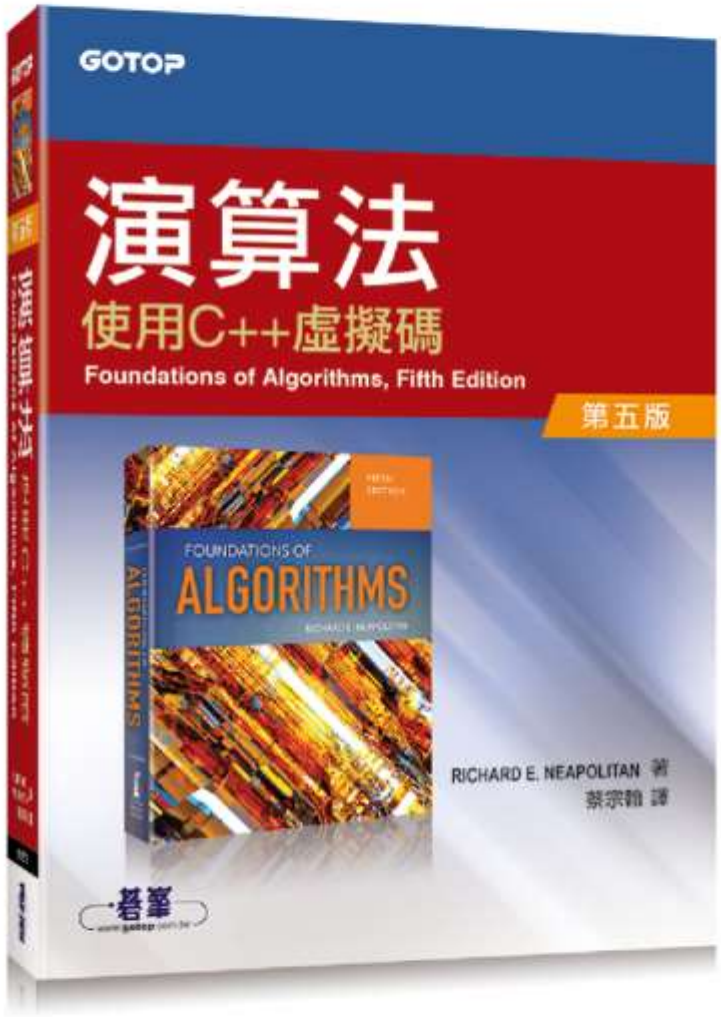


第六章
Branch-and-Bound

演算法
使用C++虛擬碼
Foundations of Algorithms, Fifth Edition
第五版

第六章 Branch-and-Bound

6.1 以0-1 背包問題說明Branch-and-Bound法

- 藉著將branch-and-bound應用在解0-1 Knapsack問題上，我們教您如何使用這種演算法設計策略。
- 首先，我們討論一個稱為“使用branch-and-bound修剪法之廣度優先搜尋(breadth-first search)”的簡易版本。
- 之後，我們針對這個簡易版本做些修正。修正版稱為“使用branch-and-bound修剪法之最佳優先搜尋(best-first search)”。

