Chapter 5: Backtracking

Contents

5.1 The Backtracking Technique

5.2 The n-Queens Problem

5.3 Using a Monte-Carlo Algorithm to Estimate the Efficiency of a Backtracking Algorithm

5.4 The Sum-of-Subsets Problem

5.5 Graph Coloring

5.6 The Hamiltonian Circuits Problem

5.7 The 0/1 Knapsack Problem

5.1 The Backtracking Technique

- **Backtracking is used to solve problems in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion $P(x_1, x_2, \cdots, x_n)$**

Ex)

N-Queens problem: no two queens may be in the same row, column or diagonal.

0/1 knapsack: maximize profit $\sum_{i=1}^{n} p_i x_i$

5.1 The Backtracking Technique

- **Backtracking:**
  - **depth first search (preorder) +**
  - **if dead ends, go back to the parent (pruning) and proceed again**

- **A node is called nonpromising if when visiting the node we determine that it cannot possibly lead to a solution. Otherwise we call it promising.**

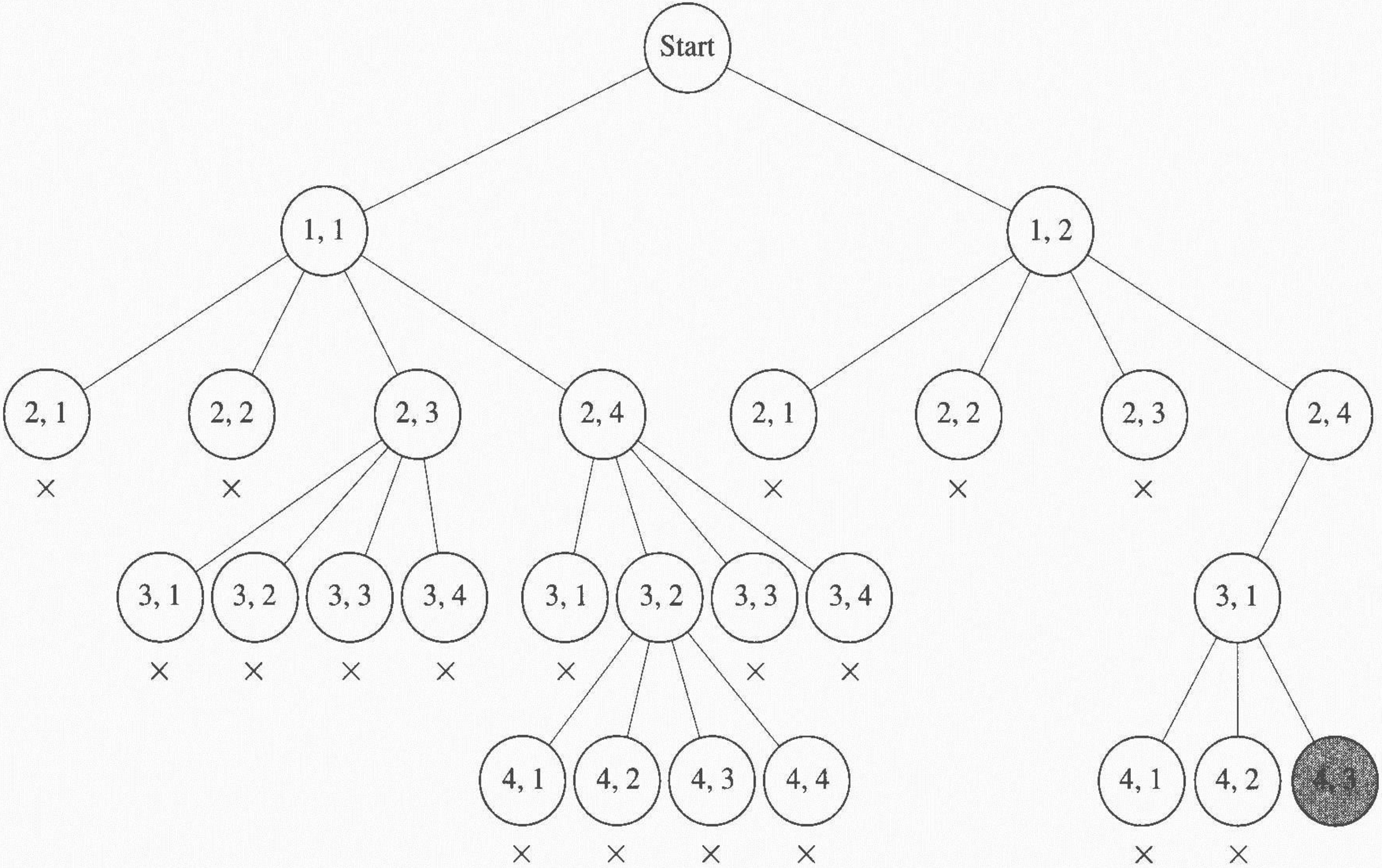
```
void depth_first_tree_search (node v)
{
    node u;
    visit v;
    for (each child u of v)
        depth_first_tree_search(u);
}
```

General Algorithm

```
void checknode (node v)
{
    node u;
    if (promising(v))    // depends on problem.
        if (there is a solution at v)
            write the solution;
        else
            for (each child u of v)
                checknode(u);
}
```

- See ex. 5.1 – Fig. 5.4 and Fig. 5.5

Figure 5.4 A portion of the pruned state space tree produced when backtracking is used to solve the instance of the *n*-Queens Problem in which $n = 4$. Only the nodes checked to find the first solution are shown. That solution is found at the shaded node. Each nonpromising node is marked with a cross.



