# Chapter 6: Branch-and-Bound

## Contents

## BB (cf. BT)

(1) **Used for optimization problem**

(2) **Traverse the tree variously (breadth-first,  best-first)**

- **Compute a number (bound) at a node**
  - **The number is a bound on the value of the solution that could be obtained by expanding beyond the node.**
  - **Nonpromising: The bound ≤ best solution found so far**

## Breadth-first-search

```
void breadth_first_search (tree T);
{
    queue_of_node Q;
    node u, v;
    initialize(Q);                  // Initialize Q to be empty.
    v = root of T;
    visit v;
    enqueue(Q, v);                  // Insert an item at the end of the Q.
    while (! empty(Q)){
        dequeue(Q, v);              // Remove an item from the front.
        for (each child u of v){
            visit u;
            enqueue(Q, u);
        }
    }
}
```

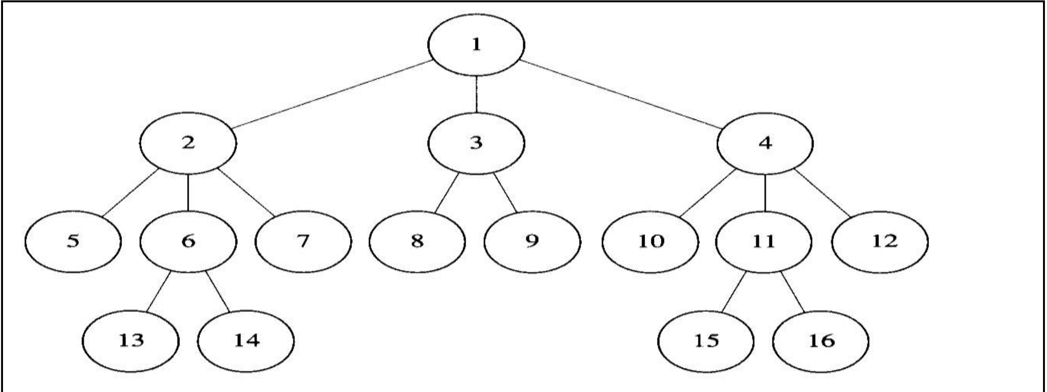


***Figure 6.1*** A breadth-first search of a tree. The nodes are numbered in the order in which they are visited. The children of a node are visited from left to right.

## 6.1 The 0−1 Knapsack Problem

### 6.1.1 Breadth-First Search with Branch-and-Bound Pruning

- **See ex. 6.1 – same as ex. 5.6 – Fig. 6.2**
- **General algorithm: next slide**
- **See Alg. 6.1**
  - **Struct node {**

    **int level; // use level instead of *i* in BB**

    **int profit, weight;**

## General Algorithm

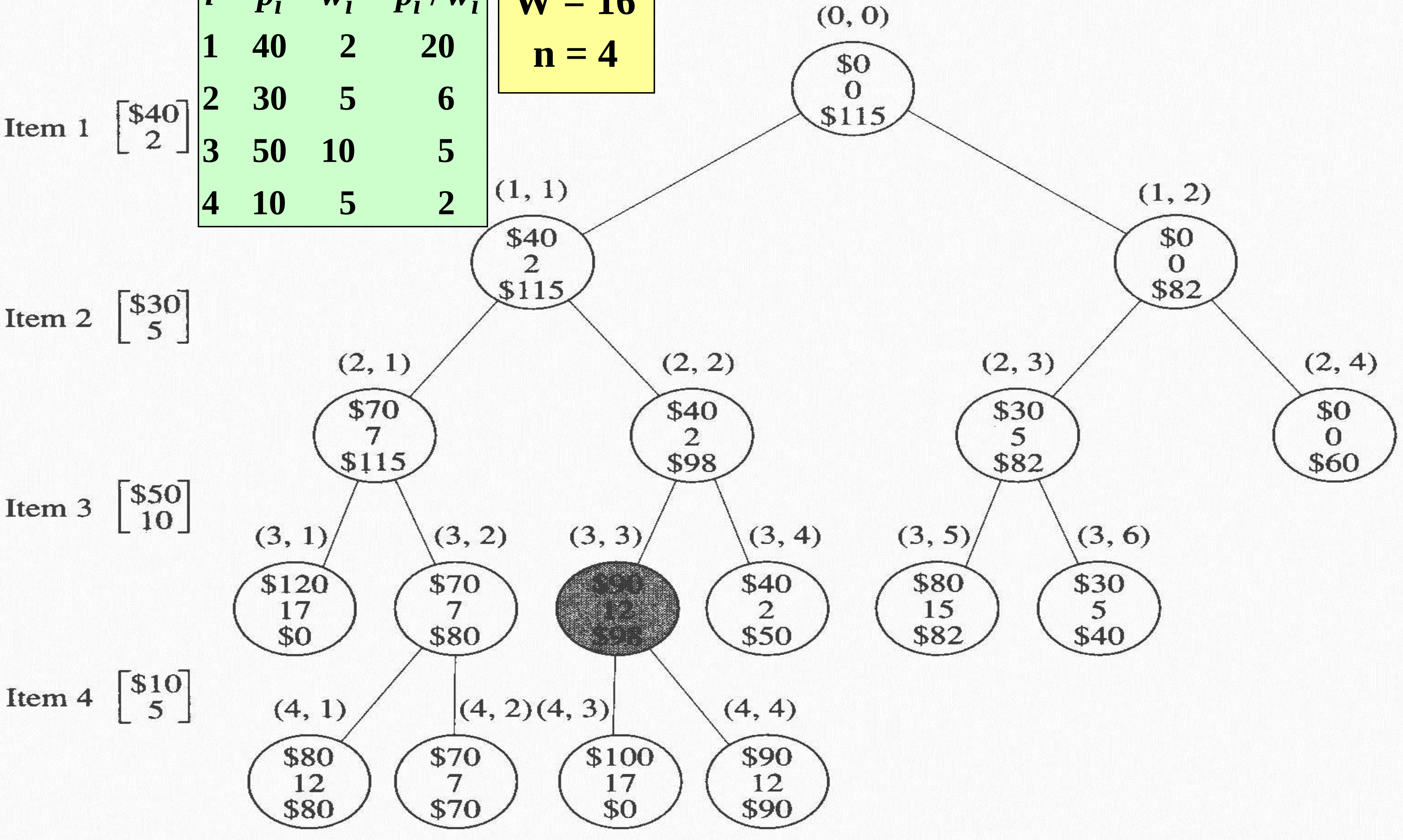
```
void breadth_first_branch_and_bound (state_space_tree T,
                                                    number& best)
{
   queue_of_node Q;
   node u, v;
   initialize(Q);                  // Initialize Q to be empty.
   v = root of T;                  // Visit root.
   enqueue(Q, v);
   best = value(v);
   while (! empty(Q)){
      dequeue(Q, v);
      for (each child u of v){     // Visit each child.
         if (value(u) is better than best)
            best = value(u);
         if (bound(u) is better than best)
            enqueue(Q, u);
      }
   }
}
```

