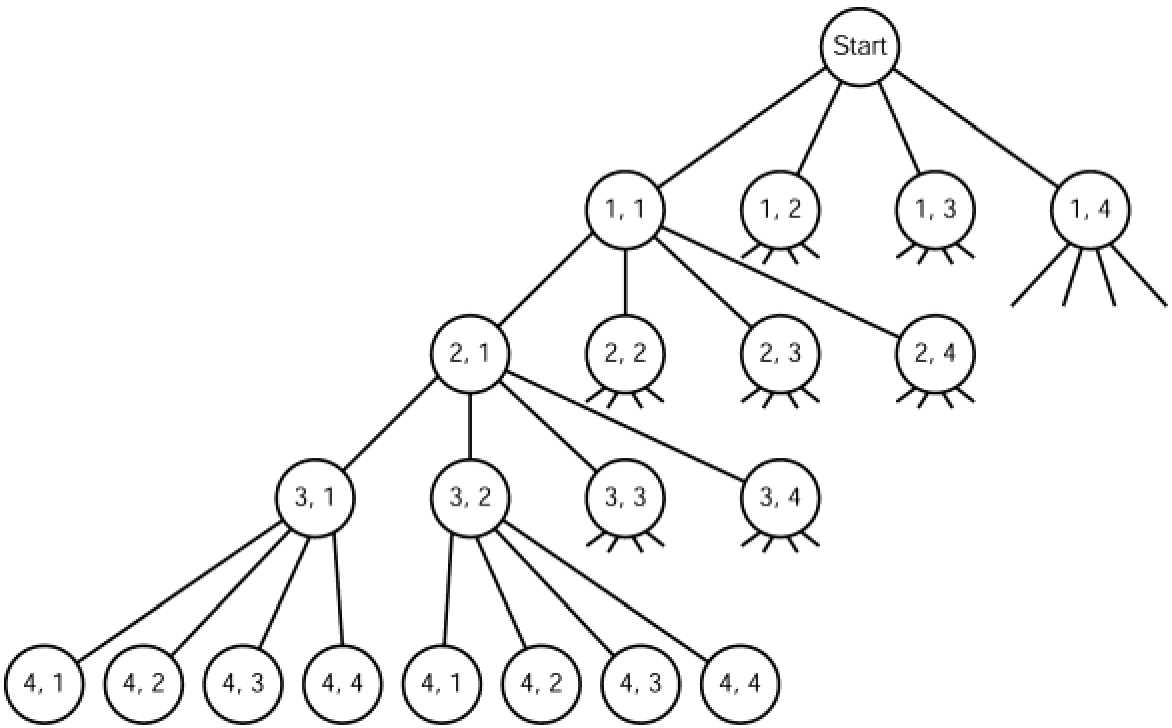


圖5.2 4-Queen問題一部分的狀態空間樹。

節點上的<*i*,*j*>代表的是*i*列的皇后是放在第*j*行上。

任何一條從樹根節點到葉節點的路徑就代表著一個可能的答案。

從左至右檢查每一個可能的答案組合，判斷它是不是正確答案

前幾個路徑是這樣的：

[<1,1>,<2,1>,<3,1>,<4,1>]
[<1,1>,<2,1>,<3,1>,<4,2>]
[<1,1>,<2,1>,<3,1>,<4,3>]
[<1,1>,<2,1>,<3,1>,<4,4>]
[<1,1>,<2,1>,<3,2>,<4,1>]

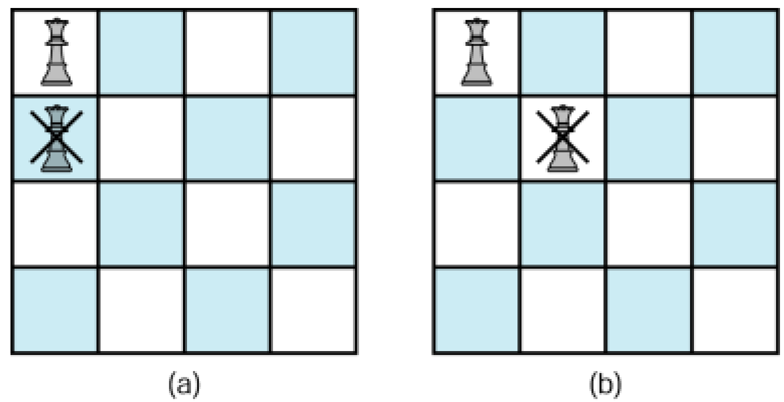


圖5.3

- 用DFS來走訪狀態空間樹，就好像試著探索迷宮裡面的每一條路，直到遇到死路才又繼續嘗試另一條一樣，完全不管沿路上的任何死路標誌
- 其實可以沿路注意是否有死路標誌來讓搜尋更有效率。舉例來說如圖5.3(a) 所示，根據規則任兩個皇后不可以放在同一列上，因此根本就不需要搜尋甚至產生在圖5.2中從節點以下的子樹。

- 回溯的意思就是指一旦我們知道此節點一定會通往死路時，我們就立即返回至父節點，繼續走訪那個父節點的其他子節點
- 當我們知道某一節點一定不會帶我們找到答案時，我們就稱該節點沒有希望的(nonpromising)，反之則稱之為有希望的(promising)
- 總而言之，回溯就是以DFS去走訪狀態空間樹，並檢查每個節點是不是promising。
  - 要是走到nonpromising的節點，我們就立刻返回父節點修剪(pruning)狀態空間樹，而所剩下的那些走訪過的節點就稱為修剪過的狀態空間樹(pruned state space tree)。

範例 5.1

- 對於*n*-皇后問題，當我們發現某節點和所有它的親代節點(ancestors)將皇后擺在相同的列或對角線上時，promising函式就必須傳回false。
- 在圖5.4中我們可以看到一個已經被修剪過的狀態空間樹(時)，而你所看到的這些節點都是在找到第一個答案之前所走訪過的節點。
- 圖5.5則是一個真的西洋棋盤。你可以對照圖5.4與5.5，在圖5.4中被畫叉(x)的節點代表這個節點是nonpromising，同樣的，你也可以看到對應的西洋棋盤中的格子也被畫叉。而在圖5.4中有顏色的那個節點就是包含解答的那個節點。

- 接下來，我們就來一步一步地來檢視這個過程。在這裡我們用有序數對來表示節點，而這個數對就是存在該節點的皇后列數。你可以注意到有些節點有相同的數對，只要你跟著圖5.4所畫的圖來走訪這棵樹，你就會知道我們目前所指的是哪個節點。