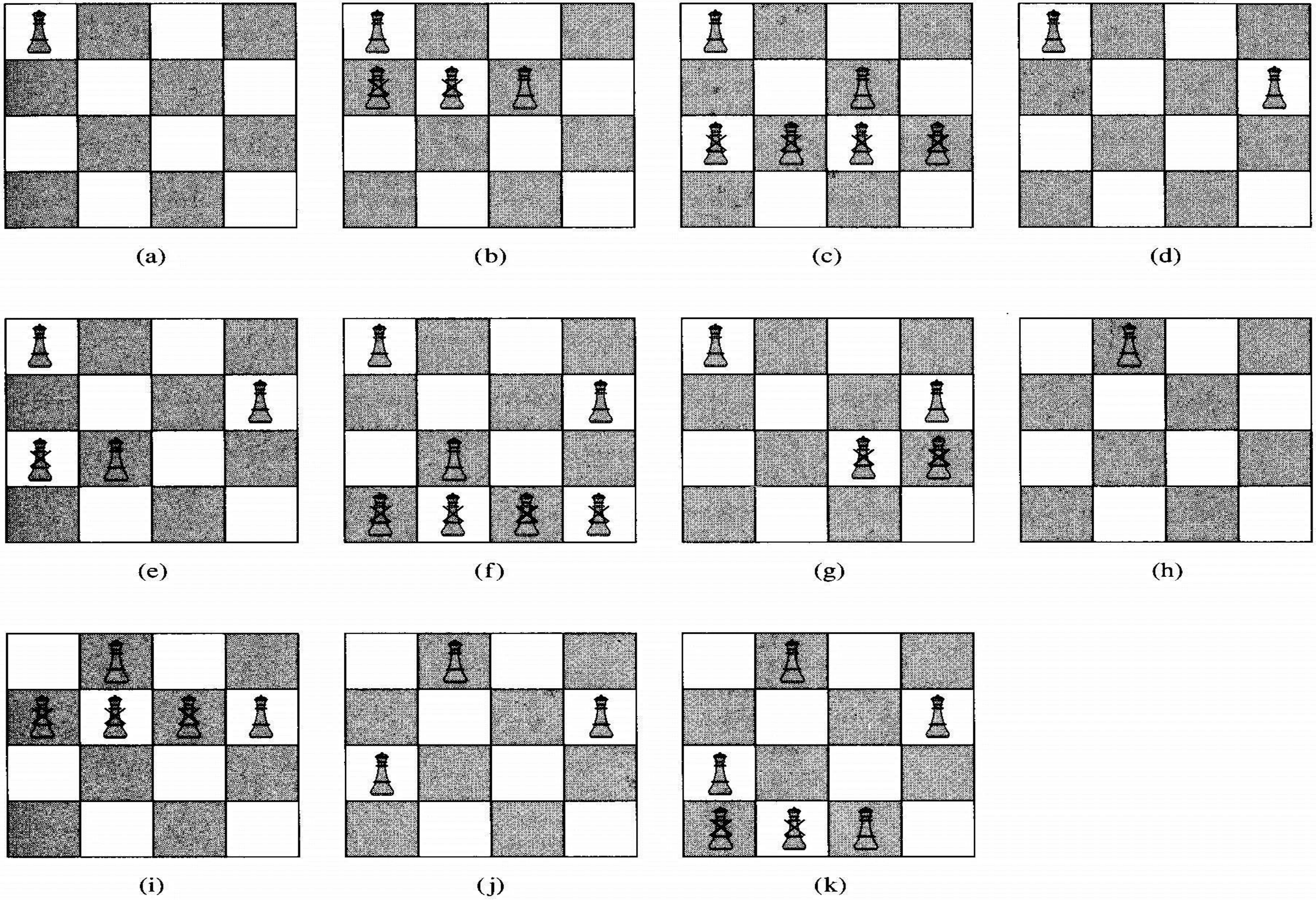


Figure 5.5 The actual chessboard positions that are tried when backtracking is used to solve the instance of the *n*-Queens Problem in which *n* = 4. Each nonpromising position is marked with a cross.

(a)

(b)

(c)

(d)

(e)

(f)

(g)

(h)

(i)

(j)

(k)

Better Algorithm

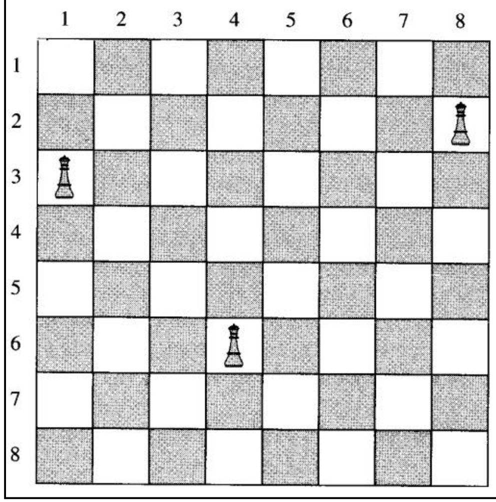
```
void expand (node v)
{
    node u;
    for (each child u of v)
        if (promising(u))
            if (there is a solution at u)
                write the solution;
            else
                expand(u);
}
```

- **Check whether a node is promising before passing it (not place the activation records for nonpromising nodes on the recursive stack)**

5.2 The n–Queens Problem

- **Position *n* queens on a chessboard so that no two are in the same row, column, or diagonal.**
- **Col(*i*) be the column where the queen in the *i*-th row is located.**
  - **Assume $i > k$**
  - **Q_i and Q_k in the same column?**
    - **col(*i*) = col(*k*)?**
  - **Q_i and Q_k in the same diagonal?**
    - **$|\,\mathrm{col}(i) - \mathrm{col}(k)\,| = i - k$**
  - **See Fig. 5.6**
- **See Alg. 5.1**
  - **Initial call: queens(0)**

Figure 5.6 The queen in row 6 is being threatened in its left diagonal by the queen in row 3 and in its right diagonal by the queen in row 2.



Algorithm 5.1 The Backtracking Algorithm for the n-Queens Problem (1/3)

- **Problem: Position *n* queens on a chessboard so that no two are in the same row, column, or diagonal.**
- **Inputs: positive integer *n*.**
- **Outputs: all possible ways *n* queens can be placed on an *n* x *n* chessboard so that no two queens threaten each other. Each output consists of an array of integers *col* indexed from 1 to *n*, where *col*[*i*] is the column where the queen in the *i*-th row is placed.**

Algorithm 5.1 The Backtracking Algorithm for the n-Queens Problem (2/3)

```
void queens (index i)
{
    index j;
    if (promising(i))
        if (i == n)
            cout << col[1] through col[n];
        else
            for (j = 1; j <= n; j++) {            // See if queen in (i + 1)-th
                col[i + 1] = j;                   // row can be positioned in
                queens(i + 1);                    // each of the n columns.
            }
}
```

Algorithm 5.1 The Backtracking Algorithm for the n-Queens Problem (3/3)

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

    k = 1;
    switch = true;                      // Check if any queen threatens
    while (k < i && switch) {           // queen in the i-th row.
        if (col[i] == col[k]  ||  abs(col[i] – col[k]) == i – k)
            switch = false;
        k++;
    }
    return switch;
}
```

- Q_i and Q_k in the same column?
  - col(i) = col(k)?
- Q_i and Q_k in the same diagonal?
  - | col(i) – col(k) | = $i – k$

Following the Algorithm 5.1 step by step