***Figure 6.2*** The pruned state space tree produced using breadth-first search with branch-and-bound pruning in Example 6.1. Stored at each node from top to bottom are the total profit of the items stolen up to that node, their total weight, and the bound on the total profit that could be obtained by expanding beyond the node. The shaded node is the one at which an optimal solution is found.

| $i$ | $p_i$ | $w_i$ | $p_i / w_i$ |
|---|---|---|---|
| 1 | 40 | 2 | 20 |
| 2 | 30 | 5 | 6 |
| 3 | 50 | 10 | 5 |
| 4 | 10 | 5 | 2 |

**W = 16**

**n = 4**



## Algorithm 6.1 The Breadth-FS with BB Pruning Algorithm for the 0-1 Knapsack Problem (1/3)

- **Problem: Let *n* items be given, where each item has a weight and a profit. The weights and profits are positive integers. Furthermore, let a positive integer *W* be given. Determine a set of items with maximum total profit, under the constraint that the sum of their weights cannot exceed *W*.**
- **Inputs: positive integers *n* and *W*, arrays of positive integers *w* and *p*, each indexed from 1 to *n*, and each of which is sorted in nonincreasing order according to the values of *p*[*i*]/*w*[*i*].**
- **Outputs: an integer *maxprofit* that is the sum of the profits in an optimal set.**

## Algorithm 6.1 The Breadth-FS with BB Pruning Algorithm for the 0-1 Knapsack Problem (2/3)

```
void knapsack2 (int n, const int p[ ], const int w[ ], int W, int& maxprofit)
{
   queue_of_node Q;    node u, v;
   initialize(Q);                                 // Initialize Q to be empty.
   v.level = 0;  v.profit = 0;  v.weight = 0;   // Initialize v to be the root.
   maxprofit = 0;    enqueue(Q, v);
   while (! Empty(Q)) {
      dequeue(Q, v);
      u.level = v.level + 1;                      // Set u to a child of v.
      u.weight = v.weight + w[u.level];          // Set u to the child that includes
      u.profit = v.profit + p[u.level];          // the next item.
      if (u.weight <= W && u.profit > maxprofit)
         maxprofit = u.profit;
      if (bound(u) > maxprofit)
         enqueue(Q, u);
      u.weight = v.weight;                        // Set u to the child that does not
      u.profit = v.profit;                        // include the next item.
      if (bound(u) > maxprofit)
         enqueue(Q, u);
   }
}
```

## Algorithm 6.1 The Breadth-FS with BB Pruning Algorithm for the 0-1 Knapsack Problem (3/3)

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j \le W < totweight + w_k$$

```
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) {  // Grab as many items
            totweight = totweight + w[j];                  // as possible.
            result = result + p[j];
            j++;
        }
        k = j;                              // Use k for consistency with formula in text.
        if (k <= n)
            result = result + (W – totweight) * p[k]/w[k]; // Grab fraction of k-th
        return result;                                          // item.
    }
}
```

$$bound = \left( profit + \sum_{j=i+1}^{k-1} p_j \right) + (W - totweight) \times \frac{p_k}{w_k}$$