(a)<1,1> 是promising {因為這是第一個被擺入皇后}

(b)<2,1> 是nonpromising {因為皇后一擺在第一行}

<2,2> 是nonpromising {因為皇后一擺在左對角線上}

<2,3> 是promising

(c)<3,1> 是nonpromising {因為皇后一擺在第一行}

<3,2> 是nonpromising {因為皇后二擺在右對角線上}

<3,3> 是nonpromising {因為皇后二擺在第三行}

<3,4> 是nonpromising {因為皇后二擺在左對角線上}

(d)回溯至 <1,1>

<2,4> 是promising

(e)<3,1> 是nonpromising {因為皇后一擺在第一行}

<3,2> 是promising {注意，這是我們第二次遇到<3,2>}

(f) <4,1> 是nonpromising {因為皇后一擺在第一行}

<4,2> 是nonpromising {因為皇后三擺在第二行}

<4,3> 是nonpromising {因為皇后三擺在左對角線上}

<4,4> 是nonpromising {因為皇后二擺在第四行}

(g) 回溯至 <2,4>

<3,3> 是nonpromising {因為皇后二擺在右對角線上}

<3,4> 是nonpromising {因為皇后二擺在第四行}

(h)回溯至根節點

<1,2> 是promising

(i) <2,1> 是nonpromising {因為皇后一擺在右對角線上}

<2,2> 是nonpromising {因為皇后一擺在第二行}

<2,3> 是nonpromising {因為皇后一擺在左對角線上}

<2,4> 是promising

(j) <3,1> 是promising {注意，這是我們第三次遇到<3,1>}

(k)<4,1> 是nonpromising {因為皇后三擺在第一行}

<4,2> 是nonpromising {因為皇后一擺在第二行}

<4,3> 是promising

此時，我們找到了第一個解答。在圖5.5(k)顯示出這個狀況，而且你也可以在圖5.4看到那個被著色的節點就是該節點。

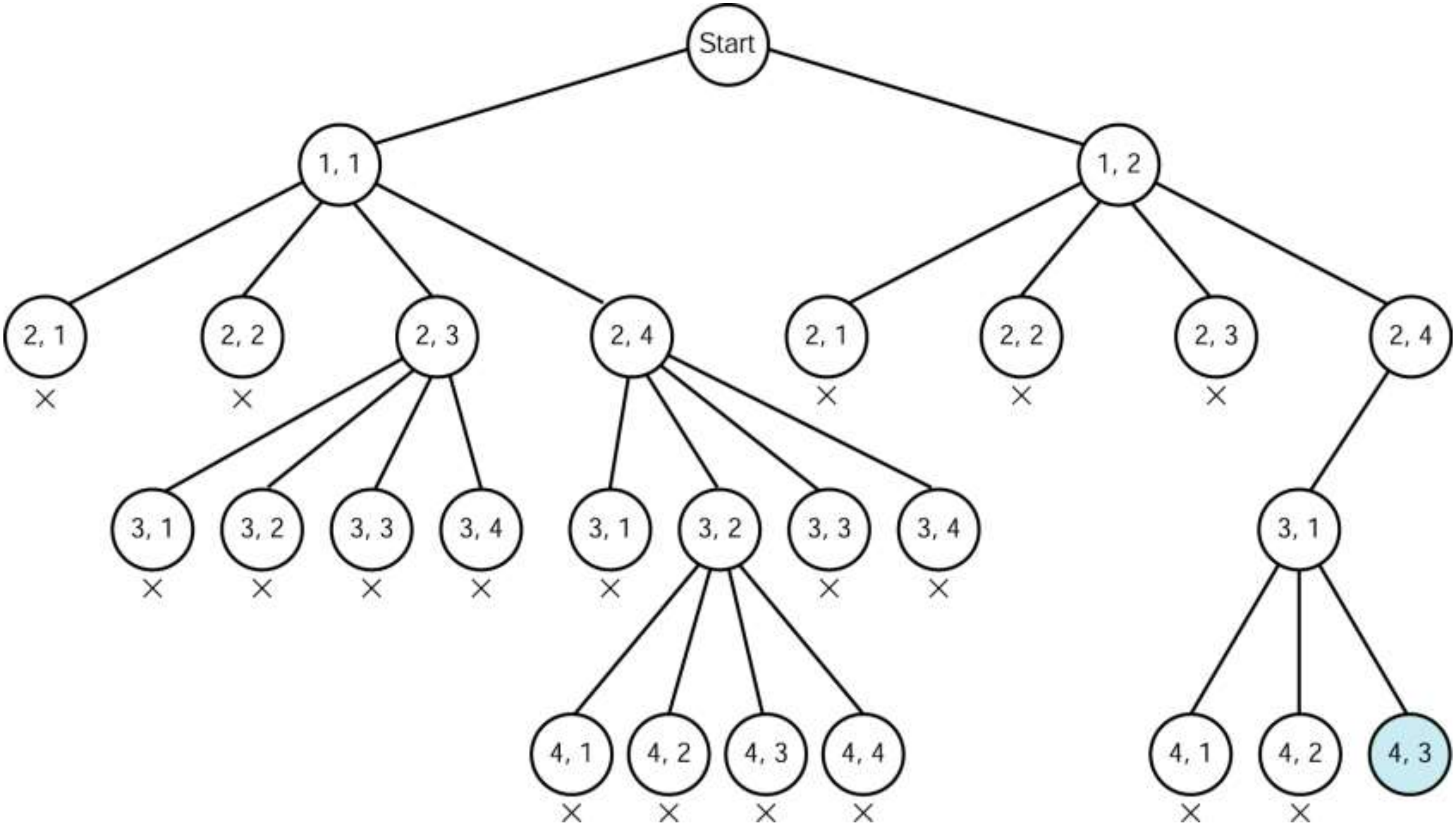


圖5.4 使用回溯演算法解決n=4時的n-皇后問題所產生的狀態空間樹的一小部分

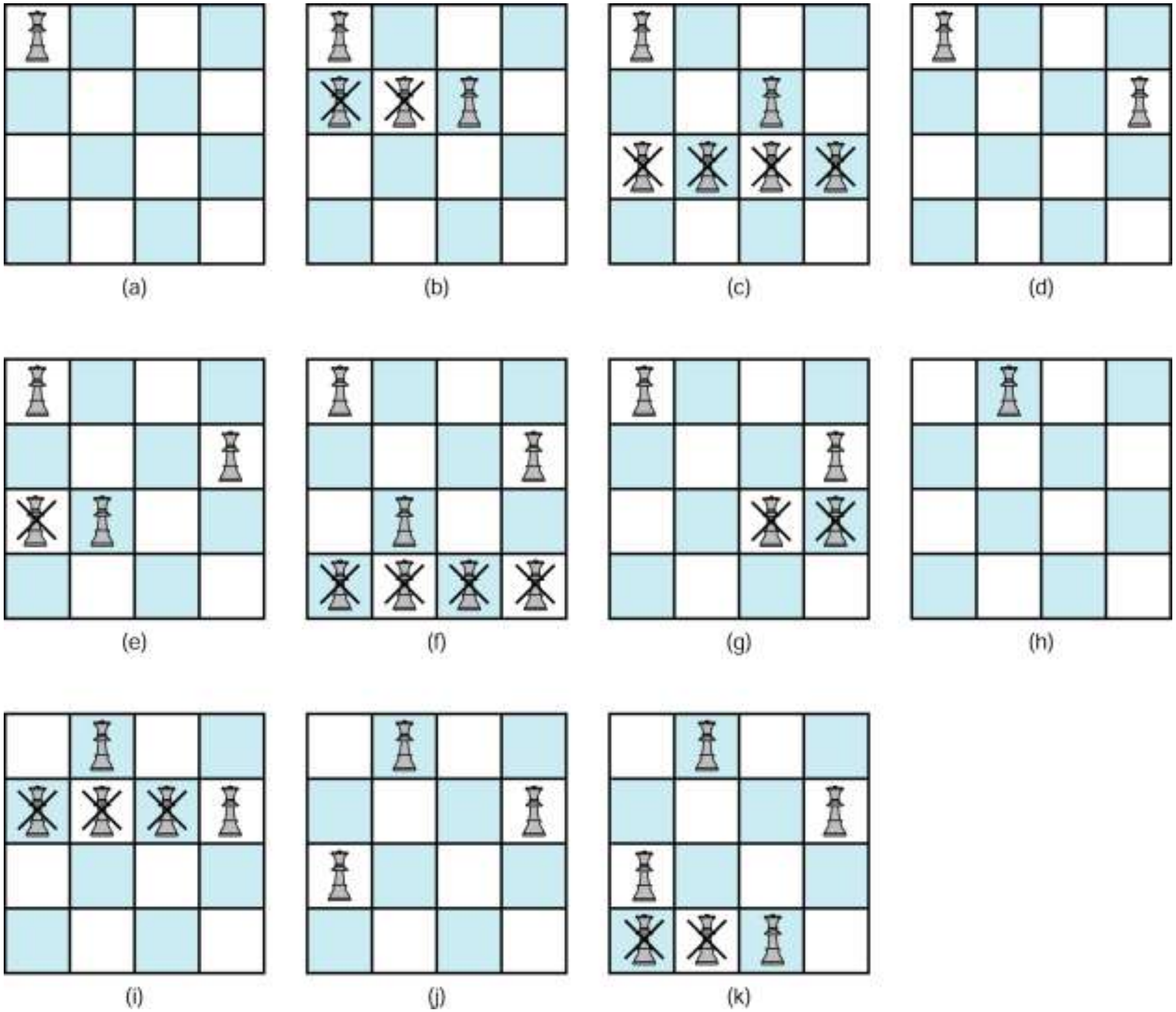


圖5.5 n=4時n-皇后問題所使用的西洋棋盤

5.2 *n*-皇后問題

- 檢查是否有兩個皇后放在同一行或同一對角線上。如果函式 可以傳回位於第i列的皇后所在的行數，則我們可以用下面的關係來檢查放在第k列的皇后是否站在同樣行上

$$col(i) = col(k)$$

- 接下來，檢查兩皇后是否在同一對角線上。圖5.6顯示出$n = 8$時的狀況。我們可看到位於第6列的皇后會被在它左對角線上，位於第3列的皇后所攻擊，也會被它右對角線上，位於第2列的皇后所攻擊