```
q(0)
  if p(0) T
    for (j=1
    c[1] = 1
    q(1)
      if p(1) T
       for (j=1            2                 3
        c[2] = 1           c[2] = 2          c[2] = 3
        q(2)               q(2)              q(2)
          if p(2) F          if p(2) F         if p(2) T
                                                 for (j=1
                                                   c[3] = 1
                                                   q(3)
                                                     if p(3) F
```

5.3 Using a Monte Carlo Algorithm to Estimate the Efficiency of a Backtracking Algorithm

- **Monte Carlo Algorithms are probabilistic algorithms.**
- **All the algorithms discussed so far are deterministic algorithms.**
- **Probabilistic Algorithm: the next instruction executed is sometimes determined at random according to some probabilistic distribution(e.g., uniform distribution).**
- **Monte Carlo Algorithms are used to estimate the efficiency of a backtracking algorithm for a particular instance.**

Monte Carlo Algorithm

- **The conditions for the algorithm**
  - ***The same promising functions must be used on all nodes at the same level in the sate space tree.***
  - ***Nodes at the same level in the state space tree must have the same number of children.***

Monte Carlo Algorithm

- **An estimate of the total number of nodes checked by the backtracking algorithm to find all solutions is given by $1 + t_0 + m_0t_1 + m_0m_1t_2 + \ldots + m_0m_1\ldots m_{i-1}\, t_i + \ldots$, where**
  - t_i *= total number of children of a node at level i*
  - m_i *= an estimate of the average number of promising children of nodes at level i*

Algorithm 5.2：Monte Carlo Estimate

- **Problem: Estimate the efficiency of a backtracking algorithm using a Monte Carlo algorithm.**
- **Inputs: an estimate of the problem that the backtracking algorithm solves.**
- **Outputs: an estimate of the number of nodes in the pruned state space produced by the algorithm, which is the number of the nodes the algorithm will check to find all solutions to the instance.**

Algorithm 5.2 : Monte Carlo Estimate

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

    v = root of state space tree;
    numnodes = 1;    m = 1;    mprod = 1;
    while (m != 0) {
        t = number of children of v;
        mprod = mprod * m;
        numnodes = numnodes + mprod * t;
        m = number of promising children of v;
        if (m != 0)
            v = randomly selected promising child of v;
    }
    return numnodes;
}
```

Algorithm 5.3：Monte Carlo Estimate for Algrithm 5.1(The Backtracking Alg. for n−queens Problem)(1/3)

- **Problem: Estimate the efficiency of Algorithm 5.1.**
- **Inputs: positive integer *n*.**
- **Outputs: an estimate of the number of nodes in the pruned state space produced by Algorithm 5.1, which is the number of the nodes the algorithm will check before finding all ways to position n queens on an n x n chessboard so that no two queens threaten each other.**
- **This is a specific version of Monte Carlo Estimate for the Backtracking Algorithm for n-Queens problem.**

Algorithm 5.3 : Monte Carlo Estimate for Algrithm 5.1(The Backtracking Alg. for n-queens Problem)(2/3)

- **When a Monte Carlo algorithm is used, the estimate should be run more than once, and the average of the results should be used as the actual estimate.**
- **A rule of thumb: around 20 trials are ordinarily sufficient.**

```
// A specific version of Alg. 5.2 for Alg. 5.1.

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;
```

Algorithm 5.3：Monte Carlo Estimate for Algrithm 5.1(The Backtracking Alg. for n−queens Problem)(3/3)

```
    while (m != 0 && i != n) {
        mprod = mprod * m;
        numnodes = numnodes + mprod * n;      /* # children t is n */
        i++;
        m = 0;
        prom_children = Ø ;  /* initialize set of promising children to
        empty*/
        for (j = 1; j <= n; j++) {
            col[i] = j;
            if (promising(i))
               { m++;  prom_children = prom_children ∪ {j}; }
        }
        if (m != 0)
          { j = random selection from prom_children; col[i] = j; }
    }              /* end of while */
    return numnodes;
}                  /* end of estimate_n_queens */
```