depth-first search (DFS) 演算法

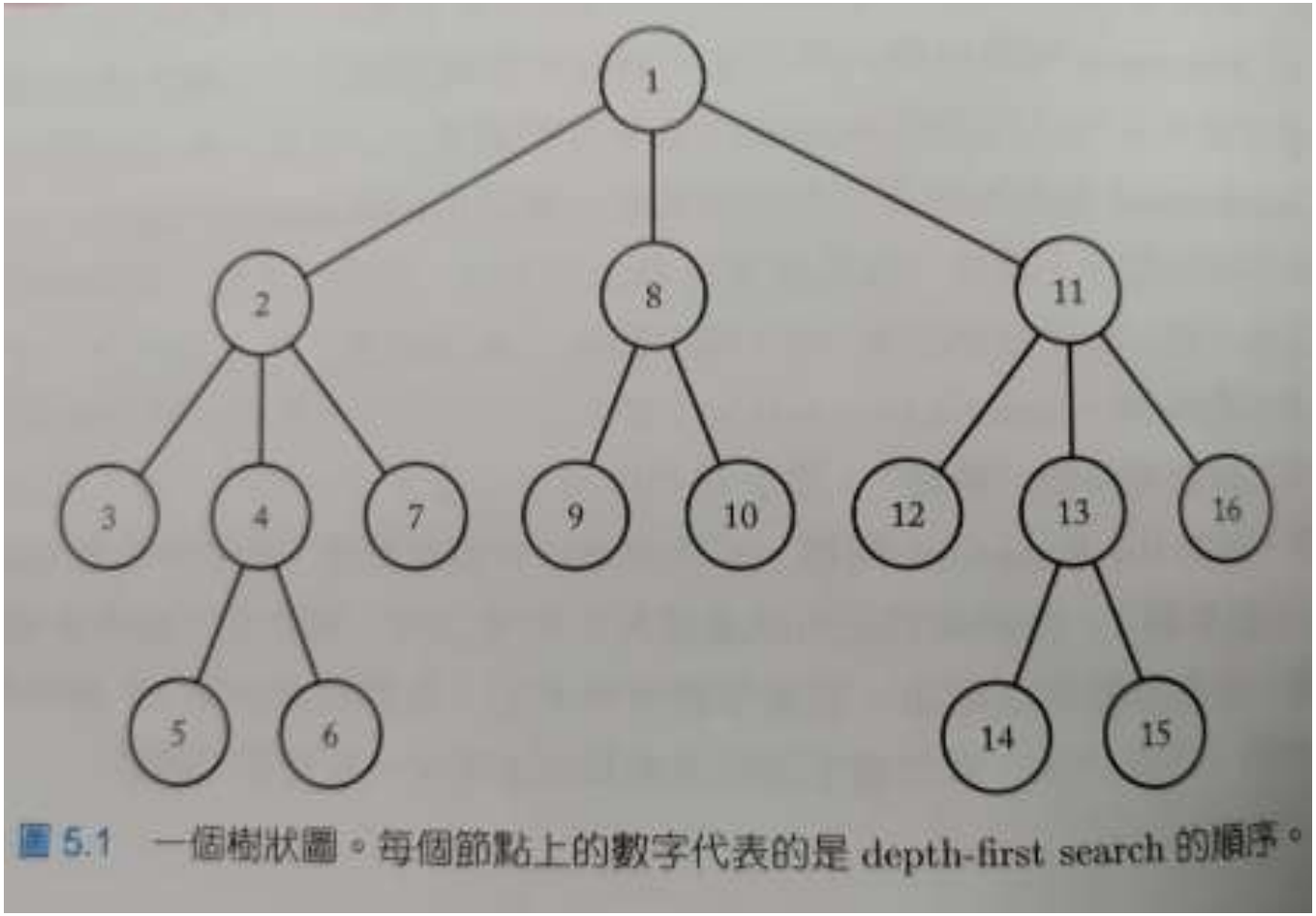


圖 5.1 一個樹狀圖。每個節點上的數字代表的是 depth-first search 的順序。

Breadth-first search (BFS) 演算法

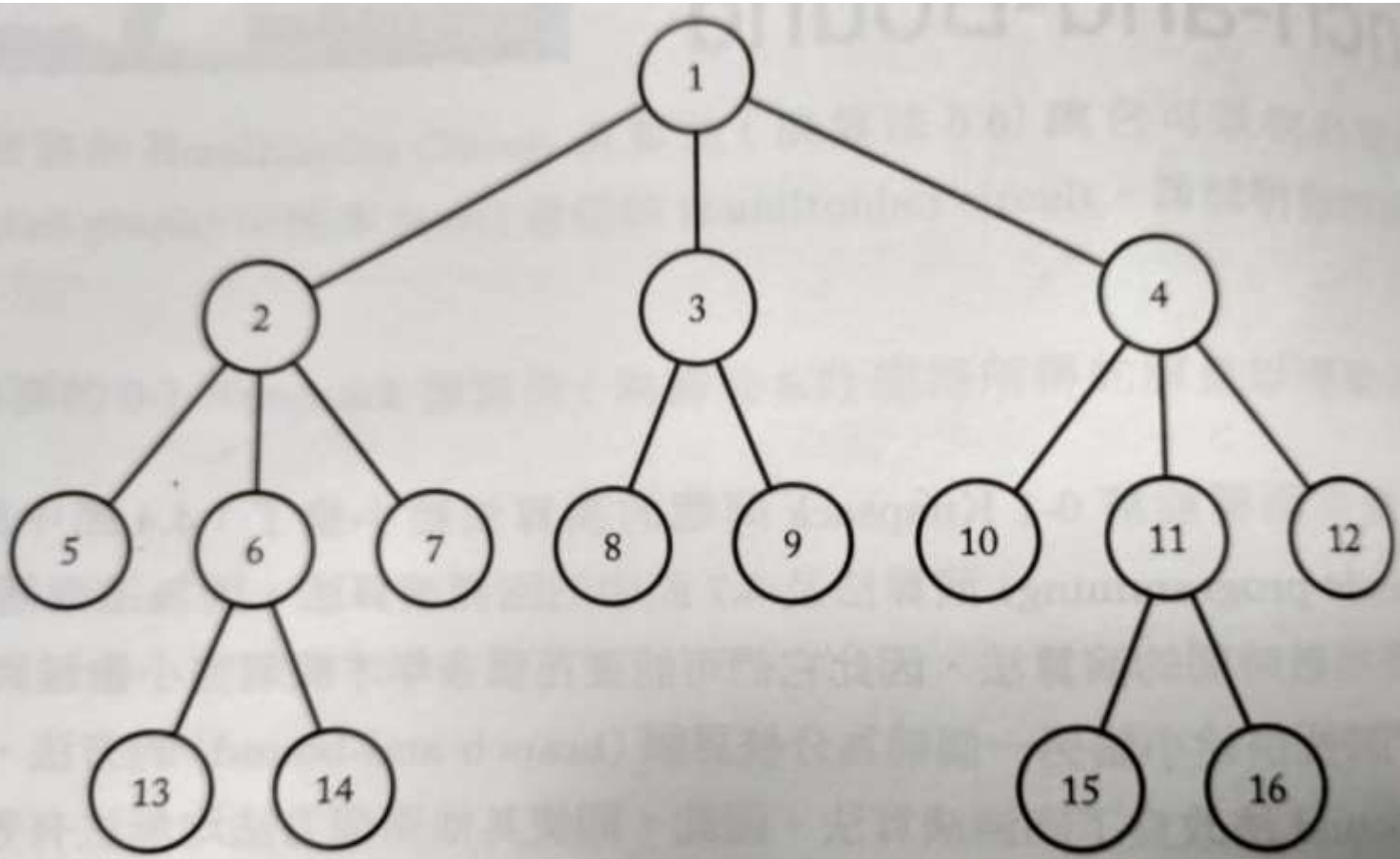


圖 6.1 對樹進行的一次廣度優先搜尋。各節點上的數字是它們被走訪的順位。在廣度優先搜尋中，我們會由左到右走訪一個節點的子節點。

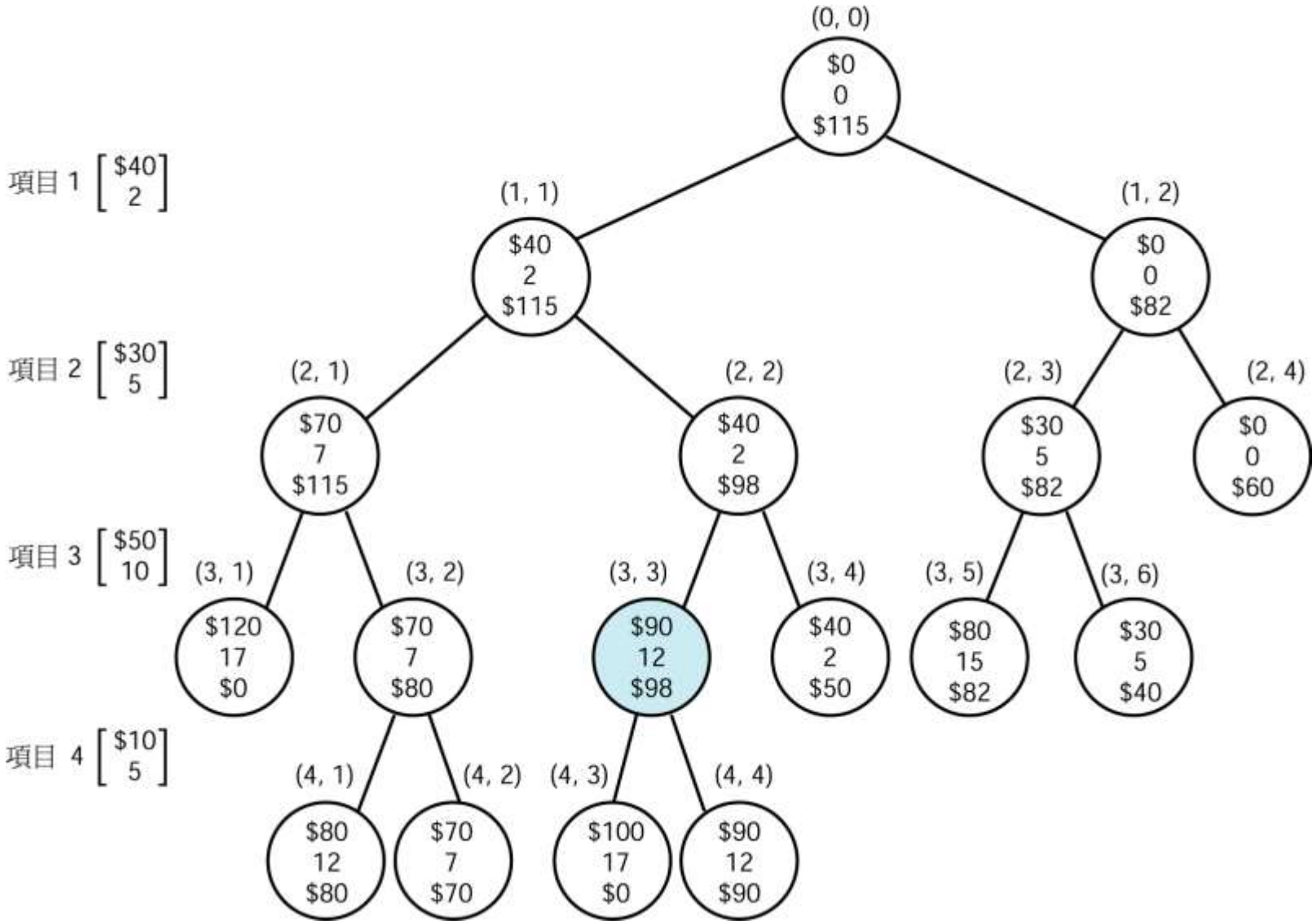
6.1.1 使用Branch-and-Bound修剪法之廣度優先搜尋

- **範例6.1**

  假設我們遭遇的是範例5.6中提過的0-1 Knapsack問題實例。也就是說，$n = 4$，$W = 16$，此外，我們還有下面的資料：

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

圖6.2 範例6.1中使用branch-and-bound修剪法之廣度優先搜尋而產生的已修剪狀態空間樹



- 節點(層級, 左邊算過來的位置)
- 節點(3,1)與(4,3)的bound為$0
  - 由檢查一個節點的bound是否優於目前找到最佳解的值來決定是否要展開這個節點
  - 因此，當一個節點為nonpromising，由於它的weight不比*W*小，因此我們把它的bound設為0
  - 以這種方式，我們確保這個節點的bound不會優於目前為止找到最佳解的值。
- 節點(1,2)被走訪的時候，*maxprofit*的值僅為$40。由於它的bound $82超過此時的*maxprofit*，因此我們展開該節點。

- 走訪某節點時決定是否要走訪其子節點
  - 例如，在走訪節點(2,3)時，我們決定要走訪它的子節點，因為在那個時候，*maxprofit*的值只有$70，而該節點的bound為$82。
  - 與深度優先搜尋不同的是，在廣度優先搜尋中，*maxprofit*的值可以在我們真的走訪到該子節點之前改變。
  - 在這個例子中，*maxprofit*在我們走訪(2,3)的子節點之前就已經變成$90了。

演算法6.1

用來解決0-1 Knapsack問題的

“使用Branch-and-Bound修剪法之廣度優先搜尋”