## 6.1.2 Best-First Search with Branch-and-Bound Pruning

- **Expand first the promising node with the best bound**
- **See ex. 6.2 – Fig. 6.3**
- **Implementation: use priority queue**
  - **General algorithm: next slide**
  - **Struct node {**

    **int level, profit, weight;**

    **float bound;    /* if bound ≤ maxprofit, then discard */**
- **See Alg. 6.2**

## Priority queue (See Section 7.6)

- **The element with the highest priority is always removed next.**
- **A priority queue can be implemented efficiently as a heap.**
- **A complete binary tree**
  - **All internal nodes have two children**
  - **All leaves have depth *d*.**
- **An essentially complete binary tree**
  - **It is a complete binary tree down to a depth of *d*-1**
  - **The nodes with depth *d* are as far to the left as possible**
- **A heap**
  - **The values stored at the nodes come from an ordered set.**
  - **The value stored at each node is greater than or equal to the values stored at its children. This is called heap property.**

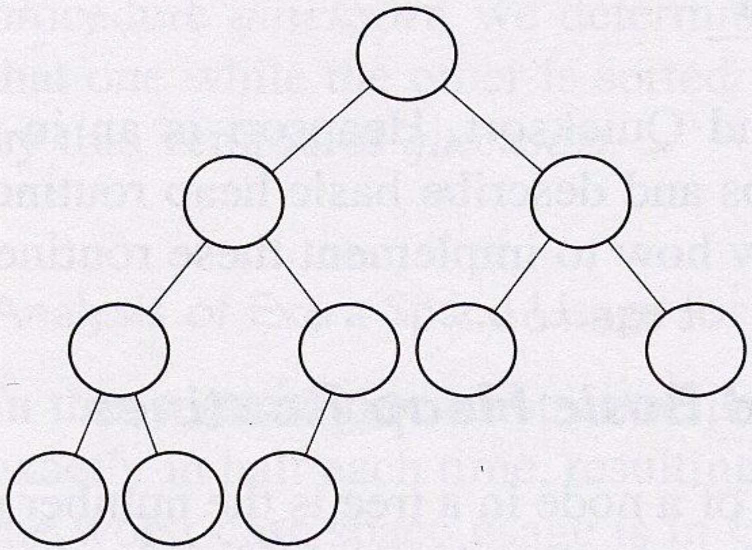


**Figure 7.4** An essentially complete binary tree.

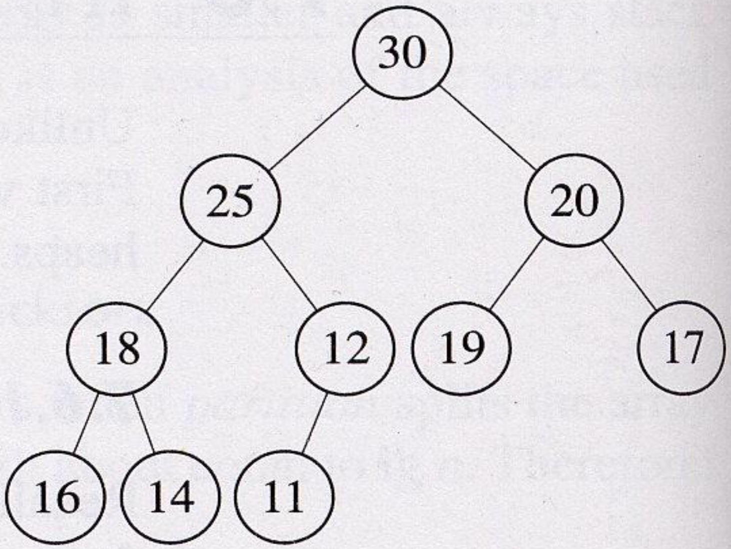
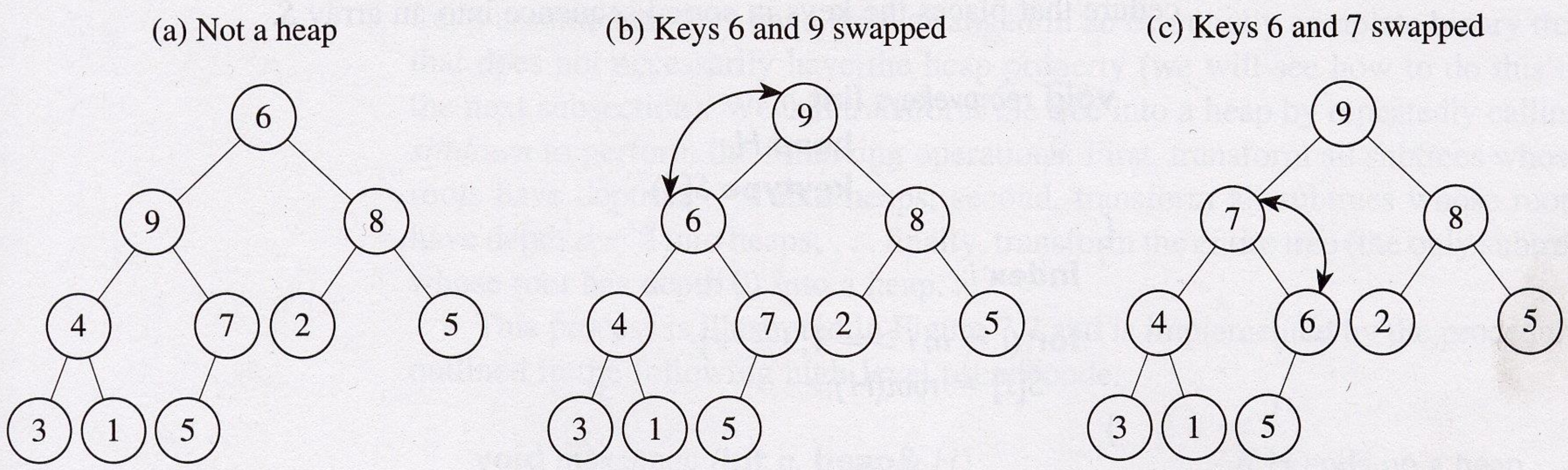