5.4 The Sum-of Subsets Problem

- **Given** $w_i > 0,\ 1 \le i \le n,$ and $W > 0$

  Find all subsets that sum to *W*

  - **Ex. 5.2** $w_1 = 5,\ w_2 = 6,\ w_3 = 10,\ w_4 = 11,\ w_5 = 16,\quad W = 21$

$$\{w_1,\ w_2,\ w_3\},\ \{w_1,\ w_5\},\ \{w_3,\ w_4\}$$

- **Pruning**

  sort w_i's **in nondecreasing order.**

  $w_1 \le w_2 \le \cdots \le w_n$

  Let *weight* be the sum of weights that have been included up to node at level i

  Let $total = w_{i+1} + \cdots + w_n$

  Node at level i is nonpromising if

  1) $weight + w_{i+1} > W$

  2) $weight + total < W$

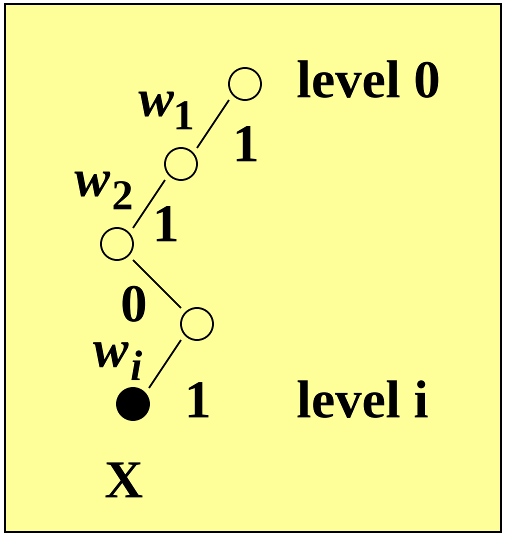
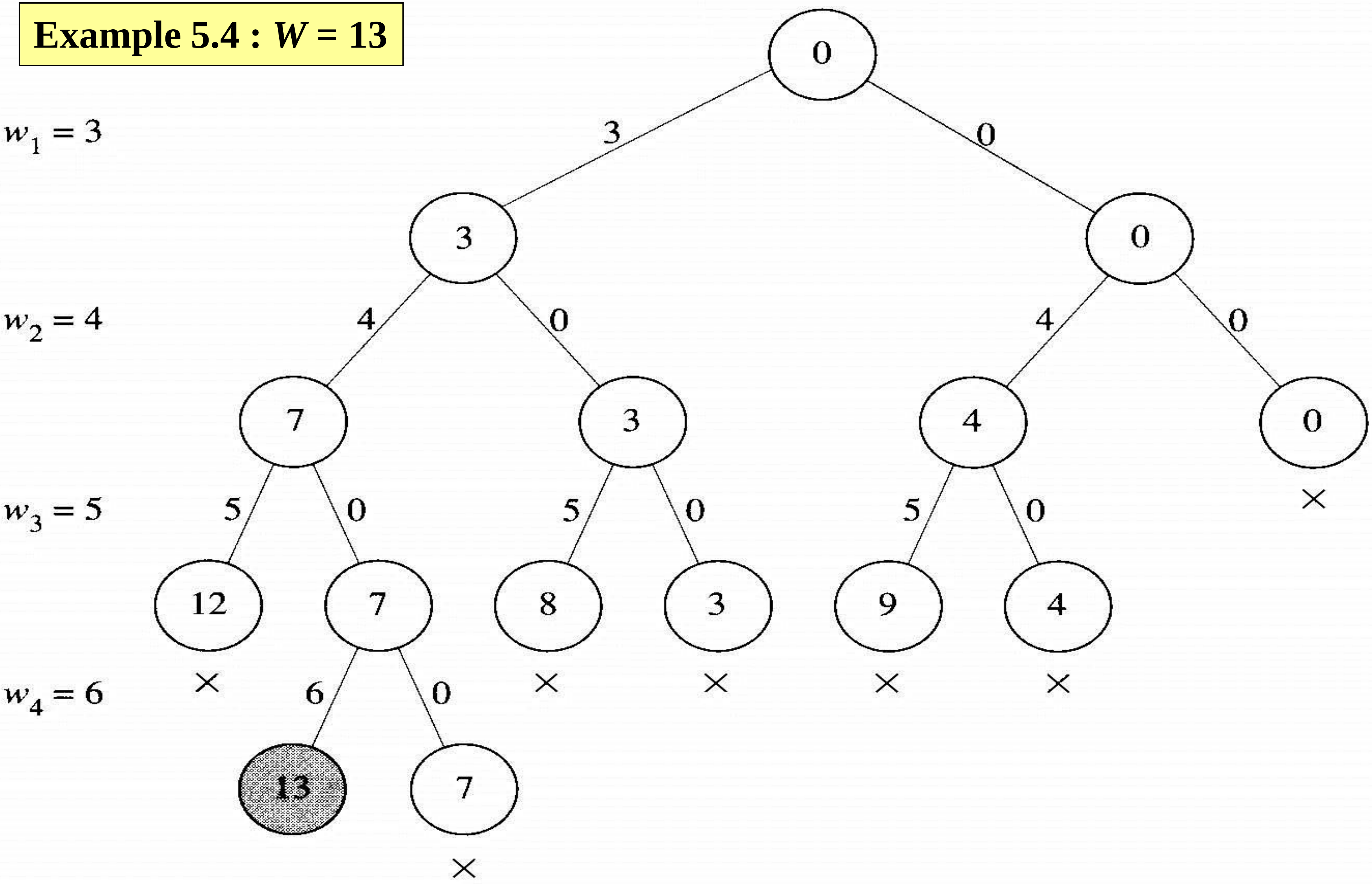


Figure 5.9 The pruned state space tree produced using backtracking in Example 5.4. Stored at each node is the total weight included up to that node. The only solution is found at the shaded node. Each nonpromising node is marked with a cross.

Example 5.4 : *W* = 13



Algorithm

- **See Alg. 5.4**
- **sum_of_subsets(i, weight, total)**
- **Initial call (0, 0, $\sum_{j=1}^{n} w_j$)**

Algorithm 5.4 The Backtracking Algorithm for the Sum-of-Subsets Problem (1/2)

- **Problem: Given *n* positive integers (weights) and a positive integer *W*, determine all combinations of the integers that sum to *W*.**
- **Inputs: positive integer *n*, sorted (nondecreasing order) array of positive integers *w* indexed from 1 to *n*, and a positive integer *W*.**
- **Outputs: all combinations of the integers that sum to *W*.**

Algorithm 5.4 The Backtracking Algorithm for the Sum-of-Subsets Problem (2/2)

```
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”;                // Include w[i + 1].
            sum_of_subsets(i + 1, weight + w[i + 1], total – w[i + 1]);
            include[i + 1] = “no”;                 // Do not include 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);
}
```

Node at level *i* is nonpromising if

1) $weight + w_{i+1} > W$;   2) $weight + total < W$

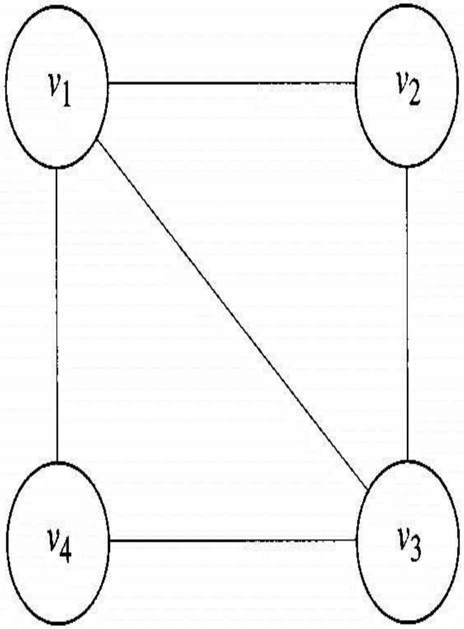
Following the Algorithm 5.4 for Ex. 5.4

Ex. 5.4 :
$W = 13$
$w_1 = 3$
$w_2 = 4$
$w_3 = 5$
$w_4 = 6$

```
ss(0, 0, 18)
  if p(0) T
    i[1] = y
    ss(1, 3, 15)
      if p(1) T
        i[2] = y
        ss(2, 7, 11)
          if p(2) T
            i[3] = y
            ss(3, 12, 6)
              if p(3) F
            i[3] = n
            ss(3, 7, 6)
              if p(3) T
                i[4] = y
                ss(4, 13, 0)
                  if p(4) T
                    if (weight=W) T
                i[4] = n
                ss(4, 7, 0)
                  if p(4) F
        i[2] = n
        ss(2, 3, 11)
                    ……
```

5.5 Graph Coloring

- **Color an undirected graph using at most *m* different colors so that no two adjacent vertices are the same color.**