- 問題：給定*n*個項目，其中每個項目都有自己的weight與profit。這些weight與profit均為正整數。再者，令正整數*W*也是給定的。試求出一組具有最大profit總和的項目，其中這些項目的weight總和不得超過*W*。
- 輸入：正整數*n*與*W*，正整數陣列*w*與*p*，索引值均由1到 *n*。且這兩個陣列都已根據 的值，依非遞減的順序排好。
- 輸出：整數*maxprofit*，也就是最佳組中各項目之profit的總和。

```
void knapsack2 (int n,
                const int p [], const int w[],
                int W,
                int& maxprofit)
{
  queue_of_node Q;
   node u, v;

   initialize (Q);                              // 啟始時將 Q 清空
   v.level = 0; v.profit = 0; v.weight = 0;
                                                // 啟始時將 v 設為根節點
   maxprofit = 0;
   enqueue (Q, v);
   while (! empty (Q)) {
      dequeue (Q, v);
      u.level = v.level + 1;                    // 將 u 設成 v 的子節點之一
      u.weight = v.weight + w[u.level];         // 將 u 設成包含下個項目的子節點
      u.profit = v.profit + p [u.level];
      if (u.weight <= W && u.profit > maxprofit)
         maxprofit = u.profit;
      if (bound (u) > maxprofit)
         enqueue (Q, u);
      u.weight = v.weight;                      // 將 u 設成不含下個項目的子節點
      u.profit = v.profit;
      if (bound (u) > maxprofit)
          enqueue (Q, u);
   }
}
```