**Figure 7.5** A heap.

**Figure 7.6** Procedure *siftdown* sifts 6 down until the heap property is restored.



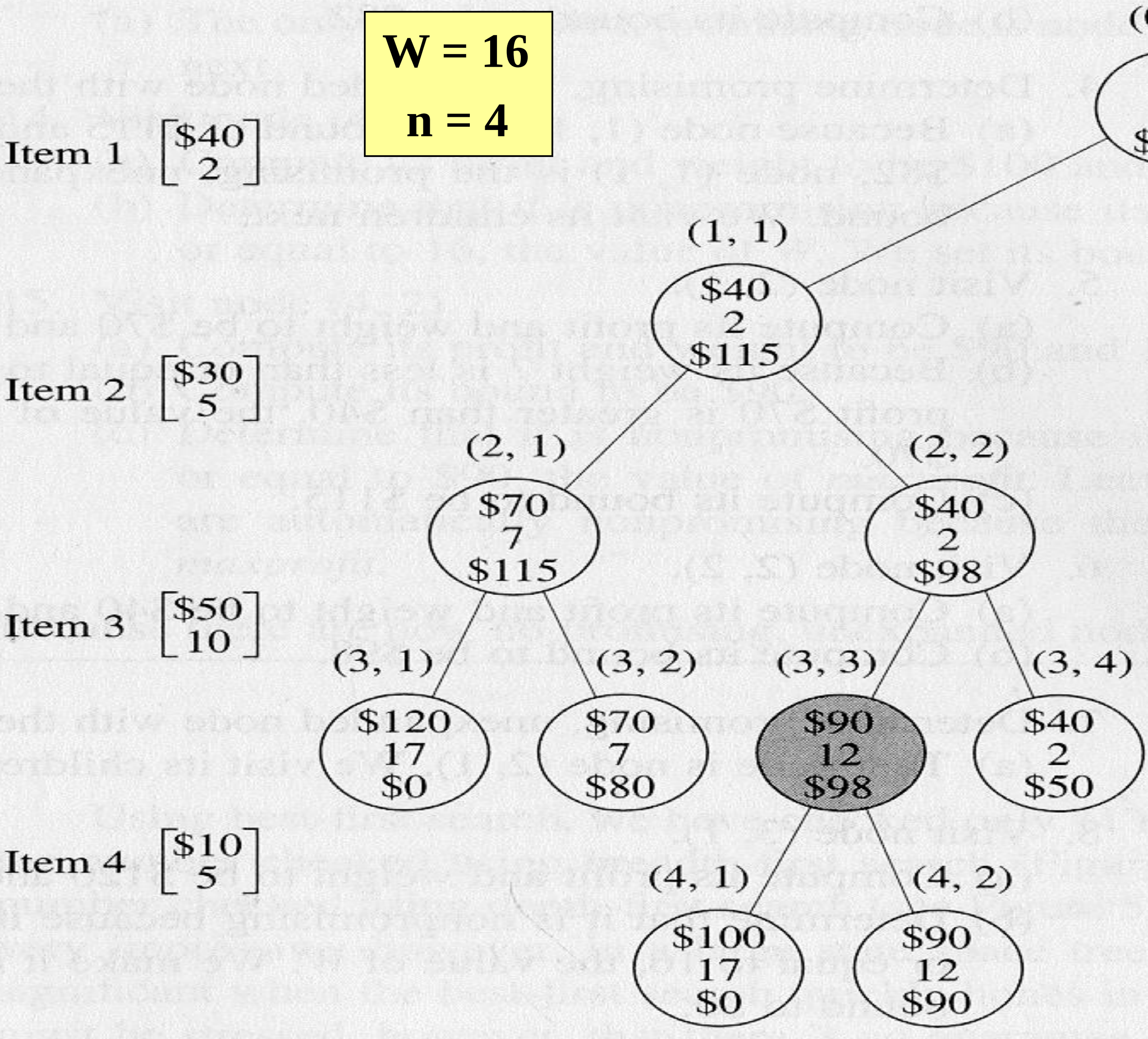
## General Algorithm

```
void best_first_branch_and_bound (state_space_tree T, number& best)
{
    priority_queue_of_node PQ;
    node u, v;
    initialize(PQ);                          // Initialize PQ to be empty.
    v = root of T;
    best = value(v);
    insert(PQ, v);
    while (! empty(PQ)){
        remove(PQ, v);                       // Remove node with best bound.
        if (bound(v) is better than best)    // Check if node is still promising.
            for (each child u of v){
                if (value(u) is better than best)
                    best = value(u);
                if (bound(u) is better than best)
                    insert(PQ, u);
        }
    }
}
```