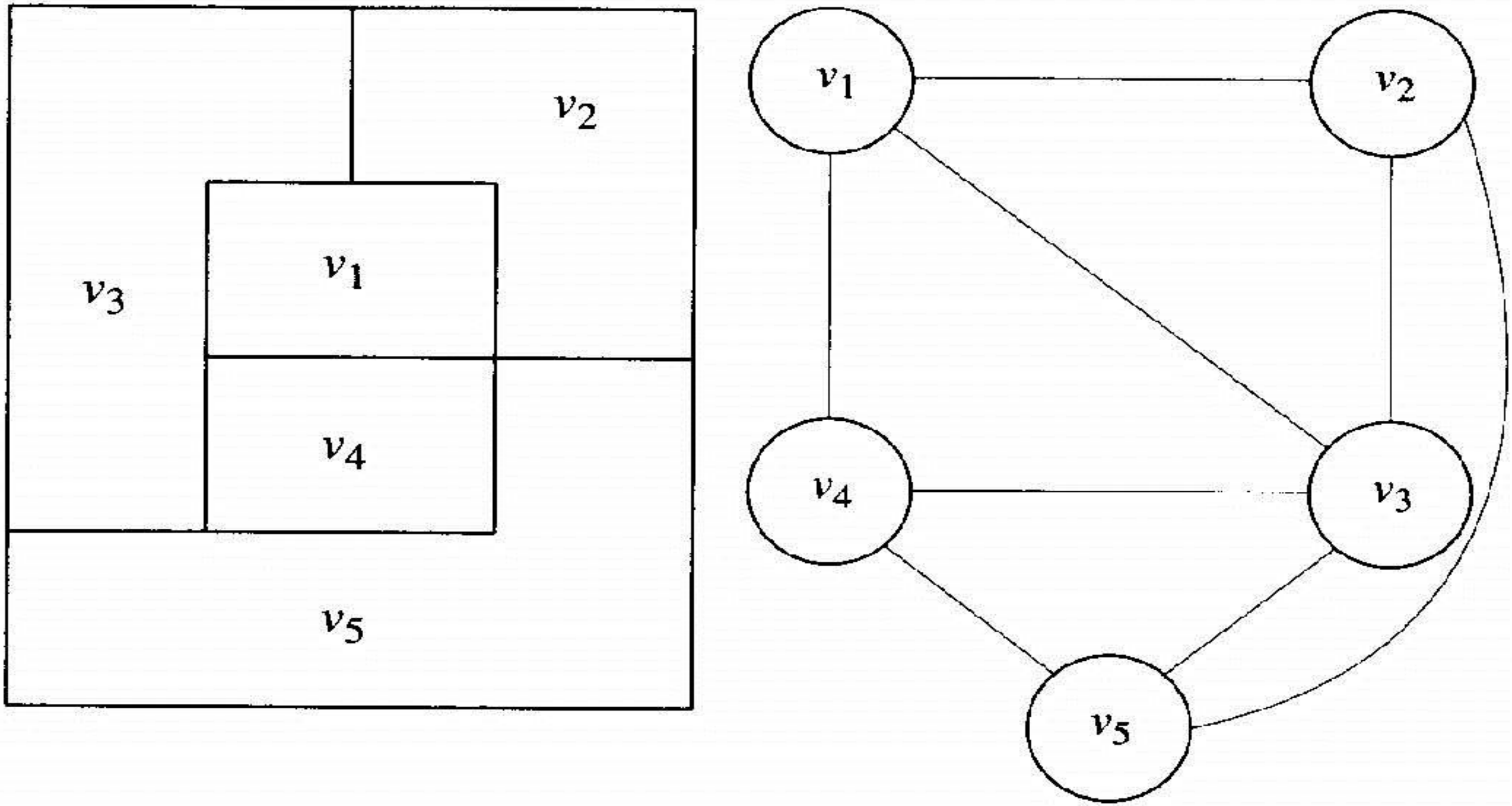
Vertex	***Color***
v_1	**color 1**
v_2	**color 2**
v_3	**color 3**
v_4	**color 2**

Six(3!) solutions

Figure 5.10 Graph for which there is no solution to the 2-Coloring Problem. A solution to the 3-Coloring Problem for this graph is shown in Example 5.5.

Application: coloring maps

Figure 5.11 Map (top) and its planar graph representation (bottom).



Algorithm 5.5 The Backtracking Algorithm for the m–coloring Problem (1/3)

- **Problem: Determine all ways in which the vertices in an undirected graph can be colored, using only *m* colors, so that adjacent vertices are not the same color.**
- **Inputs: positive integers *n* and *m*, and an undirected graph containing *n* vertices. 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 true if there is an edge between *i*-th vertex and the *j*-th vertex and false otherwise.**
- **Outputs: all possible colorings of the graph, using at most *m* colors, so that no two adjacent vertices are the same color. The output for each coloring is an array *vcolor* indexed from 1 to *n*, where *vcolor*[*i*] is the color (an integer between 1 and *m*) assigned to the *i*-th vertex.**

Algorithm 5.5 The Backtracking Algorithm for the m−coloring Problem (2/3)