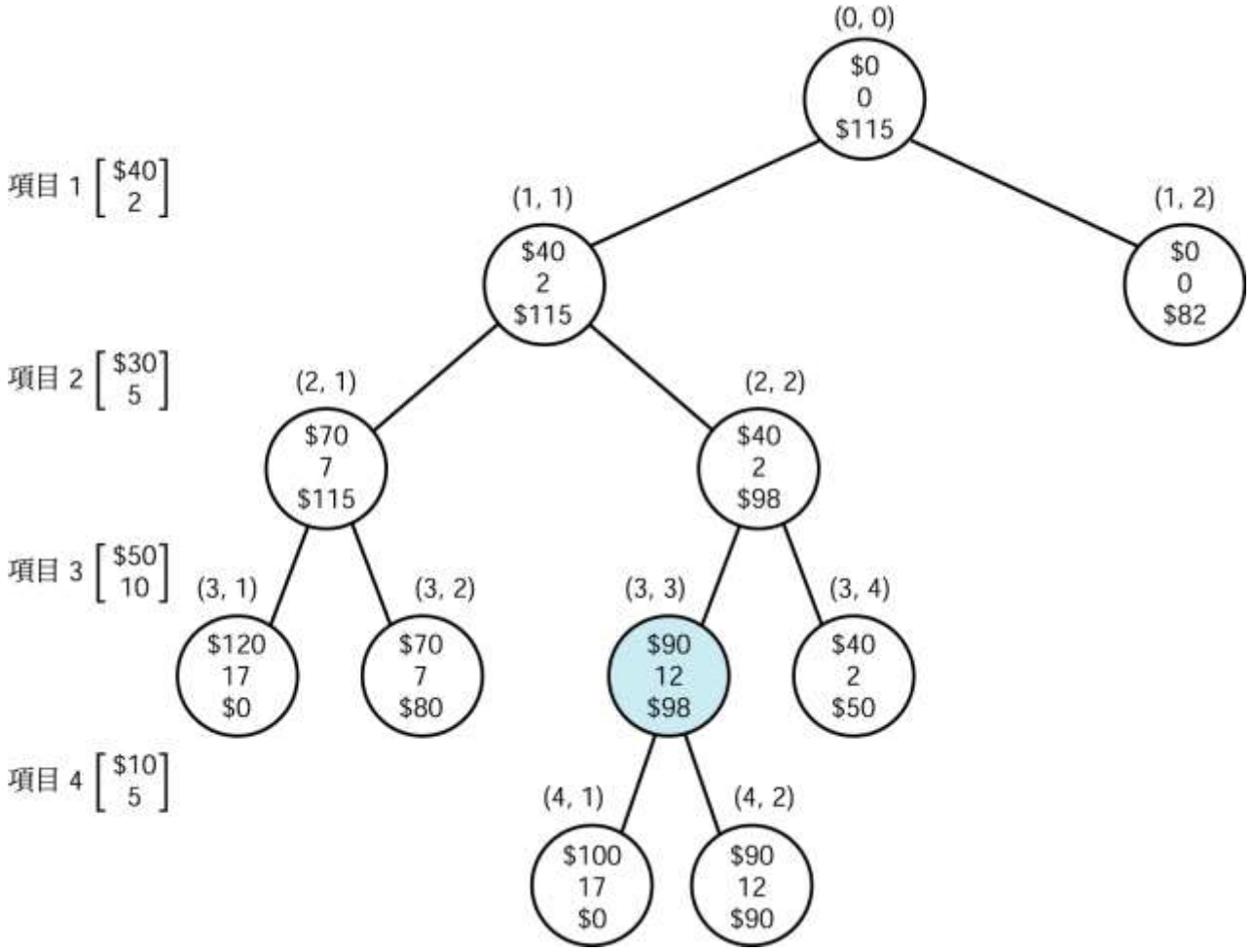
```
float bound (node u)
{
  index j, k;
  int totweight;
  float result;

  if (u.weight >=W)
     return 0;
  else {
     result = u.profit;
     j = u.level + 1;
     totweight = u.weight;
     while (j <= n && totweight + w[j] <= W){
       totweight = totweight + w[j];          // 盡可能地多抓一些項目
        result = result + p [j];
        j++;
      }
      k = j; // 使用 k 以便與文章中的方程式保持一致
      if (k <=n)
        result = result + (W - totweight) * p [k] /w[k];
                                          // 抓住第 k 項的部分
      return result;
  }
}
```

6.1.2 使用Branch-and-Bound修剪法之最佳優先搜尋

- 在一般情況下，與深度優先搜尋(回溯)比較起來，廣度優先搜尋並不佔有優勢。然而，我們可以改進廣度優先搜尋，讓bound不只是決定一個節點是否promising
- 在走訪給定節點的所有子節點之後，我們可以檢視所有未被展開的promising節點，從中挑出具有最佳bound的節點展開
- 回想一個節點被認為是promising的定義：該節點的bound必須優於到目前為止找到最佳解的值。利用這種方式，通常會比遵循預定順序來前進到最佳解的速度要快。下面的範例就是在描述這種改良方法

範例6.2
範例6.1中的0-1 Knapsack問題案例

- 最佳優先搜尋產生了圖6.3中的已修剪狀態空間樹



產生這棵樹的步驟如下：

- 走訪節點(0,0) (根節點)。
  - 將它的profit及weight分別設為$0與0。
  - 計算出它的bound為$115。(這段計算請參考範例5.6)
  - 將*maxprofit*設為0。
- 走訪節點(1,1)。
  - 計算出它的profit及weight分別為$40與2。
  - 由於2小於等於16(*W*的值)，且它的profit $40比$0(*maxprofit*的值)大，因此將*maxprofit*設為$40。
  - 計算出它的bound為$115。
- 走訪節點(1,2)。
  - 計算出它的profit及weight分別為$0與0。
  - 計算出它的bound為$82。

- 找出具有最大bound之promising且尚未被展開的節點
  - 由於節點(1,1)之bound為$115，且節點(1,2)之bound為$82，因此節點(1,1)就是那個具有最大bound之promising且尚未被展開的節點。接著我們就走訪它的子節點
- 走訪節點(2,2)
  - 計算出它的profit及weight分別為$70與7。
  - 由於它的weight 7小於等於16(*W*的值)，且它的profit $70比$40(*maxprofit*的值)大，因此將*maxprofit*設為$70
  - 計算出它的bound為$115。(這段計算請參考範例5.6)
- 走訪節點(2,2)
  - 計算出它的profit及weight分別為$40與2
  - 計算出它的bound為$98