**Figure 6.3** The pruned state space tree produced using best-first search with branch-and-bound pruning in Example 6.2. Stored at each node from top to bottom are the total profit of the items stolen up to the node, their total weight, and the bound on the total profit that could be obtained by expanding beyond the node. The shaded node is the one at which an optimal solution is found.

**W = 16**

**n = 4**



| $i$ | $p_i$ | $w_i$ | $p_i / w_i$ |
|---|---|---|---|
| 1 | 40 | 2 | 20 |
| 2 | 30 | 5 | 6 |
| 3 | 50 | 10 | 5 |
| 4 | 10 | 5 | 2 |

## Algorithm 6.2 The Best-FS with BB Pruning Algorithm for the 0-1 Knapsack Problem (1/3)

- **Problem: Let *n* items be given, where each item has a weight and a profit. The weights and profits are positive integers. Furthermore, let a positive integer *W* be given. Determine a set of items with maximum total profit, under the constraint that the sum of their weights cannot exceed *W*.**
- **Inputs: positive integers *n* and *W*, arrays of positive integers *w* and *p*, each indexed from 1 to *n*, and each of which is sorted in nonincreasing order according to the values of *p*[*i*]/*w*[*i*].**
- **Outputs: an integer *maxprofit* that is the sum of the profits of an optimal set.**

## Algorithm 6.2 The Best-FS with BB Pruning Algorithm for the 0-1 Knapsack Problem (2/3)

```
void knapsack3 (int n,
                const int p[ ], const int w[ ],
                int W,
                int& maxprofit)
{
   priority_queue_of_node PQ;
   node u, v;
   initialize(PQ);                                  // Initialize PQ to be empty.
   v.level = 0; v.profit = 0; v.weight = 0;  // Initialize v to be the root.
   maxprofit = 0;
   v.bound = bound(v);
   insert(PQ, v);
```

## Algorithm 6.2 The Best-FS with BB Pruning Algorithm for the 0-1 Knapsack Problem (3/3)

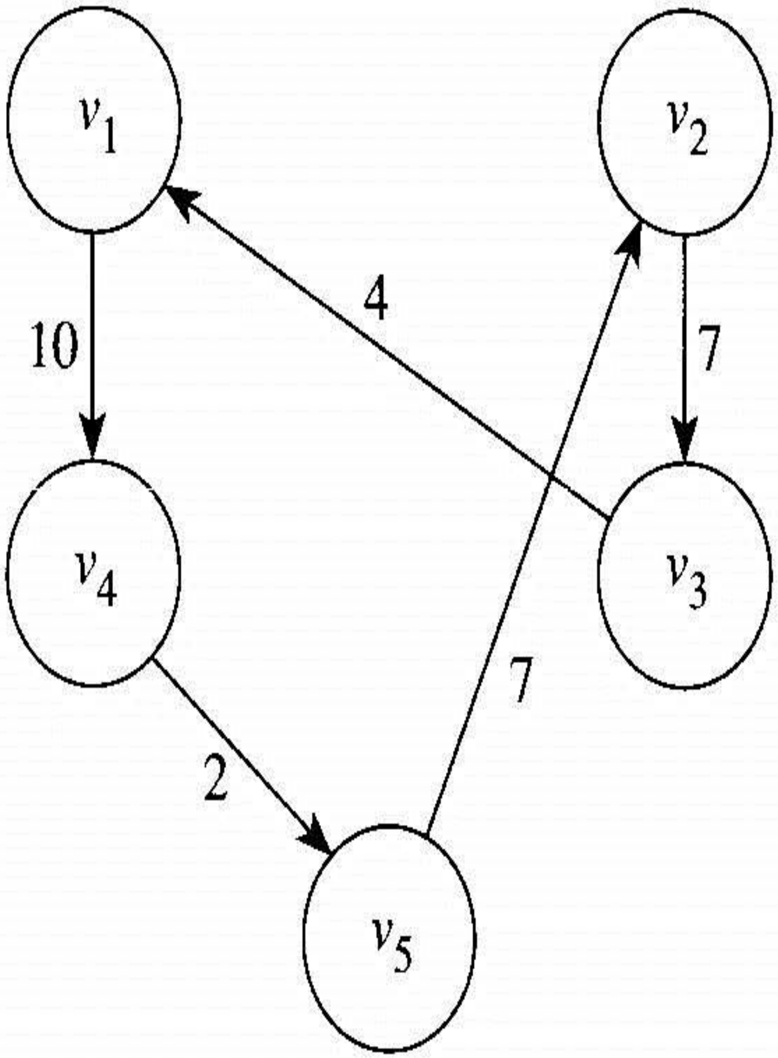
```
while (! empty(PQ)) {
    remove(PQ, v);                          // Remove node with best bound.
    if (v.bound > maxprofit) {  // Check if node is still promising.
        u.level = v.level +1;
        u.weight = v.weight + w[u.level];     // Set u to the child that
        u.profit  = v.profit   + p[u.level];     // includes the next item.
        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;                  // Set u to the child that
        u.profit  = v.profit;                  // does not include the next item.
        u.bound = bound(u);
        if (u.bound > maxprofit)
            insert(PQ, u);
    }
  }
}
```