$$col(6) - col(3) = 4 - 1 = 3 = 6 - 3$$

- 也就是說，相對於左邊的那個皇后，它們位置上行數的差額等於列數的差額。此外，

$$col(6) - col(2) = 4 - 8 = -4 = 2 - 6$$

- 相對於右邊的那個皇后，它們位置上行數的差額也等於列數差額取負號。下面就是位於第k列的皇后會沿對角線攻擊位於第i列皇后的通則

$$col(i) - col(k) = i - k \text{ 或} col(i) - col(k) = k - i$$

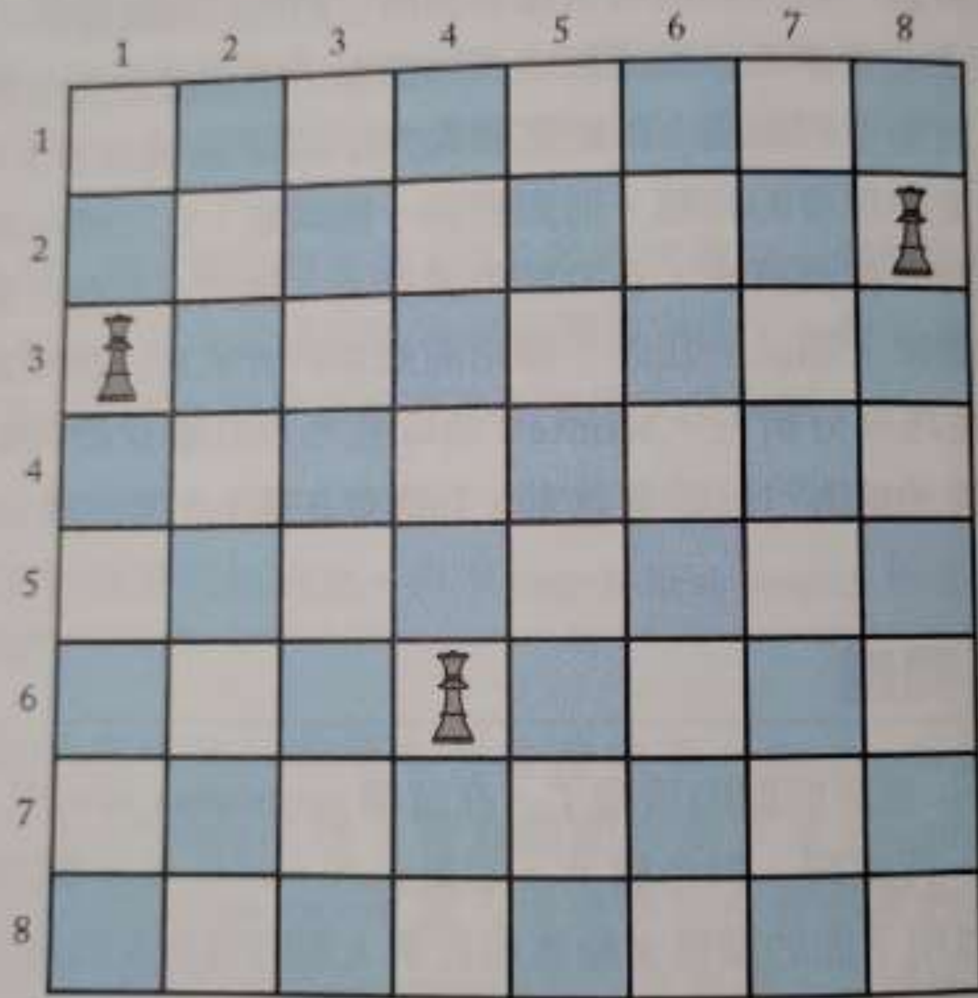


圖 5.6 位於第 6 列的皇后會被在它左對角線上，位於第 3 列的皇后所攻擊，也會被它右對角線上，位於第 2 列的皇后所攻擊。

目錄

演算法 5.1 用回溯解決*n*-皇后問題

- **問題**：將*n*個皇后放在西洋棋盤上使得任兩個皇后不在同一列、同一行或同一對角線上
- **輸入**：正整數*n*
- **輸出**：所有可以將*n*個皇后放在一個的西洋棋盤上而彼此不互相攻擊的方法。每個輸出都要包含一個標註從1到*n*的陣列 *col*，而*col*[*i*]就是位於第*i*列上皇后所在的行數

```
void queens (index i)
{
  index j;

  if (promising (i))
    if (i == n)
      cout << col[1] 至 col[n];
    else
      for (j = 1; j <= n; j ++){
        col[i + 1] = j;
        queens (i + 1);
        // 檢查位於第 ( i + 1 ) 列的皇后可否放在第 n 行上
      }
}
```

```
bool promising (index i)
{
  index k;
  bool switch;

  k = 1;
  switch = true;                    // 檢查有沒有其他皇后會攻擊第 i 列的皇后
  while (k < i && switch) {
     if (col[i] == col[k] || abs (col[i] - col[k]) == i - k)
        switch = false;
     k++;
  }
  return switch;
}
```

比較-檢查節點數

表 5.1 本表列出使用回溯演算法來解決 n- 皇后問題所能避免檢查的節點數*

n	演算法一所走訪的節點數 1†	演算法二所走訪的節點數 2‡	回溯走訪的節點數	回溯找到的 promising 節點數
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	9.73×10^{12}	4.79×10^{8}	1.01×10^{7}	8.56×10^{5}
14	1.20×10^{16}	8.72×10^{8}	3.78×10^{8}	2.74×10^{7}

* 每個數字代表的是找出所有的解所需走訪的節點數

† 演算法一是使用 depth-first search 但並沒有使用回溯

‡ 演算法二產生所有 $n!$ 種可能的解答，將皇后放在不同行列中

5.3 使用蒙地卡羅演算法估計回溯演算法的效率

- 可估計到底使用回溯演算法對於某個問題有效與否
- 或然式(probabilistic)演算法
  - 有時候接下來要執行的指令是由某個機率分佈來決定的(除非特別指明，我們假設此機率分佈是均等分佈 – uniform distribution)

- 產生一條“典型”的狀態空間樹路徑，不過這條路徑必須包含會被走訪的節點
- 然後用這條路徑來估算這個狀態空間樹上到底有多少節點會被走訪。這個方法可以用來估計在我們找到所有解前所需走訪的節點數，也就是說，這是一個對修剪過狀態空間樹內節點數的估計值。
- 要是用這個技巧來做估計，演算法必須滿足下面兩個條件：
    1. 對於狀態空間樹內同一層的節點，我們所使用的promising函式必須是相同的
    2. 狀態空間樹內，每個同一層的節點都必須有相同數量的子節點
- 演算法5.1(用回溯演算法來解決*n*-皇后問題)滿足這兩個條件

產生虛擬的隨機promising子節點

- 令m_0代表根節點所有promising子節點的個數
- 在樹的第一層隨機產生一個promising節點，此時讓m_1代表的是這個節點所有promising的子節點個數。
- 對於上一步所產生的那個節點，再隨機地產生它的promising子節點，然後讓m_2代表的是這個節點的所有promising子節點個數

……

- 對於上一步所產生的那個節點，我們再隨機地產生它的promising子節點，然後讓m_i代表的是這個節點的所有promising子節點個數
- 一直執行這個過程，直到再也找不到promising子節點為止

- 因為我們之前假設狀態空間樹內，每個同一層的節點都有相同數量的子節點，所以m_i就是我們對狀態空間樹中，第i層節點所具的promising子節點數的預估平均值。現在我們讓

$$t_i = \text{第}i\text{層節點所具有的子節點個數}$$

- 因為對任一節點，我們會走訪它所有個子節點，而且這個子節點中只有m_i個有promosing子節點，所以在回溯演算法找到所有解之前所需走訪的節點數，可以用下面的式子來預估

$$1 + t_0 + m_0 t_0 + m_0 m_1 t_2 + \cdots + m_0 m_1 \cdots m_{i-1} t_i + \cdots$$

- 下面列出這個一般的演算法。在這演算法中，我們用$mprod$變數來代表每一層的$m_0 m_1 \cdots m_{i-1}$值。

演算法 5.2 Monte Carlo估計演算法

- **問題**：使用Monte Carlo演算法來預估回溯演算法的效率。
- **輸入**：使用回溯演算法所要解的問題。
- **輸出**：修剪過狀態空間樹內節點數的預估值，也就是要找到所有解之前所需走訪節點數的預估值。

```
int estimate ()
{
  node v;
  int m, mprod, t, numnodes;

  v = 狀態空間樹的根節點;
  numnodes = 1;
  m = 1;
  mprod = 1;
  while (m != 0){
      t = v的子節點個數;
      mprod = mprod * m;
      numnodes = numnodes + mprod * t;
      m = v的promising子節點個數;
      if (m != 0)
          v = 從 v所有的promising子節點中隨機選一個節點;
  }
  return numnodes;
}
```

演算法 5.3 Monte Carlo估計演算法評估演算法5.1的效率

- **問題**：使用Monte Carlo演算法來預估演算法一的效率
- **輸入**：正整數n
- **輸出**：對於演算法一，其修剪過狀態空間樹內節點數的預估值，也就是找到能將n個皇后放在西洋棋盤且彼此不相互攻擊所有可能方法之前，所需走訪節點數的預估值

```
int estimate_n_queens (int n)
{
   index i, j, col[1..n];
   int m, mprod, numnodes;
   set_of_index prom_children;
   i = 0;
   numnodes = 1;
   m = 1;
   mprod = 1;
```

```
while (m != 0 && i !=n){
    mprod = mprod * m;
    numnodes = numnodes + mprod * n;  // 每個節點有 n 個子節點
    i ++;
    m = 0;
    prom_children = Ø;          // 將目前 promising 子節點集合初始化成空集合
    for (j = 1; j <= n; j ++){
        col[i] = j;
        if (promising (i)) {
            m++;
            prom_children = prom_children ∪ {j };
        // 判斷是不是 promising 節點，這個 promising 函式在演算法 5.1 中定義過了
        }
    }
    if (m != 0){
        j = random selection from prom_children;
        col[i] = j;
    }
}
return numnodes;
```

5.4 Sum-of-Subsets 問題

- 給定n個正整數(重量)和一個正整數W，找出所有重量和為W的子集合
- 範例5.2

假設$n = 5$，$W = 21$且

$w_1 = 5 \quad w_2 = 6 \quad w_3 = 10 \quad w_4 = 11 \quad w_5 = 16$

因為

$$w_1 + w_2 + w_3 = 5 + 6 + 10 = 21$$
$$w_1 + w_5 = 5 + 16 = 21$$
$$w_3 + w_4 = 10 + 11 = 21$$

所以解答就是$\{w_1, w_2, w_3\}$、$\{w_1, w_5\}$與$\{w_3, w_4\}$

建立狀態空間樹

- 圖5.7就畫出了一種建立狀態空間樹(state space tree)的方法。為了簡潔起見，圖中的樹只能處理三種不同重量的問題。
- 如果要拿w_1物件，我們就從根節點往左走，如果不拿w_1物件，我們就從根節點往右走
- 要拿w_2物件，我們就從第一層的節點往左走，如果不拿w_2物件，我們就從第一層的節點往右走，以此類推。
- 每一條路徑就代表一個子集合。在慣例上，如果我們有拿w_i，我們就在節線(edge)上標上w_i，如果我們沒拿，我們就在節線上標上0。