- 找出具有最大bound之promising且尚未被展開的節點。
  - 答案就是節點(2,1)。接著我們就走訪它的子節點。
- 走訪節點(3,1)。
  - 計算出它的profit及weight分別為$120與17。
  - 發現它是nonprimising的，因為它的weight 17大於16(*W*的值)，且它的profit $70比$40(*maxprofit*的值)大，我們將它的bound設為$0讓它變成nonpromising。
- 走訪節點(3,2)。
  - 計算出它的profit及weight分別為$70與7。
  - 計算出它的bound為$80。

- 找出具有最大bound之promising且尚未被展開的節點。
  - 答案就是節點(2,2)。接著我們就走訪它的子節點。
- 走訪節點(3,3)。
  - 計算出它的profit及weight分別為$90與12。
  - 由於它的weight 12小於等於16(*W*的值)，且它的profit $90比$70(*maxprofit*的值)大，因此將*maxprofit*設為$90。
  - 在這個時間點，節點(1,2)與(3,2)變成nonpromising，因為它們的bound $82與$80都小於等於$90(最新的*maxprofit*值)。
  - 計算出它的bound為$98。
- 走訪節點(3,4)。
  - 計算出它的profit及weight分別為$40與2。
  - 計算出它的bound為$50。
  - 發現它變成nonpromising，因為它的bound $50小於等於$90(*maxprofit*的值)。

- 找出具有最大bound之promising且尚未被展開的節點。
  - 剩下唯一的promising且尚未被展開的節點就是節點(3,3)了。接著我們就走訪它的子節點。
- 走訪節點(4,1)。
  - 計算出它的profit及weight分別為$100與17。
  - 發現它變成nonpromising，因為它的weight 17大於等於16(*W*的值)。我們將它的bound設為$0讓它變成nonpromising。
- 走訪節點(4,2)。
  - 計算出它的profit及weight分別為$90與12。
  - 計算出它的bound為$90。
  - 發現它變成nonpromising，因為它的bound $90小於等於$90(*maxprofit*的值)。這棵狀態空間樹的leaf自動成為nonpromising，因為它們的bound不可能會超過*maxprofit*。
- 由於已經沒有promising且未展開的節點了，因此搜尋過程到此結束

演算法6.2

用來解決0-1 Knapsack問題的

“使用Branch-and-Bound修剪法之最佳優先搜尋”

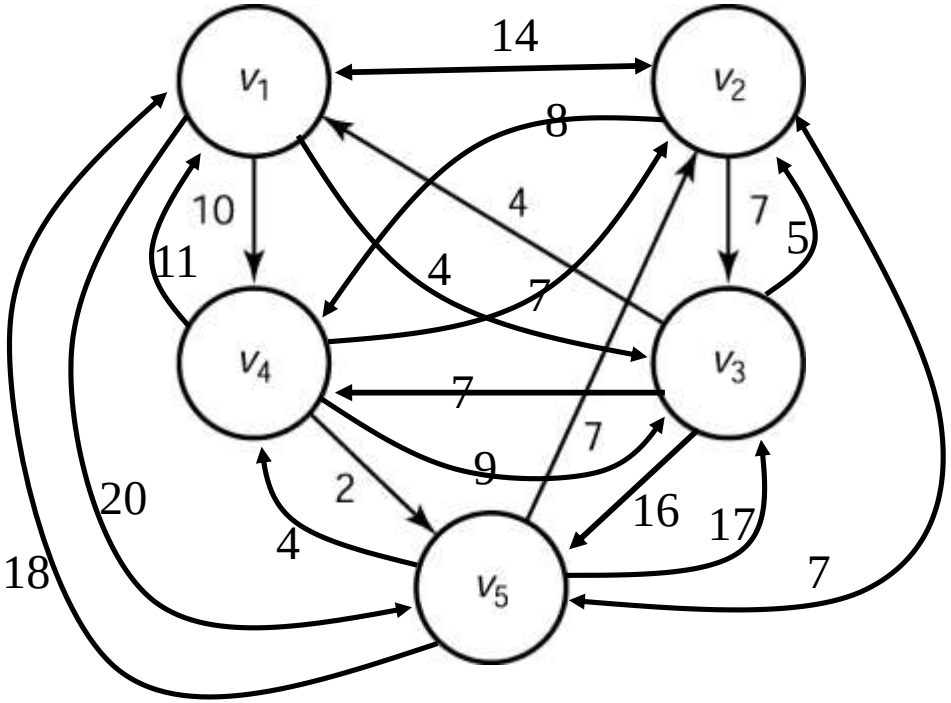
- 問題：給定*n*個項目，其中每個項目都有自己的weight與profit。這些weight與profit均為正整數。再者，令正整數*W*也是給定的。試求出一組具有最大profit總和的項目，其中這些項目的weight總和不得超過*W*。
- 輸入：正整數*n*與*W*，正整數陣列*w*與*p*，索引均由1到 *n*。且這兩個陣列都已根據 的值，依非遞減的順序排好。
- 輸出：整數*maxprofit*，也就是最佳組中各項目之profit的總和。

```
void knapsack3 (int n,
                const int p [], const int w[],
                int W,
                int& maxprofit)
{
  priority queue_of_node PQ;
  node u, v;

  initialize (PQ);                          // 啟始時將 PQ 清空
  v.level = 0; v.profit = 0; v.weight = 0;
  maxprofit = 0;                            // 啟始時將 v 設為根節點
  v.bound = bound (v);
  insert(PQ, v);
  while (! empty (PQ)) {                    // 將具有最佳 bound 的節點移除
     remove (PQ, v);
     if (v.bound > maxprofit) {             // 檢查這個節點是否仍然 promising
        u.level = v.level + 1;
        u.weight = v.weight + w[u.level]; // 將 u 設成包含下個項目的子節點
        u.profit = v.profit + p [u.level];

        if (u.weight <=W && u.profit > maxprofit)
           maxprofit = u.profit;
        u.bound = bound (u);
        if (u.bound > maxprofit)
           insert(PQ, u);
        u.weight = v.weight;                // 將 u 設成不含下個項目的子節點
        u.profit = v.profit;
        u.bound = bound (u);
       if (u.bound > maxprofit)
          insert(PQ, u);
     }
  }
}
```

6.2 售貨員旅行問題

- 圖6.4 左邊是鄰接矩陣(adjacency matrix)