## 6.2 TSP

- **Goal: find an optimal (minimum cost) tour**
  - **cf.: 0/1 knapsack: maximum profit**
- **Use best-first-search**
- **Pruning**
  - **Compute lower bound (on the length of any tour that can be obtained by expanding beyond a given node)**
  - **The node is nonpromising if**
    - **The bound ≥ min length found so far**
- **How to compute a lower bound**
- **See Fig. 6.4**

***Figure 6.4*** Adjacency matrix representation of a graph that has an edge from every vertex to every other vertex (left), and the nodes in the graph and the edges in an optimal tour (right).

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



# Steps for the Graph in the Figure 6.4 (1/3)

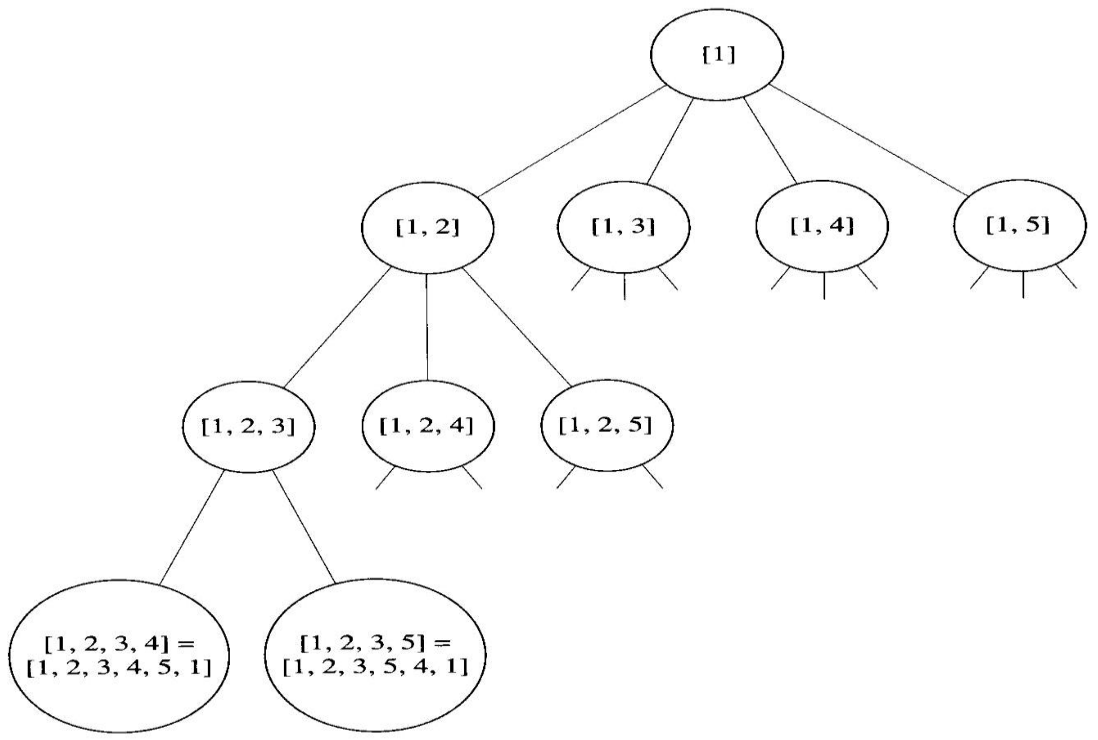
$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 14 & 4 & 10 & 20 \\ 2 & 14 & 0 & 7 & 8 & 7 \\ 3 & 4 & 5 & 0 & 7 & 16 \\ 4 & 11 & 7 & 9 & 0 & 2 \\ 5 & 18 & 7 & 17 & 4 & 0 \end{array}$$

**Root[1]**

$$\mathbf{v}_1 \quad \min(14, 4, 10, 20) = 4$$
$$\mathbf{v}_2 \quad \min(14, 7, 8, 7) = 7$$
$$\mathbf{v}_3 \quad \cdots = 4$$
$$\mathbf{v}_4 \quad \cdots = 2$$
$$\mathbf{v}_5 \quad \cdots = 4$$

$$v_1 \rightarrow \mathrm{O} \rightarrow \mathrm{O} \rightarrow \mathrm{O} \rightarrow \mathrm{O} \rightarrow v_1$$
$$\textbf{bound} = 4 + 7 + 4 + 2 + 4 = 21$$

***Figure 6.5*** A state space tree for an instance of the Traveling Salesperson Problem in which there are five vertices. The indices of the vertices in the partial tour are stored at each node.