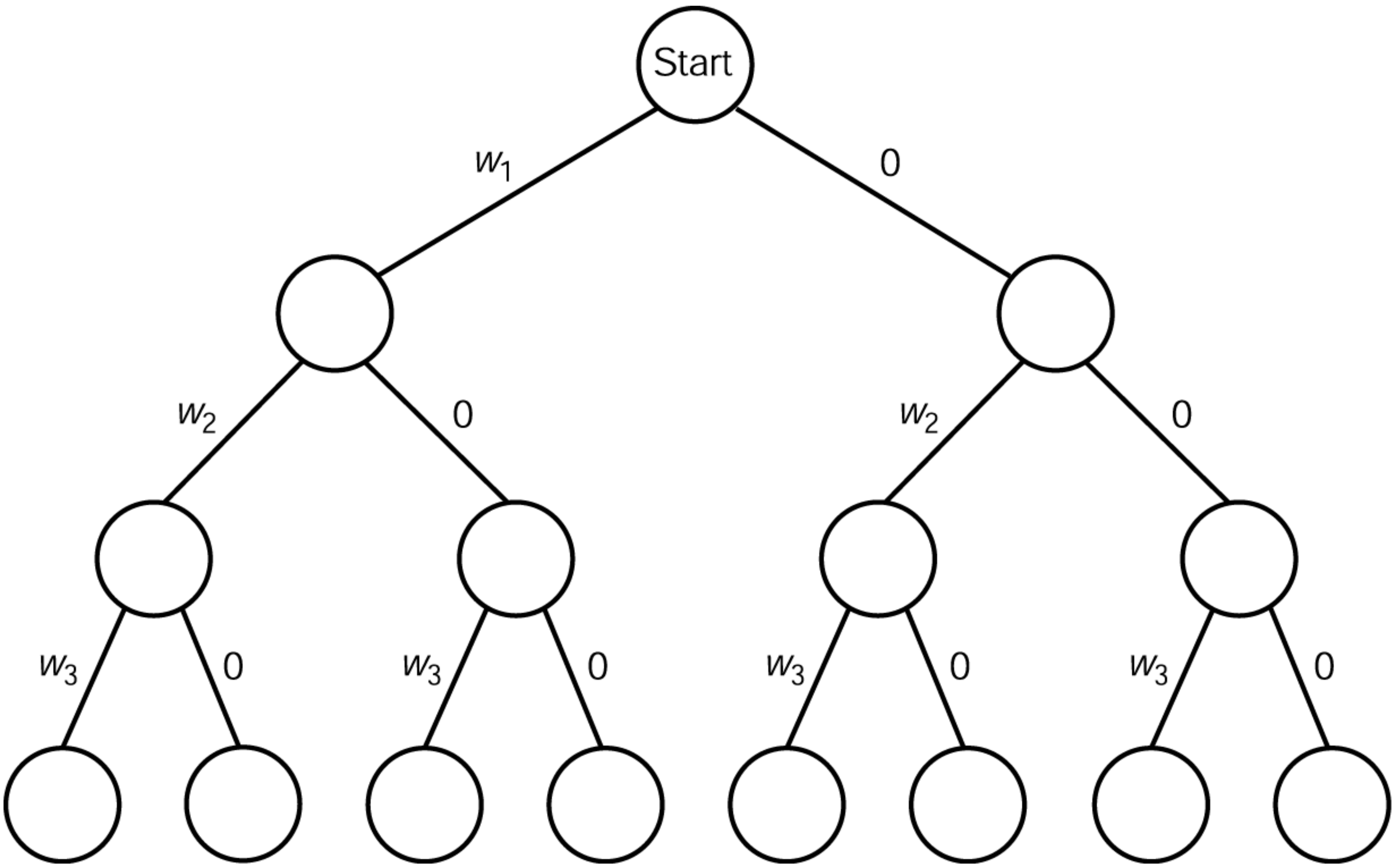


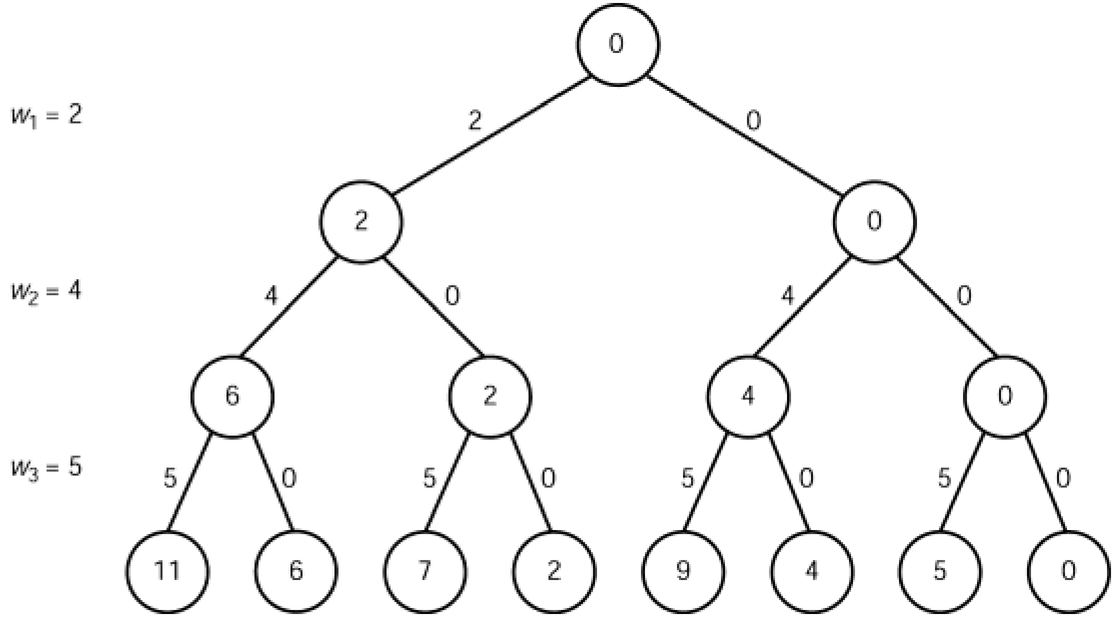
圖5.7˙ Sum-of-Subsets問題在n=3時的狀態空間樹

範例 5.3

圖5.8代表的是當$n = 3$，$W = 6$的狀態空間圖

$$w_1 = 2 \quad w_2 = 4 \quad w_3 = 5$$

在每個節點上寫了到目前為止所拿物件重量總合。因此，在葉節點上所寫的數字就是所拿的物件的重量總合。你可以注意到，只有從左邊數過來的第二個葉節點上的數字是6。因為到此葉節點的路徑代表的是這個子集合，所以我們知道這個子集合是唯一的解。



- 若事先能將所有重量先以遞增的方式排序，很明顯地，就能知道哪個節點是promising，哪個是nonpromising。
- 如果所有的重量是以這樣的方法排序的話，那麼在第i層的節點上，w_{i+1}就是所剩下的物件最輕的物品
- ***weight***代表到第i層節點時的物件重量總合
- 若w_{i+1}加上了***weight***會超過W的話，那麼加上其他的物件也必定會超過W。因此，除非***weight***等於W(也就是說在這個節點上有一解)，在第i層的節點如果是