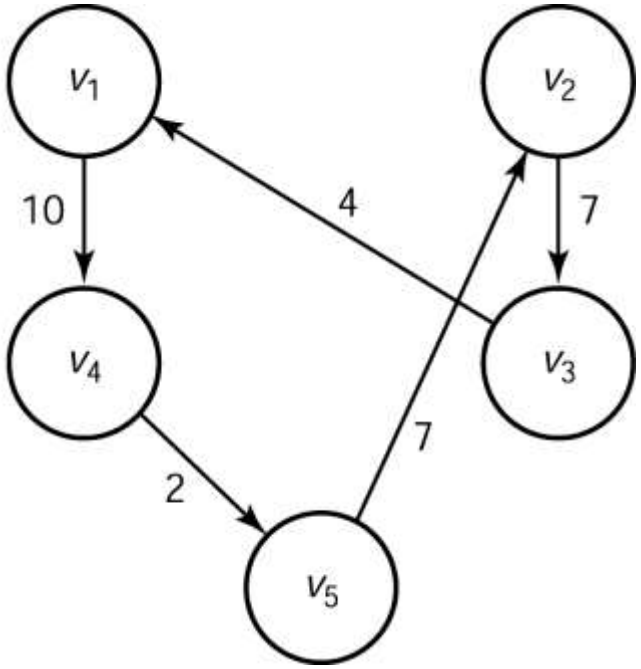
	V1	V2	V3	V4	V5
V1	0	14	4	10	20
V2	14	0	7	8	7
V3	4	5	0	7	16
V4	11	7	9	0	2
V5	18	7	17	4	0



6.2 售貨員旅行問題

- 圖6.4 左邊是每個頂點間均有邊線連接的圖。右邊則是此圖中的節點及最佳旅程中的邊線

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$



最佳優先搜尋

- 在售貨員旅行問題中，我們必須求出展開給定節點得到的所有旅程的長度下限，只有在節點的bound小於目前最小的長度時，才稱該節點為promising
- 計算bound的方法如下。在任意旅程中，當離開一個頂點所採用邊線的長度必須至少與由該頂點發出之最短邊線長度相等

- 因此，可從鄰近矩陣第一列所有非0項目中挑出一個最小的，這個最小項目的值就是離開頂點v_1的*cost*下限
- 我們可從鄰近矩陣第二列所有非0項目中挑出一個最小的，這個最小項目的值就是離開頂點 v_2 的*cost*下限，依此類推。離開圖6.4的5個頂點之*cost*下限如下所示：

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$

$$\begin{aligned} v_1 \quad & minimum \quad (14;4;10;20) = 4 \\ v_2 \quad & minimum \quad (14;7;8;7) = 7 \\ v_3 \quad & minimum \quad (4;5;7;16) = 4 \\ v_4 \quad & minimum \quad (11;7;9;2) = 2 \\ v_5 \quad & minimum \quad (18;7;17;4) = 4 \end{aligned}$$