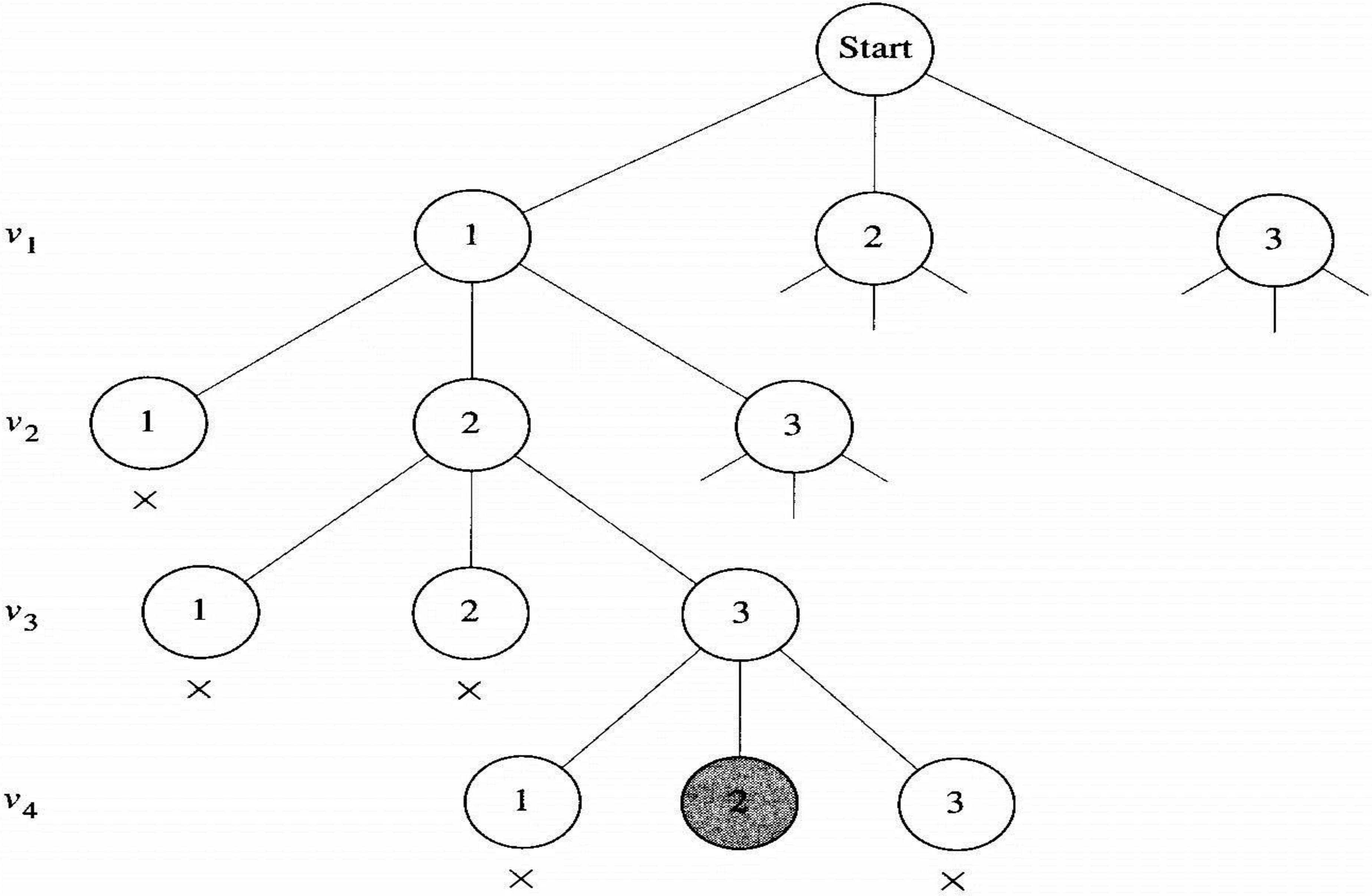
```
void m_coloring (index i )
{
    index color;
    if (promising(i))
        if (i == n)
            cout << vcolor[1] through vcolor[n];
        else
            for (color = 1; color <= m; color++) {   // Try every color for
                vcolor[i + 1] = color;                // next vertex.
                m_coloring(i + 1);
            }
}
```

Algorithm 5.5 The Backtracking Algorithm for the m–coloring Problem (3/3)

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

    switch = true;
    j = 1;
    while (j < i && switch) {                          // Check if an adjacent
        if (W[i][j] && vcolor[i] == vcolor[j])         // vertex is already this
            switch = false;                            // color.
        j++;
    }
    return switch;
}
```

Figure 5.12 A portion of the pruned state space tree produced using backtracking to do a 3-coloring of the graph in Figure 5.10. The first solution is found at the shaded node. Each nonpromising node is marked with a cross.

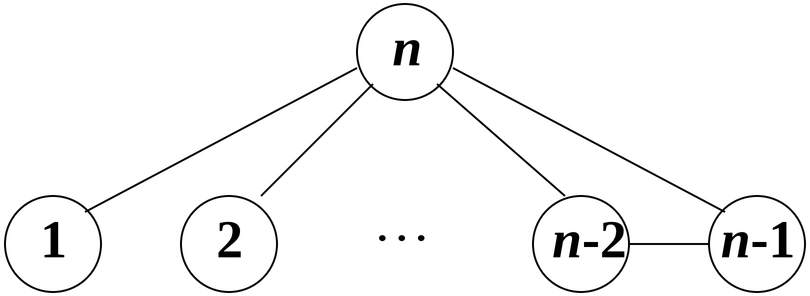


Analysis of Algorithm 5.5

- **Complexity**

$$1+m+m^2+\cdots+m^n=\frac{m^{n+1}-1}{m-1}\in\Theta(m^n)$$

- $m = 2$



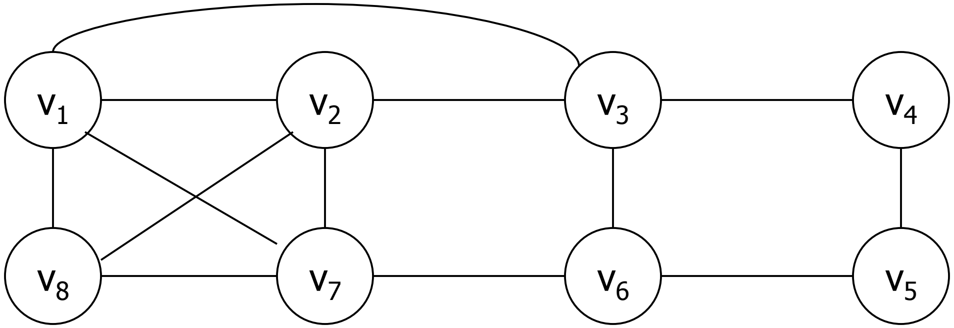
no solution

$\Theta(2^n)$

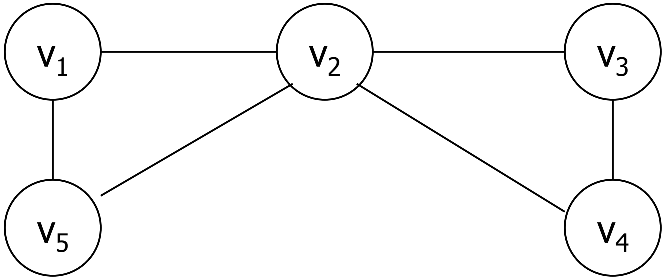
5.6 The Hamiltonian Circuits Problem

- **Hamiltonian Circuit(also called a tour) : is a path that starts at a given vertex(say v1), visits each vertex in the graph exactly once, and ends at the starting vertex.**
- **Traveling Salesman Problem : finds the lowest cost Hamiltonian circuit.**
  - **$T(n) = (n-1)(n-2)\ 2^{n-3}$,**
  - **For the basic operation time = 1 micro second,**
    - **If n = 20, T(n) = 45 seconds**
    - **However, if n = 40, T(n) = 6.46 years**
- **Hamiltonian Circuits Problem : determines the Hamiltonian Circuits in a connected, undirected graph.**

Figure 5.13 Example graphs



The graph contains the Hamiltonian Circuit
$[v_1,v_2,v_8,v_7,v_6,v_5,v_4,v_3,v_1]$



The graph contains no Hamiltonian Circuit
Fig. 5.13

State Space Tree for Hamiltonian Circuits

A state space tree for this problem

Put the starting vertex at level 0 in the tree; call it the 0-th vertex.