$$\boldsymbol{weight} + w_{i+1} > W$$

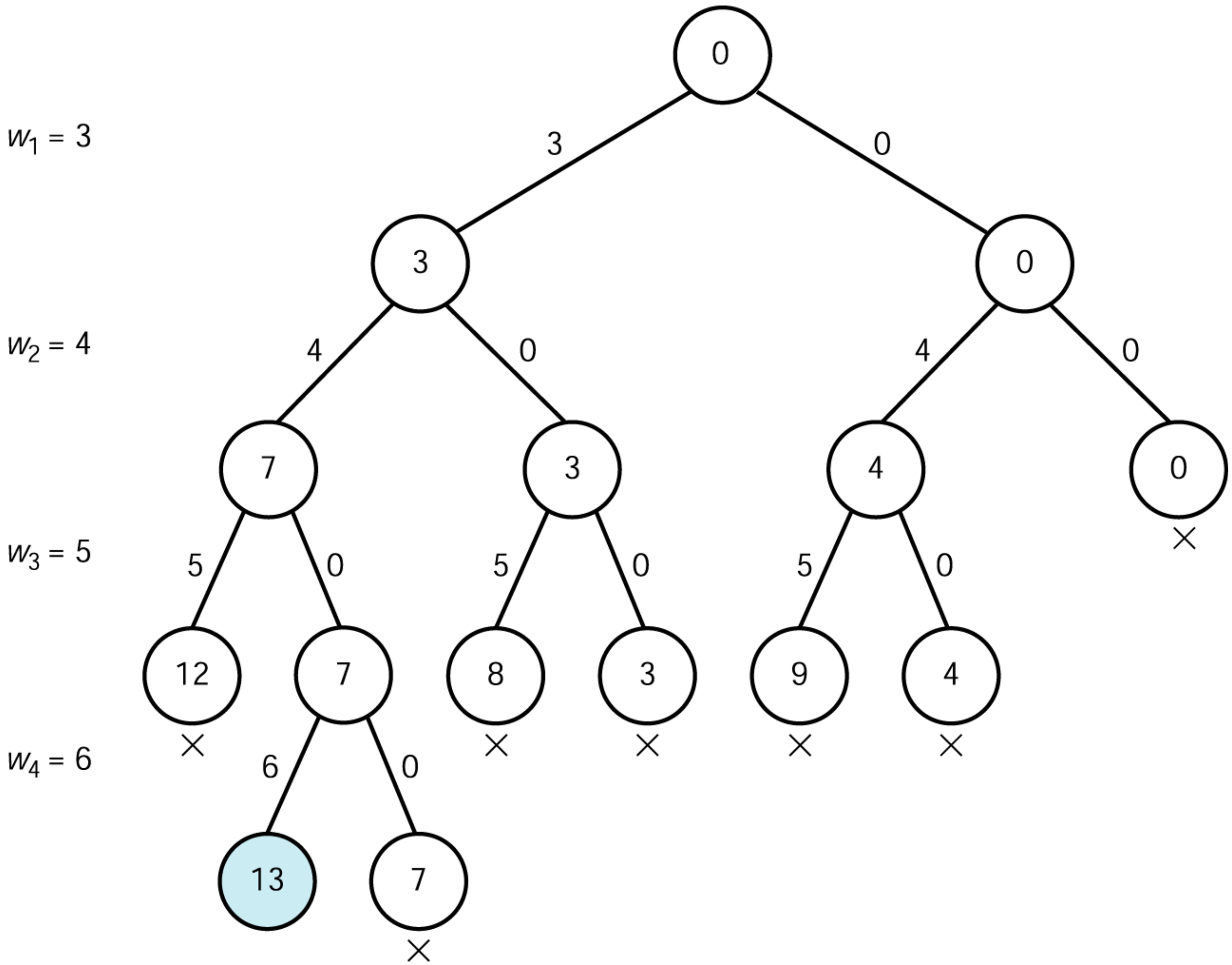
- 就表示它是nonpromising。

範例 5.4

- 圖5.9展示使用回溯法來處理$n = 4$、$W = 13$以及
- $w_1 = 3 \quad w_2 = 4 \quad w_3 = 5 \quad w_4 = 6$
- 之案例過後的已修剪狀態空間樹，這個問題只有一個解，我們將那個節點著上色。這個解是$\{w_1, w_2, w_4\}$。

- 我們也用叉號來標記nonpromising節點。數字是12、8及9的節點都是nonpromising因為當我們再補上下一個物件重量(6)時，weight就會超過W。
- 數字是7、3、4及0的節點也都是nonpromising因為沒有足夠的重量會讓它加起來是W。
- 這裡請注意一點，所有不含解答的葉節點都是nonpromising 因為再也沒有任何物品可以被加入至集合中讓weight等於W。數字是7的葉節點就是這種狀況。此外我們可以發現，在這個修剪過的狀態空間樹中總共有15個節點，而整個狀態空間數共有31個節點。

習題5.4中使用回溯所產生的修剪狀態空間樹



演算法5.4 用回溯解決Sum-of-Subsets問題

- **問題**：給定n個正整數(重量)和另一個正整數W，找出所有重量總和是W的正整數集合。
- **輸入**：正整數n，已經排序過的遞增正整數w，正整數W。
- **輸出**：所有重量總和為W的正整數集合。

```
void sum_of_subsets (index i,
                     int weight, int total)
{
  if (promising (i))
     if (weight ==W)
        cout << include [1] through include [i];
     else {
        include [i + 1] = "yes";          // 包含 w[i + 1].
        sum_of_subsets (i + 1, weight + w[i +1], total - w[i + 1]);
        include [i + 1] = "no";           // 不包含 w[i + 1].
        sum_of_subsets (i + 1, weight, total - w[i + 1]);
     }
}
bool promising (index i);
{
return (weight + total >=W) && (weight ==W || weight + w[i + 1] <= W);
}
```

5.5 圖形著色

- *m*-著色問題的目標就是要找出所有可能的方法，用至多*m*種顏色，來對一個沒有方向性的圖形著色，並使得任兩個相鄰的頂點不會被塗上相同的顏色

範例 5.5

現在討論圖5.10。這個圖沒有2-著色問題的解，因為若只用至多兩種顏色來著色，我們無法讓相鄰兩個頂點有不同的顏色。而3-著色問題的解如下：

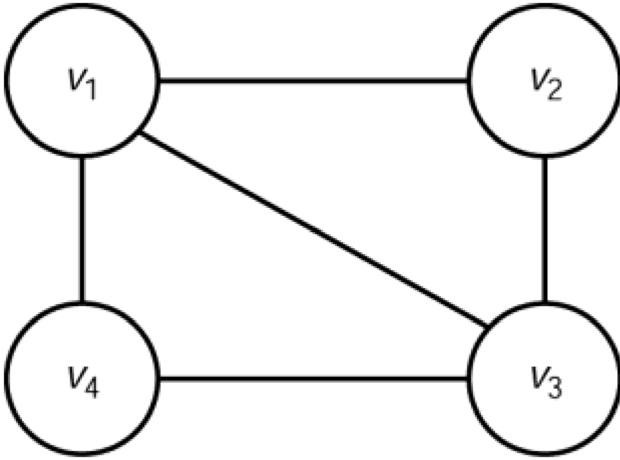


圖 5.10

v_1:顏色1

v_2:顏色2

v_3:顏色3

v_4:顏色2

地圖的著色

- 如果一個圖形可以在平面上用這樣的方式著色並使任何節線不相交，我們就稱這圖形是平面的 (planar)圖形。
  - 例：圖5.11下面的那個圖形就是planar
- 每個地圖都會有一個對應的planar圖形。每個區域就是一個圖形的頂點，如果兩個地區是相鄰的，則對應的頂點間就有節線所連接。圖5.11上面就是一個地圖而下面就是它的圖形表示法。

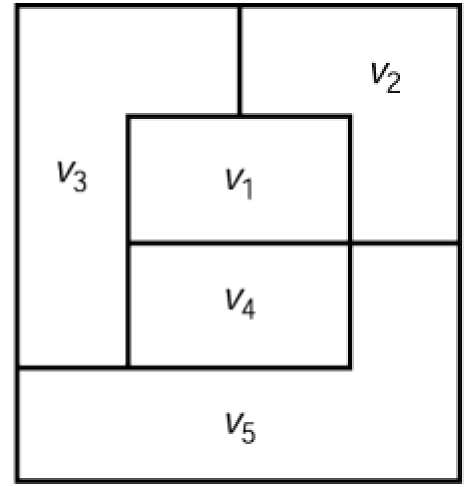
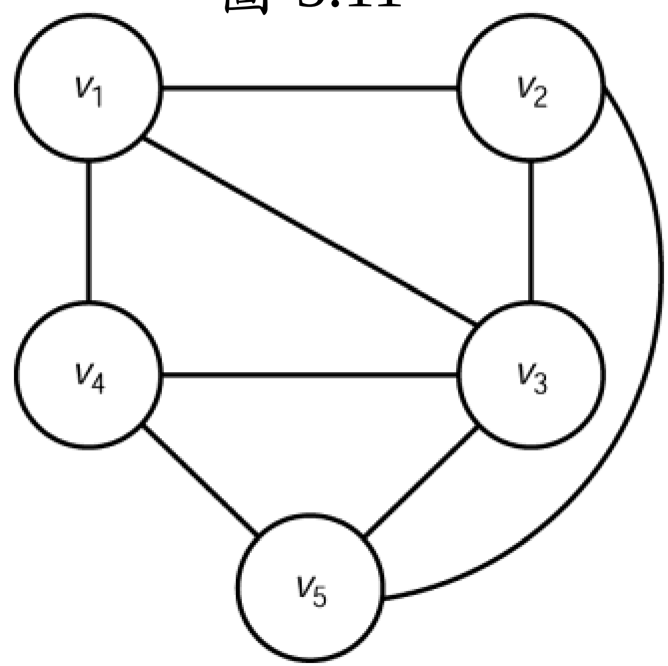
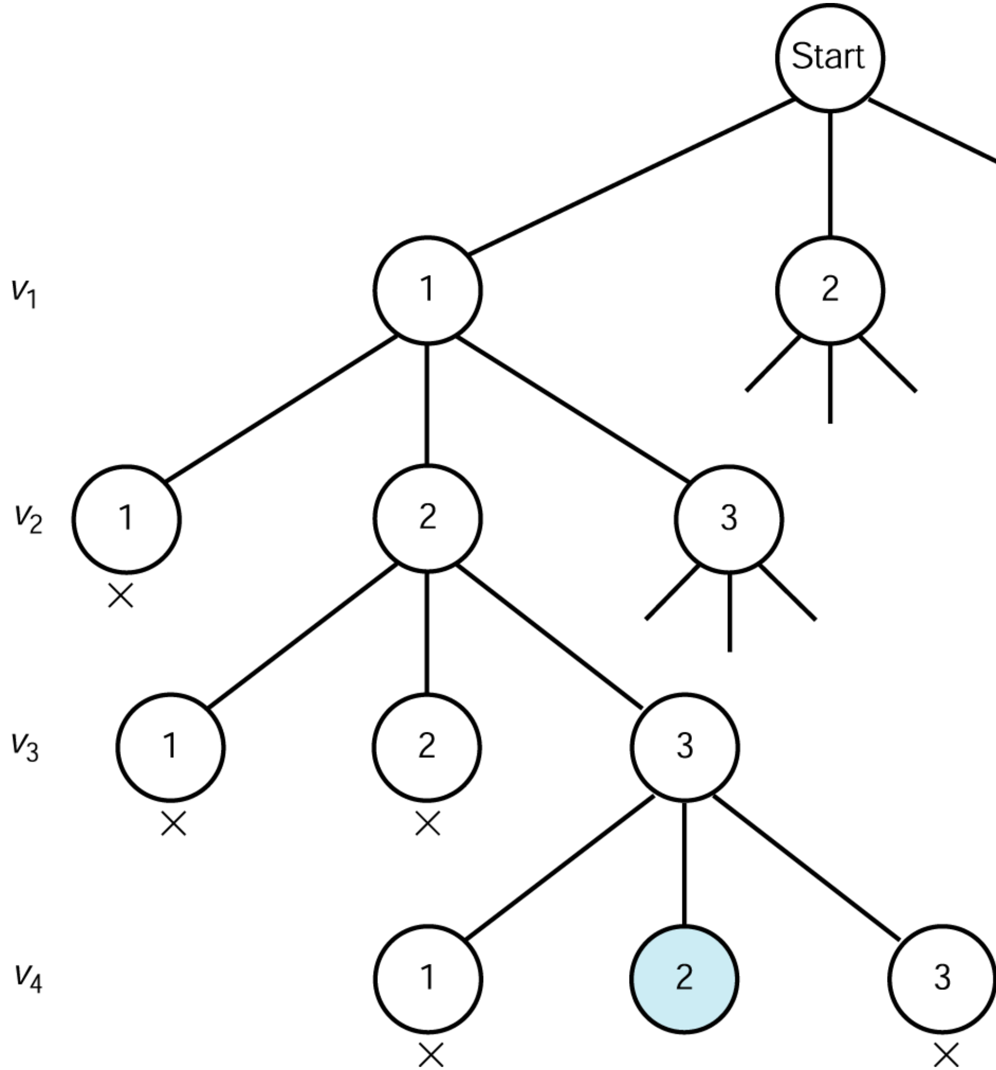
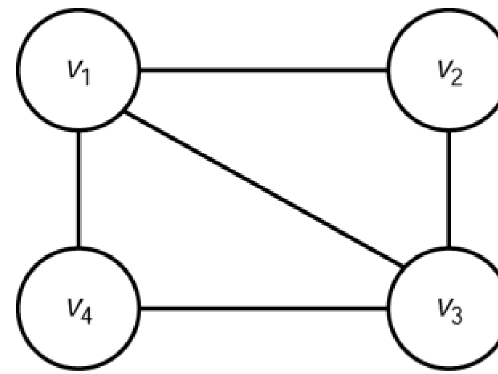