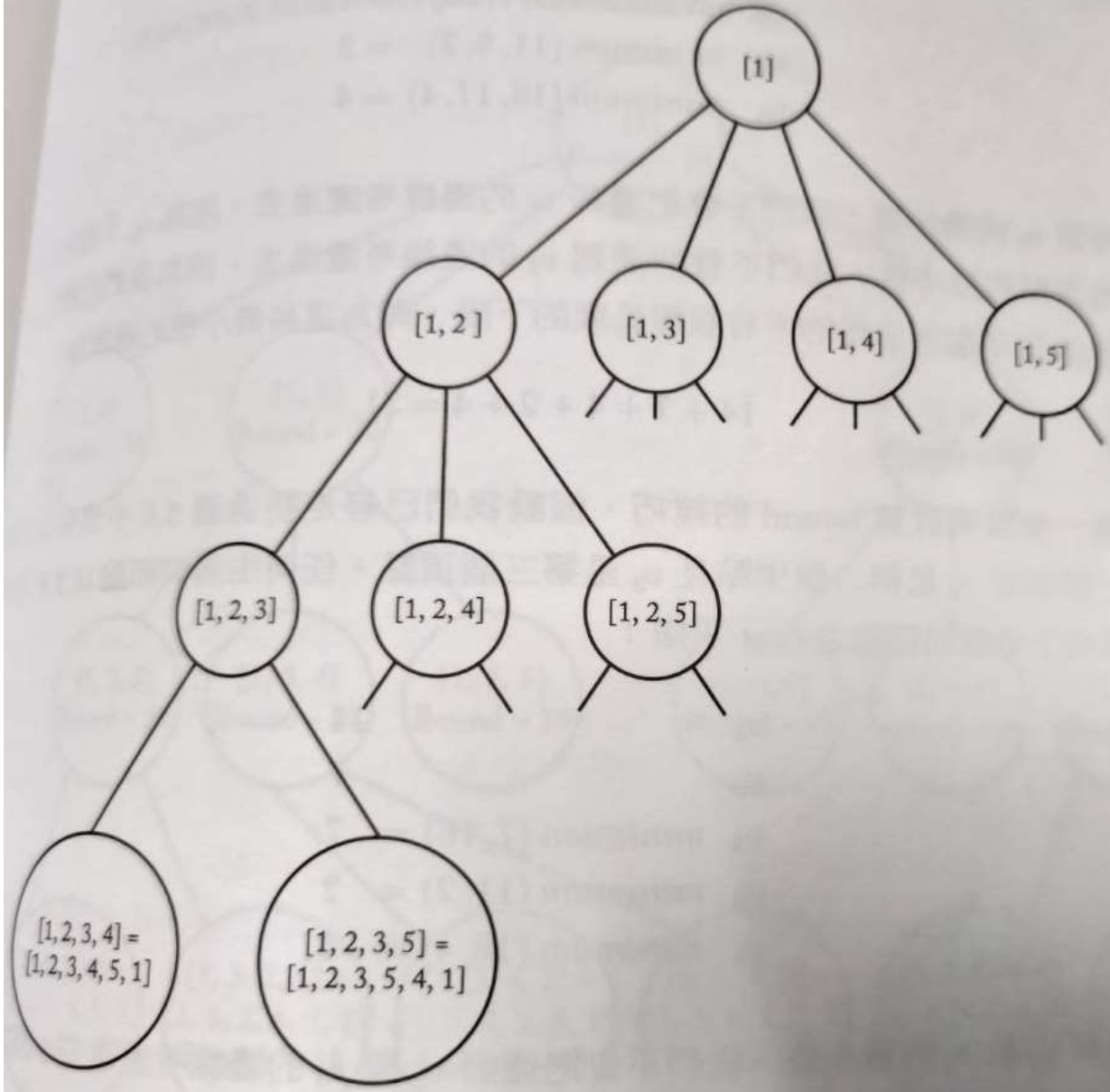


圖 6.5 具有 5 個頂點之售貨員旅行問題對應的狀態空間樹。部份旅程中的頂點索引儲存在每個節點中。

- 由於一個旅程必須離開每個頂點至少一次，因此一個旅程長度的下限就是這些最小值(minimum)的總和，故一個旅程長度的下限為

$$4 + 7 + 4 + 2 + 4 = 21$$

假設我們已經走訪過圖6.5中含有[1,2]的節點。在這個情況中，我們已經確定v_2是旅程中第二個頂點，而抵達v_1的$cost$就是由v_1到v_2那條邊線的weight，也就是14。因此，任何由展開節點[1,2]而得到的旅程，均具有下列離開頂點之cost下限：

$$\begin{array}{lll} v_1 & & 14 \\ v_2 & minimum\ (7;8;7) & = 7 \\ v_3 & minimum\ (4;7;16) & = 4 \\ v_4 & minimum\ (11;9;2) & = 2 \\ v_5 & minimum\ (18;7;17;4) & = 4 \end{array}$$

- 若要得到v_2的最小值，我們不會把通到v_1的邊線考慮進去，因為v_2不能回到v_1。若要得到其他頂點的最小值，我們不會把通到v_2的邊線考慮進去，因為我們已經在v_2了。展開含有[1,2]的節點所求得的所有旅程長度的下限，即為這些最小值的總和為：

$$14 + 7 + 4 + 2 + 4 = 31$$

進一步說明計算bound的技巧

- 假設我們已經走訪過圖6.5中含有[1,2,3]的節點。我們已經確定v_2是第二個頂點且v_3是第三個頂點。任何由展開節點[1,2,3]而得到的旅程，均具有下列離開頂點之cost下限：

$$
\begin{aligned}
v_1 & & & = 14 \\
v_2 & & & = 7 \\
v_3 & \mathit{minimum} & (7;16) & = 7 \\
v_4 & \mathit{minimum} & (11;2) & = 2 \\
v_5 & \mathit{minimum} & (18;4) & = 4
\end{aligned}
$$

- 若要得到v_4與v_5的最小值，我們不會把通到v_2與v_3的邊線考慮進去，因為我們已經去過這些頂點了。展開含有[1,2,3]的節點所求得的所有旅程長度的下限為：

$$14 + 7 + 7 + 2 + 4 = 34$$

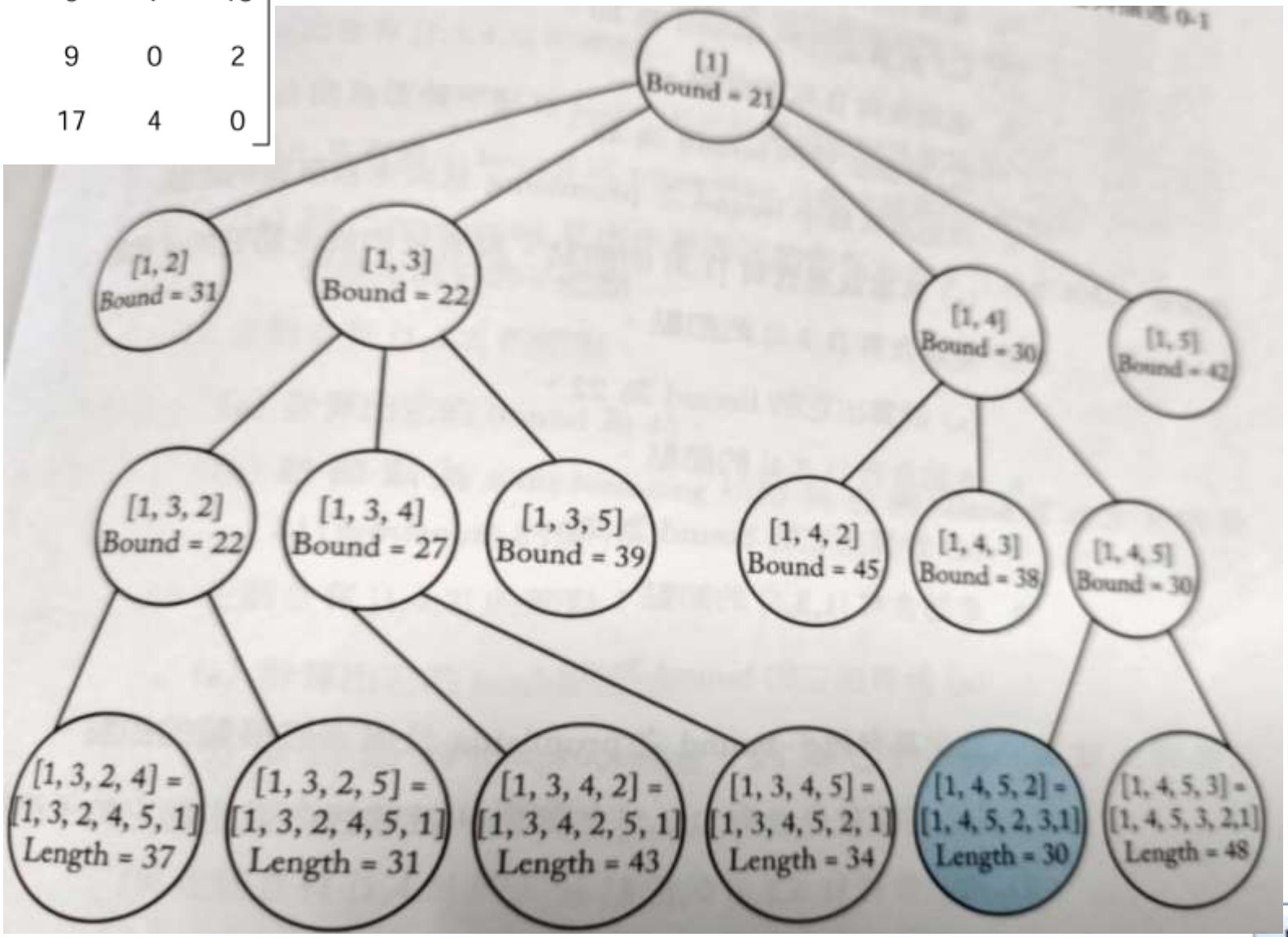
- 用同樣的方式，我們可以算出展開狀態空間樹中任一個節點所求得的所有旅程長度的下限