```
For i = 1 to n-1,
    At level i, consider each vertex other than 0-th vertex
```

Considerations

(1) i-th vertex on the path must be adjacent to (i-1)st vertex.

(2) The (n-1)st vertex must be adjacent to the 0-th vertex.

(3) The i-th vertex cannot be one of the first i – 1 vertices.

Algorithm 5.6 The Backtracking Algorithm for the Hamiltonian Circuits Problem (1/4)

- **Problem: Determine all Hamiltonian Circuits in a connected, undirected graph.**
- **Inputs: positive integer *n* and an undirected graph containing *n* vertices. 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 true if there is an edge between the *i*-th vertex and the *j*-th vertex and false otherwise.**
- **Outputs: For all paths that start at a given vertex, visit each vertex in the graph exactly once, and end up at the starting vertex. The output for each path is an array of indices *vindex* indexed from *0* to *n-1*, where *vindex*[*i*] is the index of the *i*th vertex on the path. The index of the starting vertex is *vindex[0]*.**

Algorithm 5.6 The Backtracking Algorithm for the Hamiltonian Circuits Problem (2/4)

```
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++) {          // Try all vertices as
                vindex[i + 1] = j;              // next one.
                hamiltonian(i + 1);
            }
}
```

Algorithm 5.6 The Backtracking Algorithm for the Hamiltonian Circuits Problem (3/4)

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

    if (i == n-1 && ! W[vindex[n-1]][vindex[0]]) switch = false;
    else if (i > 0 && ! W[vindex[i-1]][vindex[i]]) switch = false;
    else {
        switch = true;
        j = 1;
        while (j < i && switch) {                  // Check if vertex is
            if (vindex[i] == vindex[j])             // already selected
                switch = false;
            j++;
        }
    }
    return switch;
}
```

Algorithm 5.6 The Backtracking Algorithm for the Hamiltonian Circuits Problem (4/4)

The top level called to hamiltonian

- vindex[0] = 1; // make v_1 the starting vertex
- hamiltonian(0);

The number of nodes in the state space tree : much worse than exponential

- $1 + (n-1) + (n-1)^2 + \ldots + (n-1)^{n-1} = [(n-1)^n - 1] /(n-2)$

The possibility exists that the backtracking algorithm (for Hamiltonian Circuits Problem) will take even longer than the dynamic programming algorithm (for TSP)

However, the Monte Carlo technique estimates the time to find all circuits.

When you need one tour, stop when the 1st one is found.

5.7 The 0/1 Knapsack Problem

- **Backtracking algorithm for optimization problems**

```
void checknode(node v)
{
    node u
    if (value(v) is better than best)
                    // value(v): value of the solution at v
        best = value(v);
                    // best: best solution so far.
    if (promising(v))
        for (each child u of v)
            checknode(u);
}
```

Pruning

- ***weight***: sum of weights of the items that have been included up to the node at level ***i***.
- ***profit***: sum of profits corresponding to that node
- assume $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$

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

$$\underset{(upper)}{bound} = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j \right)}_{\substack{\text{Profit from first} \\ k\text{-1 items taken}}} + \underbrace{(W - totweight)}_{\substack{\text{Capacity available} \\ \text{for } k\text{-th item}}} \times \underbrace{\frac{p_k}{w_k}}_{\substack{\text{Profit per unit} \\ \text{weight for } k\text{-th item}}}$$

- ***maxprofit***: value of the profit in the best solution found so far
- The node at level i is nonpromising if