圖 5.11



- 著色問題可以使用回溯
  - 要是目前的節點所使用的顏色與它相鄰節點有相同的顏色時，這個節點就是nonpromising
- 圖5.12所顯示的就是一個修剪過的狀態空間圖的一小部分。它是使用回溯技巧，解決圖5.10中的圖形之3-著色問題。
- 節點內的數字代表著在對應的頂點所使用的顏色。有顏色的那個節點就是第一個找到的解答，而我們也用叉號(X)來標記nonpromising節點。
- 當我們將著上顏色1後，因為v_1與v_2是相連的，所以如果我們也將v_2著上顏色1，這個節點就會是nonpromising
- 同樣地，如果我們將v_1、 v_2與v_3分別著上顏色1、顏色2與顏色3後，接下來若將v_4著上顏色1會是這個節點變成nonpromising，原因正是因為v_1與v_4是相連的





	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

圖5.12 這是使用回溯來解決圖5.10中圖形的3-著色問題，所產生的部分修剪過後的狀態空間圖

演算法 5.5 解*m*-著色問題(回溯演算法)

- **問題**：找出所有可能的方式，只用*m*種顏色，來對一個沒有方向性的圖形著色，並使得任兩個相鄰的頂點不會被塗上相同的顏色。
- **輸入**：正整數*n*和*m*，一個有*n*個頂點且沒有方向性的圖形。這個圖形用一個二維陣列*W*來表示，其中它的行和列都是由1到*n*來表示。如果*W*[*i*][*j*]是true，就代表著第*i*個頂點與第*j*個頂點間有節線連接，要是*W*[*i*][*j*]是false就表示第*i*個頂點與第*j*個頂點間是不相連的。
- **輸出**：所有可能的方法，用至多*m*種顏色，來對一個沒有方向性的圖形著色，並使得任兩個相鄰的頂點不會被塗上相同的顏色。著色結果是儲存在索引為1到*n*的*vcolor*陣列， *vcolor*[*i*]代表的就是第*i*個頂點的顏色(正整數1到*m*)。

```
void m_coloring (index i)
{
  int color;
  if (promising (i))
     if (i == n)
        cout << vcolor [1] through vcolor [n];
     else
        for (color = 1; color <= m; color ++){ // 對下個頂點嘗試著每種顏色
           vcolor [i + 1] = color;
           m_coloring (i + 1);
        }
}
bool promising (index i)
{
  index j;
  bool switch;

  switch = true;
  j = 1;
  while (j < i && switch) {                  // 檢查是否有相連的頂點有相同顏色
     if (W[i][j] && vcolor [i] == vcolor [j])
        switch = false;
        j++;
  }
  return switch;
}
```

5.6 漢米爾頓迴路問題

- 對於任一個沒方向性的連接圖形，所謂的漢米爾頓迴路(Hamiltonian Circuit，又稱為旅程-tour)指的就是一條從某個起始點開始，經過圖形中的任何其他頂點僅一次，然後再回到原本那個起始頂點的路徑

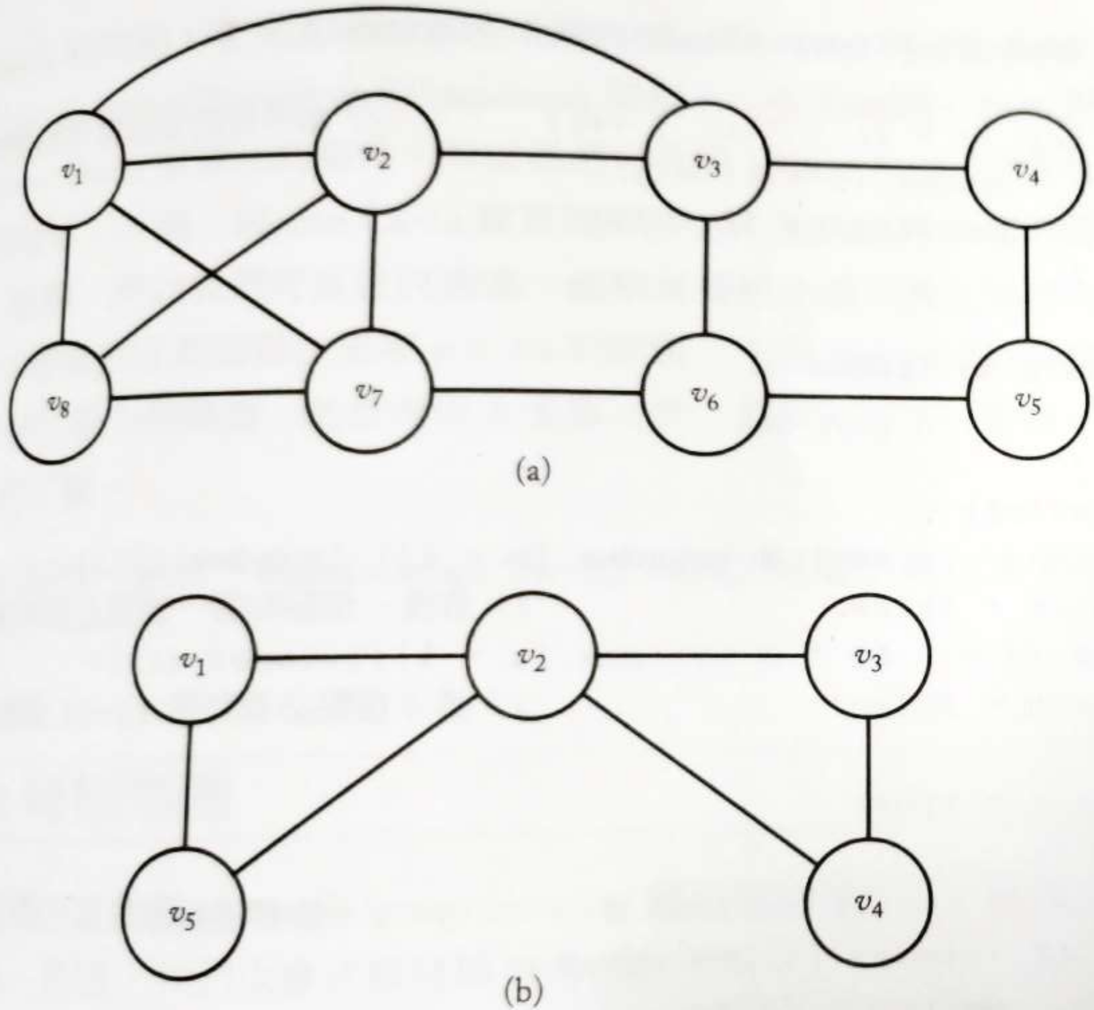
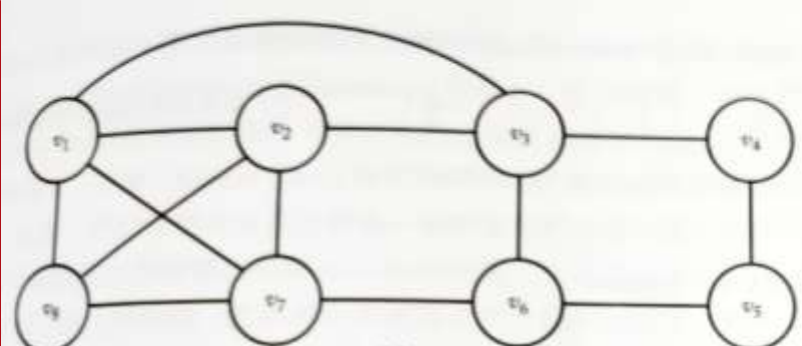