## Steps for the Graph in the Figure 6.4 (2/3)

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 14 & 4 & 10 & 20 \\ 2 & 14 & 0 & 7 & 8 & 7 \\ 3 & 4 & 5 & 0 & 7 & 16 \\ 4 & 11 & 7 & 9 & 0 & 2 \\ 5 & 18 & 7 & 17 & 4 & 0 \end{array}$$

[1, 4]

$v_1$ $\quad 10$

$v_2 \quad \min(\underset{v_1}{14}, \underset{v_3}{7}, \underset{v_5}{7}) = 7$ exclude the edge to $v_4$

$v_3 \quad \min(\underset{v_1}{4}, \underset{v_2}{5}, \underset{v_5}{16}) = 4$ exclude the edge to $v_4$

$v_4 \quad \min(\underset{v_2}{7}, \underset{v_3}{9}, \underset{v_5}{2}) = 2$ exclude the edge to $v_1$

$v_5 \quad \min(\underset{v_1}{18}, \underset{v_2}{7}, \underset{v_3}{17}) = 7$ exclude the edge to $v_4$

$v_1 \to v_4 \to O \to O \to O \to v_1$

bound $= 10 + 7 + 4 + 2 + 7 = 30$

## Steps for the Graph in the Figure 6.4 (3/3)

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 14 & 4 & 10 & 20 \\ 2 & 14 & 0 & 7 & 8 & 7 \\ 3 & 4 & 5 & 0 & 7 & 16 \\ 4 & 11 & 7 & 9 & 0 & 2 \\ 5 & 18 & 7 & 17 & 4 & 0 \end{array}$$

$[1,4,5]$

$v_1 \quad 10$

$v_2 \quad \min(\underset{v_1}{14}, \underset{v_3}{7}) = 7$ exclude the edge to $v_4$ $v_5$

$v_3 \quad \min(\underset{v_1}{4}, \underset{v_2}{5}) = 4$ exclude the edge to $v_4$ $v_5$

$v_4 \quad 2$

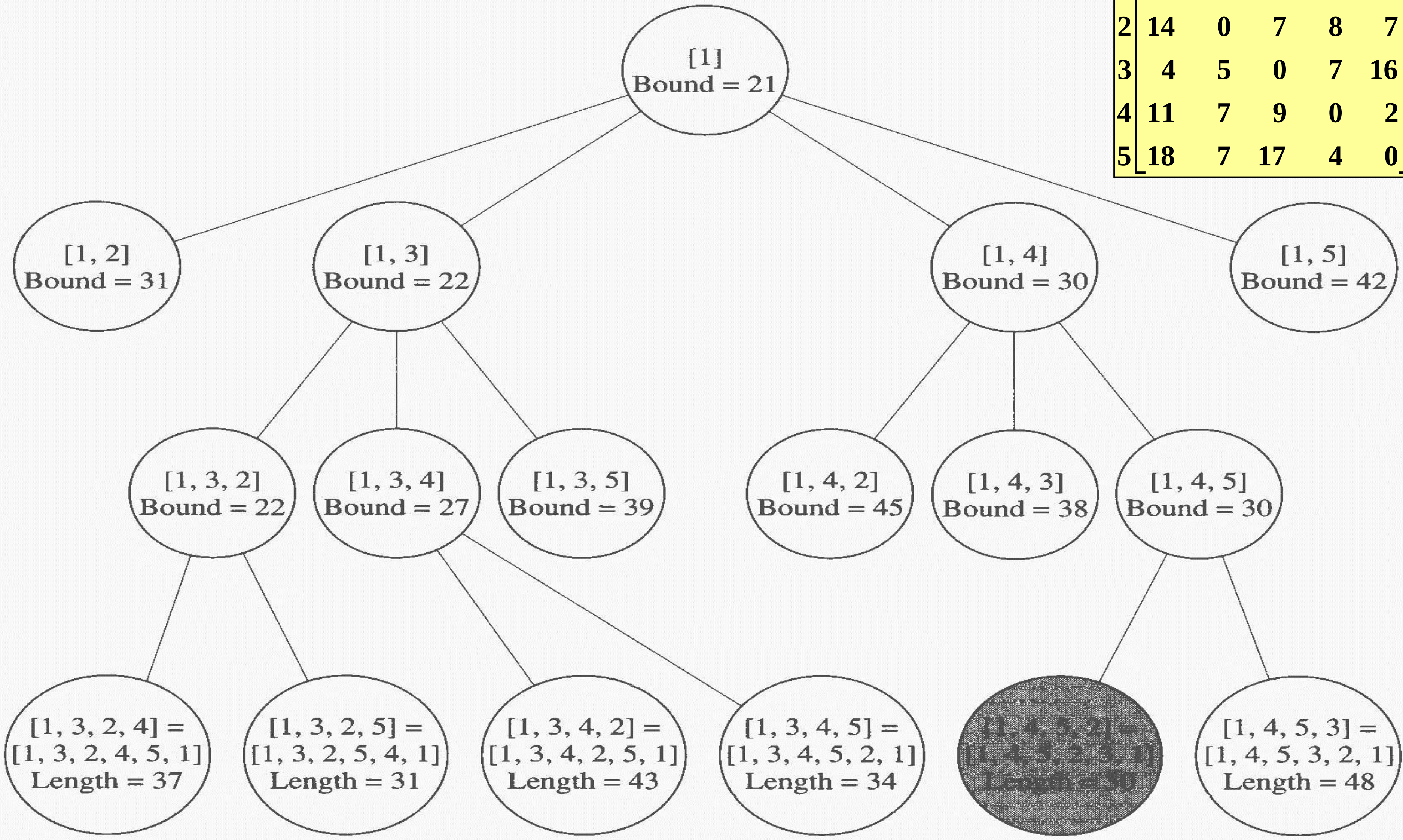
$v_5 \quad \min(\underset{v_2}{7}, \underset{v_3}{17}) = 7$ exclude the edge to $v_1$ $v_4$