範例6.3

- 給定圖6.4中的圖(graph)並使用前面剛剛說明的bound計算方式，使用branch-and-bound修剪法的最佳優先搜尋會產生圖6.6中的樹

圖6.6

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



產生這棵樹的步驟

1. 走訪含有 的節點(根節點)
   - 計算出它的bound為21{這是一個旅程的長度下限}
   - 將*minlength*設為∞
2. 走訪含有[1,2]的節點
   - 計算出它的bound為31
3. 走訪含有[1,3]的節點
   - 計算出它的bound為22
4. 走訪含有[1,4]的節點
   - 計算出它的bound為30
5. 走訪含有[1,5]的節點
   - 計算出它的bound為42

6. 找出具有最小bound之promising且尚未被展開的節點
   - 答案就是含有[1,3]的節點接著我們就走訪它的子節點
7. 走訪含有 [1,3,2]的節點
   - 計算出它的bound為22
8. 走訪含有[1,3,4]的節點
   - 計算出它的bound為27
9. 走訪含有[1,3,5]的節點
   - 計算出它的bound為39
10. 找出具有最小bound之promising且尚未被展開的節點
    - 答案就是含有[1,3,2]的節點接著我們就走訪它的子節點
11. 走訪含有[1,3,2,4]的節點
    - 因為這個節點為leaf，因此計算出旅程長度為37
    - 由於長度37小於 (*minlength*的值)，因此將*minlength*設為37
    - 含有 的節點與含有 的節點變成了nonpromising，因為它們的bound值42與39大於等於37(*minlength*的新值)

12. 走訪含有[1,3,2,5]的節點
    - 因為這個節點為leaf，因此計算出旅程長度為31
    - 由於長度31小於37 (*minlength*的值)，因此將*minlength*設為31
    - 含有[1,2]的節點變成了nonpromising，因為它的bound值31大於等於31(*minlength*的新值)
13. 找出具有最小bound之promising且尚未被展開的節點
    - 答案就是含有 [1,3,4]的節點接著我們就走訪它的子節點
14. 走訪含有[1,3,4,2]的節點
    - 因為這個節點為leaf，因此計算出旅程長度為43
15. 走訪含有[1,3,4,5]的節點
    - 因為這個節點為leaf，因此計算出旅程長度為34
16. 找出具有最小bound之promising且尚未被展開的節點
    - 唯一promising且尚未被展開的節點為含有[1,4]的節點接著我們就走訪它的子節點

17. 走訪含有[1,4,2]的節點
    - 計算出它的bound為45
    - 該節點為nonpromising，因為它的bound值45大於等於31(*minlength*的值)
18. 走訪含有[1,4,3]的節點
    - 計算出它的bound為38
    - 該節點為nonpromising，因為它的bound值38大於等於31(*minlength*的值)
19. 走訪含有[1,4,5]的節點
    - 計算出它的bound為30
20. 找出具有最小bound之promising且尚未被展開的節點
    - 唯一promising且尚未被展開的節點為含有 的節點接著我們就走訪它的子節點
21. 走訪含有[1,4,5,2]的節點
    - 計算出它的bound為30
    - 因為它的長度30小於等於31(*minlength*的值)，故將*minlength*設為30
22. 走訪含有[1,4,5,3]的節點
    - 因為這個節點為leaf，因此計算出旅程長度為48
23. 找出具有最小bound之promising且尚未被展開的節點
    - 已找不到promising且尚未被展開的節點，因此我們已經做完了

我們已經求出含有[1,4,5,2]的節點，也就是代表[1,4,5,2,3,1]這個旅程的節點，含有一條最佳旅程，因此最佳旅程的長度就是30

演算法6.3

用來解決售貨員旅行問題的
“使用Branch-and-Bound修剪法之最佳優先搜尋”

- 問題：在一有向權重圖(weighted, directed graph)中找出一條最佳旅程。weight必須為非負整數
- 輸入：一個有向權重圖，及該圖中的頂點數，*n*。該圖(graph)以一個二維陣列*W*來表示，列與行之索引均由1到*n*，其中*W*[*i*][*j*]為第*i*個頂點連到第*j*個頂點之邊線的weight
- 輸出：變數*minlength*，其值為一條最佳旅程的長度；以及變數*opttour*，其值為一條最佳旅程

```
void travel2 (int n,
              const number W[][],
              ordered-set& opttour,
              number& minlength)
{
  priority queue_of_node PQ;
  node u, v;

  initialize (PQ);                    // 啟始時將 PQ 清空
  v.level = 0;
  v.path = [1];                       // 令第一個頂點成為起點
  v.bound = bound (v);
  minlength = ∞;
  insert(PQ, v);
  while (! empty(PQ)) {
     remove (PQ, v);                  // 將具有最佳 bound 的節點移除
     if (v.bound < minlength){
       u.level = v.level + 1;         // 將 u 設為 v 的子節點
       for (all i such that 2 ≤ i ≤ n && i is not in v.path){
          u.path = v.path;
          put i at the end of u.path;
          if (u.level == n - 2){      // 檢查是否到下個節點就完成一條旅程了
            put index of only vertex
            not in u.path at the end of u.path;
            put 1 at the end of u.path;  // 把第一個頂點放在旅程的末端
            if (length(u) < minlength) {
            //length 函式負責計算這條旅程的長度

            minlength = length(u);
            opttour = u.path;
          }
       }
       else {
       u.bound = bound (u);
       if (u.bound < minlength)
         insert(PQ, u);
       }
     }
  }
}
```