圖5.13 圖 (a) 內有一條 Hamiltonian circuit $[v_1, v_2, v_8, v_7, v_6, v_5, v_4, v_3]$ 而在圖 (b) 內找不到任何一條 Hamiltonian circuit。

- 這個問題的狀態空間樹可以這樣建立：
  - 樹的第零層就是起始頂點，我們稱之為路徑的第零個點。在樹的第一層，我們會考慮除了起始頂點外的所有頂點當作路徑的第一個點。在樹的第二層，考慮除了路徑上前一個點外的所有頂點當作路徑的第二個點，依次類推，一直到樹的第$n-1$層

- 下面這些考量可以讓我們決定什麼時候要在狀態空間樹裡做Backtrack的動作：
  1. 路徑上第*i*個點在圖形上必須與路徑上第個點是相連的
  2. 路徑上第個點在圖形上必須與路徑上第0個點是相連的
  3. 路徑上第*i*個點不可以與路徑上的前個點重複

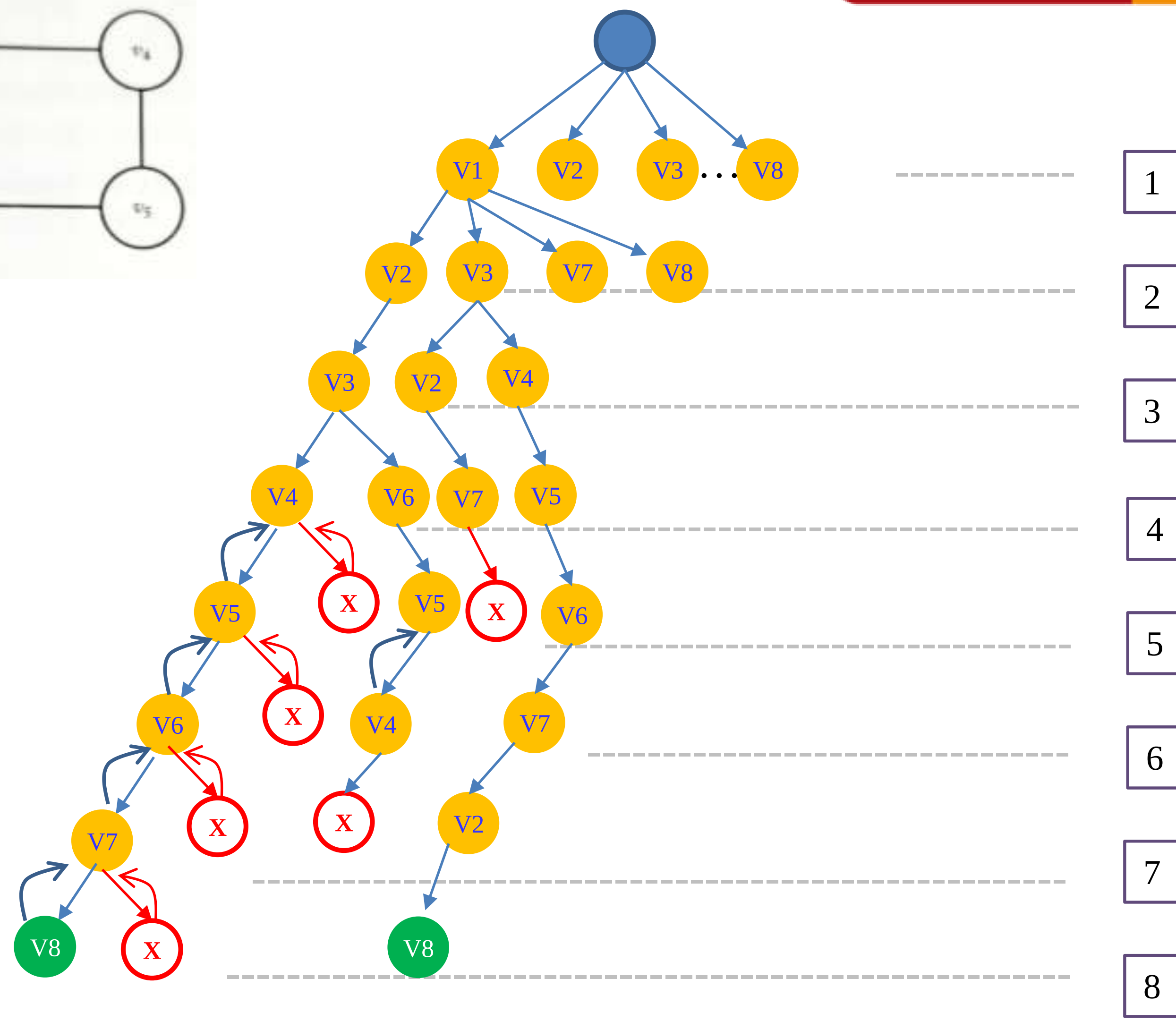


(a)

Adjacency Matrix

	v1	v2	v3	v4	v5	v6	v7	v8
v1	0	1	1	0	0	0	1	1
v2	1	0	1	0	0	0	1	1
v3	1	1	0	1	0	1	0	0
v4	0	0	1	0	1	0	0	0
v5	0	0	0	1	0	1	0	0
v6	0	0	1	0	1	0	1	0
v7	1	1	0	0	0	1	0	1
v8	1	1	0	0	0	0	1	0

1	2	3	4	5	6	7	8



目錄

演算法5.6 漢米爾頓迴路問題(用回溯演算法解)

- **問題**：在一個沒方向性的連接圖形中找出所有的漢米爾頓迴路。
- **輸入**：正整數 n 和一個有 n 個頂點的無方向性圖形。這個圖形用一個二維陣列 W 來表示，其中它的行和列都是由1 到 n 來表示。如果 $W[i][j]$ 是true，就代表著第 i 個頂點與第 j 個頂點間有節線連接，要是 $W[i][j]$ 是false就表示第 i 個頂點與第 j 個頂點間是不相連的。
- **輸出**：找出所有從某個起始點開始，經過圖形中的任何其他頂點僅一次，然後再回到原本那個起始頂點的路徑。輸出結果是儲存在索引為1到 n - 1 的 *vindex* 陣列，*vindex*[i] 代表的就是路徑上的第 i 個點。路徑的起始點為 *vindex*[0] 。

```
void hamiltonian (index i)
{
  index j;
  if (promising (i)
     if (i == n - 1)
        cout << vindex [0] through vindex [n - 1];
     else
        for (j = 2; j <=n; j ++){          // 拿所有的頂點當作路徑的下個點
           vindex [i + 1] = j;
           hamiltonian (i + 1);
        }
}
bool promising (index i)
{
  index j;
  bool switch;
  if (i == n - 1 && ! W [vindex [n - 1]] [vindex [0]])
     switch = false;                        // 最後一個點和第一個點必須相連
  else if (i > 0 && ! W [vindex [i - 1]]
     switch = false; [vindex [i]])          // 第 i 個點必須和第 (i-1) 個點相連
  else {
     switch = true;
     j = 1;
     while (j < i && switch) {             // 檢查這個點是否已經出現過
         if (vindex [i] == vindex [j])
            switch = false;
         j++;
     }
  }
  return switch;
}
```

5.7 0-1 背包問題

5.7.1 用回溯解決0-1背包問題

範例5.6

假設$n = 4$，$W = 16$而且我們有下面的狀況：

i	p_i	w_i	$\frac{p_i}{w_i}$
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5
4	\$10	5	\$2

- 圖5.14就是依據回溯策略所產生的修剪過的狀態空間樹。節點內從上而下代表的是該節點的全部獲益(profit)，全部重量(weight)及獲益上限值(bound)。
- 有顏色的那個節點就是有最大效益的那組解。每個節點也都標記了它在樹的深度以及從樹左邊數過來的位置。
  - 舉例來說，有顏色的那個節點被標了(3,3)，因為它是在樹的第三層而且是從左數過來的第三個節點