1) bound $\leq$ maxprofit

2) weight $\geq W$ /* can't expand its children

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

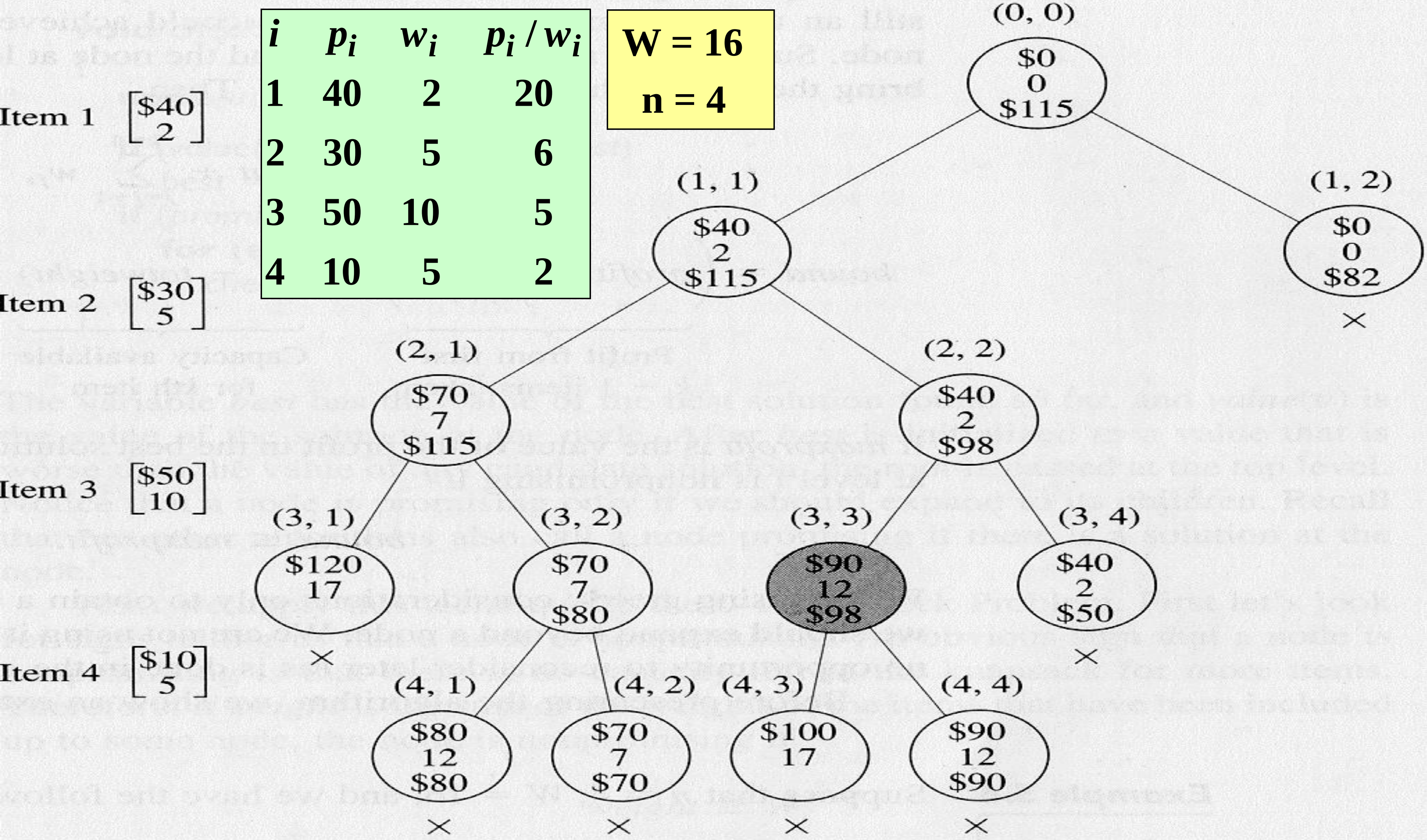


Figure 5.14 The pruned state space tree produced using backtracking in Example 5.6. Stored at each node from top to bottom are the total profit of the items stolen up to the node, their total weight, and the bound on the total profit that could be obtained by expanding beyond the node. The optimal solution is found at the shaded node. Each nonpromising node is marked with a cross.

Example 5.6 (1/4)

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

1. mp = 0
2. node (0, 0)
   - p = 0   w = 0
   - tw = w + 2 + 5 = 7
   - b = p + 40 + 30 + (16 – 7) 5 = 115 > mp
3. node (1, 1)
   - p = 40   w = 2   mp = 40
   - tw = w + 5 = 7
   - b = p + 30 + (16 – 7) 5 = 115 > mp
4. node (2, 1)
   - p = 40 + 30 = 70   w = 2 + 5 = 7   mp = 70
   - tw = w + 0 = 7
   - b = p + (16 – 7) 5 = 115 > mp
5. node (3, 1)
   - p = 70 + 50 = 120   w = 7 + 10 = 17 > W

Example 5.6 (2/4)

$W = 16$

$n = 4$

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

6. backtrack to (2, 1)

7. node (3, 2)

   p = 70 w = 7 mp = 70

   tw = w + 5 = 12 /* not necessary */

   b = p + 10 = 80 > mp

8. node (4, 1)

   p = 80 w = 12 mp = 80

   tw = w + 0 = 12

   b = p + 0 = 80 <= mp

9. backtrack to (3, 2)

10. node (4, 2)

   p = 70 w = 7 mp = 80

   tw = w + 0 = 7

   b = p + 0 = 70 < mp

Example 5.6 (3/4)

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

11. backtrack to (1, 1)
12. node (2, 2)
    - p = 40 w = 2 mp = 80
    - tw = w + 10 = 12
    - b = p + 50 + (16 – 12) 2 = 98 > mp
13. node (3, 3)
    - p = 90 w = 12 mp = 90
    - tw = w + 0 = 12
    - b = 90 + (16 – 12) 2 = 98 > mp
14. node (4, 3)
    - p = 100 w = 17 > *W*
15. backtrack to (3, 3)
16. node (4, 4)
    - p = 90 w = 12 mp = 90
    - tw = w + 0 = 12
    - b = p + 0 = 90 <= mp

Example 5.6 (4/4)

W = 16	i	p_i	w_i	p_i / w_i
n = 4	1	40	2	20
	2	30	5	6
	3	50	10	5
	4	10	5	2

17. backtrack to (2, 2)

18. node (3, 4)

    $p = 40 \quad w = 2 \quad mp = 90$

    $tw = w + 5 = 7$

    $b = p + 10 = 50 < mp$

19. backtrack to (0, 0)

20. node (1, 2)

    $p = 0 \quad w = 0 \quad mp = 90$

    $tw = w + 5 + 10 = 15$

    $b = p + 30 + 50 + (16-15)\, 2 = 82 < mp$

- Entire state space tree: 31 nodes
- Pruned state space tree: 13 nodes
- Horowitz & Sahni: BT is usually more efficient than DP

Algorithm 5.7 The Backtracking 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 *w* and *p*, each indexed from 1 to *n*, and each containing positive integers sorted in nonincreasing order according to the values of *p*[*i*]/*w*[*i*].**
- **Outputs: An array *bestset* indexed from 1 to *n*, where the values of *bestset*[*i*] is “yes” if the *i*-th item is included in the optimal set and is “no” otherwise; an integer *maxprofit* that is the maximum profit.**

Algorithm 5.7 The Backtracking Algorithm for the 0−1 Knapsack Problem (2/3)

```
void knapsack (index i ,
                         int profit, int weight)
{
   if (weight <= W && profit > maxprofit) {      // This set is best so far.
      maxprofit = profit;
      numbest = i;                               // Set numbers to number
      bestset = include;                         // of items considered. Set
   }                                             // bestset to this solution.
   if (promising(i)) {
      include[i + 1] = “yes”;                    // Include w[i + 1].
      knapsack(i + 1, profit + p[i + 1], weight + w[i + 1]);
      include[i + 1] = “no”;                     // Do not include w[i + 1].
      knapsack(i + 1, profit, weight);
   }
}
```

Algorithm 5.7 The Backtracking Algorithm for the 0−1 Knapsack Problem (3/3)

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

```
bool promising (index i)
{
   index j, k;    int totweight;    float bound;
   if (weight >= W)        // Node is promising only if we should expand to its
      return false;         // children. There must be some capacity left for the children.
   else {
      j = i + 1;       bound = profit;       totweight = weight;
      while (j <= n && totweight + w[j] <= W) {
                            // Grab as many items as possible.
         totweight = totweight + w[j];    bound = bound + p[j];    j++;
      }
      k = j;                // Use k for consistency with formula in text.
      if (k <= n)
         bound = bound + (W – totweight) * p[k]/w[k]; // Grab fraction of
      return bound > maxprofit;                         // k-th item.
   }
}
```

<< non-promising >>

1) $\text{bound} \leq \text{maxprofit}$

2) $\text{weight} \geq W$

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