$v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow \text{O} \rightarrow \text{O} \rightarrow v_1$

bound $= 10 + 7 + 4 + 2 + 7 = 30$

***Figure 6.6*** The pruned state space tree produced using best-first search with branch-and-bound pruning in Example 6.3. At each node that is not a leaf in the state space tree, the partial tour is at the top and the bound on the length of any tour that could be obtained by expanding beyond the node is at the bottom. At each leaf in the state space tree, the tour is at the top and its length is at the bottom. The shaded node is the one at which an optimal tour is found.

$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 14 & 4 & 10 & 20 \\
2 & 14 & 0 & 7 & 8 & 7 \\
3 & 4 & 5 & 0 & 7 & 16 \\
4 & 11 & 7 & 9 & 0 & 2 \\
5 & 18 & 7 & 17 & 4 & 0
\end{array}
$$


[1] Bound = 21
[1, 2] Bound = 31
[1, 3] Bound = 22
[1, 4] Bound = 30
[1, 5] Bound = 42
[1, 3, 2] Bound = 22
[1, 3, 4] Bound = 27
[1, 3, 5] Bound = 39
[1, 4, 2] Bound = 45
[1, 4, 3] Bound = 38
[1, 4, 5] Bound = 30
[1, 3, 2, 4] = [1, 3, 2, 4, 5, 1] Length = 37
[1, 3, 2, 5] = [1, 3, 2, 5, 4, 1] Length = 31
[1, 3, 4, 2] = [1, 3, 4, 2, 5, 1] Length = 43
[1, 3, 4, 5] = [1, 3, 4, 5, 2, 1] Length = 34
[1, 4, 5, 2] = [1, 4, 5, 2, 3, 1] Length = 30
[1, 4, 5, 3] = [1, 4, 5, 3, 2, 1] Length = 48

## Algorithm 6.3 The Best-FS with BB Pruning for the Traveling Salesperson Problem (1/3)

- **Problem: Determine an optimal tour in a weighted, directed graph. The weights are nonnegative numbers.**
- **Inputs: a weighted, directed graph, and *n*, the number of vertices in the graph. The graph is represented by a 2D array *W*, which has both its rows and columns indexed from 1 to *n*, where *W*[*i*][*j*] is the weight on the edge from the *i*-th vertex to the *j*-th vertex.**
- **Outputs: variable *minlength*, whose value is the length of an optimal tour, and variable *opttour*, whose value is an optimal tour.**

```
struct node{
        int level;
        ordered_set path;
        number bound }
```

## Algorithm 6.3 The Best-FS with BB Pruning for the Traveling Salesperson Problem (2/3)

```
void travel2 (int n, const number W[ ][ ],
                        ordered-set& opttour, number& minlength)
{
   priority_queue_of_node PQ;    node u, v;
   initialize(PQ);                        // Initialize PQ to be empty.
   v.level = 0;
   v.path = [1];                          // Make first vertex the starting one.
   v.bound = bound(v);
   minlength = ∞;
   insert(PQ, v);
   while (! empty(PQ)) {
      remove(PQ, v);                      // Remove node with best bound.
      if (v.bound < minlength) {
         u.level = v.level + 1;           // Set u to a child of v.
         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;
```

## Algorithm 6.3 The Best-FS with BB Pruning for the Traveling Salesperson Problem (3/3)

```
                if (u.level == n – 2) {                // Check if next vertex
                    put index of only vertex not      // completes a tour.
                    in u.path at the end of u.path;
                    put 1 at the end of u.path;       // Make first vertex the last one.
                    if (length(u) < minlength) {      // Function length computes the
                        minlength = length(u);        // length of the tour.
                        opttour = u.path;
                    }
                }
                else {
                    u.bound = bound(u);
                    if (u.bound < minlength)
                        insert(PQ, u);
                }
            }
        }
    }
}
```