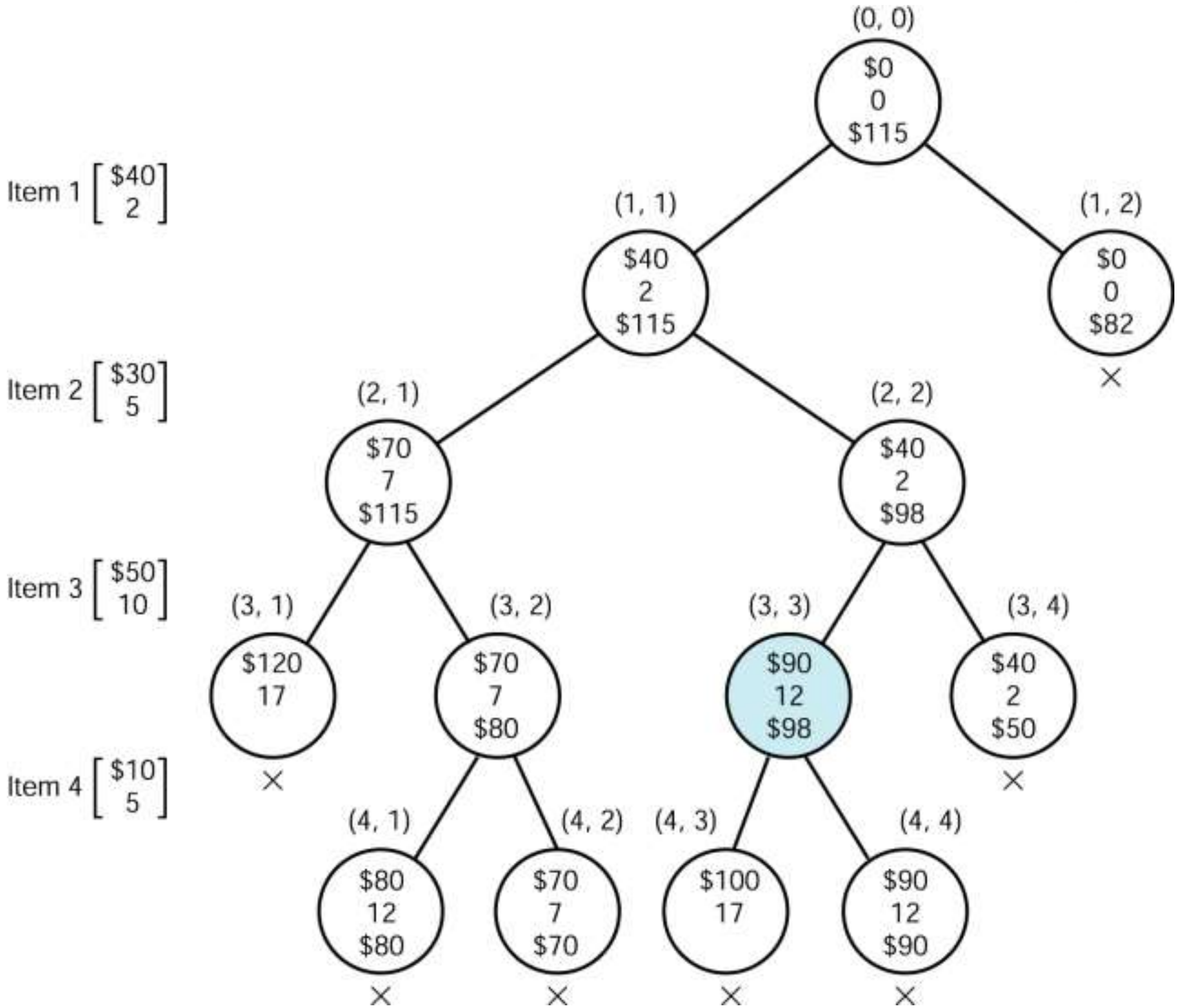


圖5.14 使用回溯演算法來解習題5.6所建立的修剪過的狀態空間樹

1. 將*maxprofit*設成 $0。
2. 走訪(0,0)節點，也就是根節點。
   a) 計算它的profit和weight。

$$profit = \$0$$
$$weight = 0$$

   b) 計算它的bound值。因為$2 + 5 + 10 = 17$且$17 > 16$ (W的值)，所以加入第三個物品就會讓總重量超過W。因此，$k = 3$，而且

$$totweight = weight + \sum_{j=0+1}^{3-1} w_j = 0 + 2 + 5 = 7$$

$$bound = profit + \sum_{j=0+1}^{3-1} p_j + (W - totweight) \times \frac{p_3}{w_3}$$

$$= \$0 + \$40 + \$30 + (16 - 7) \times \frac{\$50}{10} = \$115$$

   c) 判斷出本節點是promising因為它的小於16(W的值)並且大於$0(目前*maxprofit*的值)

3. 走訪(1,1)節點。

- 計算它的profit和weight

$$profit = \$0 + \$40 = \$40$$
$$weight = 0 + 2 = 2$$

- 因為$weight = 2$小於等於16(W的值)而且它的$profit = \$40$大於\$40(目前*maxprofit*的值)，所以將*maxprofit*設成\$40。

- 計算它的bound值。因為$2 + 5 + 10 = 17$且$17 > 16$ (W的值)，所以加入第三個物品就會讓總重量超過W。因此，$k = 3$，而且

$$totweight = weight + \sum_{j=1+1}^{3-1} w_j = 2 + 5 = 7$$

$$bound = profit + \sum_{j=1+1}^{3-1} p_j = (W - totweight) \times \frac{p_3}{w_3}$$

$$= \$40 + \$30 + (16 - 7) \times \frac{\$50}{10} = \$115$$

- 判斷出本節點是promising因為它的$weight = 2$小於16(W的值)並且大於\$0(目前*maxprofit*的值)

4. 走訪(2,1)節點

- 計算它的profit和weight

$$profit = \$40 + \$30 = \$70$$

$$weight = 2 + 5 = 7$$

- 因為$weight = 7$小於等於16(W的值)而且它的$profit = \$70$大於\$40(目前*maxprofit*的值)，所以將*maxprofit*設成\$70
- 計算它的bound值

$$totweight = weight + \sum_{j=2+1}^{3-1} w_j = 7$$

$$bound = \$70 + (16 - 7) \times \frac{\$50}{10} = \$115$$

- 判斷出本節點是promising因為它的$weight = 7$小於16(W的值)並且$bound = \$115$大於\$70(目前*maxprofit*的值)

5.走訪(3,1)節點

- 計算它的profit和weight

$$profit = \$70 + \$50 = \$120$$

$$weight = 7 + 10 = 17$$

- 因為$weight = 17$大於16(W的值)，所以$maxprofit$的值不變
- 判斷出本節點是nonpromising因為它的$weight = 17$大於16(W的值)
- 不需要算出bound值，因為它的weight值已經讓本節點變成nonpromising的節點了

6. Backtrack回(2,1)節點。

7.~21.請參照課本繼續下列步驟

演算法 5.7 0-1 Knapsack 問題

- **問題**：給定*n*個物件以及它們個別的weight與profit值。weight與profit都是正整數。此外，*W*值也是給定的。在總重量不超過*W*的條件下，找出一些物件使得它的總獲益是最大的。
- **輸入**：正整數*n*和*W*。陣列索引從1到*n*的兩陣列*w*與*p*，其中兩個陣列都儲存了正整數而且是根據*p[i]/w[i]*由大到小排序。
- **輸出**：輸出結果是索引為1到*n*的 *bestset* 陣列，其中*bestset[i]*值是yes就代表要拿第*i*個物品，no就代表不拿第*i*個物品。正整數*maxprofit*，也就是最大獲益。

```
void knapsack (index i,
               int profit, int weight)
{
  if (weight <=W && profit > maxprofit) {
     maxprofit = profit;              // 如果這個集合是目前最好的
     numbest = i;
     bestset = include;               // 將 numbest 設成目前考慮的物品個數
  }                                   // 將 bestset 設成這個解

  if (promising (i)) {
     include [i + 1] = "yes";                 // 拿 w[i + 1].
     knapsack (i + 1, profit + p[i + 1], weight + w[i + 1]);
     include [i + 1] = "no";                  // 不拿 w[i + 1].
     knapsack (i + 1, profit, weight);
  }
}
```

```
bool promising (index i)
{
   index j, k;
   int totweight;
   float bound;

   if (weight >= W)                        // 只有當我們必須擴展子節點時
      return false;                        // 這個節點才是 promising
   else {                                  // 一定還要有空間容納其他物品
      j = i + 1;
      bound = profit;
      totweight = weight;
      while (j <= n && totweight + w[j] <= W){
        totweight = totweight + w[j];      // 盡可能拿越多物品越好
        bound = bound + p[j];
        j++;
      }
      k = j;                       // 使用 k 以保持與文章中方程式的一致性
      if (k <=n)                   // 拿一部分第 k 個物品
         bound = bound + (W - totweight) * p[k]/w[k];
      return bound > maxprofit; // item.
   }
}
```

5.7.2
比較使用Dynamic Programming與回溯演算法來解決0-1背包問題的效率

- 5.4 sum-of-subset
  - https://vimeo.com/715823622/5c22c0c602
- 5.5 coloring
  - https://vimeo.com/715834975/2956179ee6
- 5.6 漢彌爾頓迴路
  - https://vimeo.com/715838652/c30df44c7b