S. J. P. N. TRUST'S
HIRASUGAR INSTITUTE OF TECHNOLOGY, NIDASOSHI
Accredited at 'A' Grade by NAAC
Programmes Accredited by NBA: CSE, ECE, EEE & ME.

Department of Computer Science & Engineering

Course: Design And Analysis of Algorithms (18CS42)

Module 5: Backtracking, Program & Bound, 0/1 Knapsack Problem, NP-Hard & NP-Complete Problems

Prof. A. A. Daptardar
Asst. Prof. , Dept. of Computer Science & Engg.,
Hirasugar Institute of Technology, Nidasoshi

Module – 5
Backtracking

1. **General Method (T2:7.1)**
2. **n-Queens Problem (T1:12.1)**
3. **Sum of Subset Problem (T1:12.1)**
4. **Graph coloring (T2:7.4)**
5. **Hamiltonian Cycle (T2:7.5)**

Backtracking General Method

- The desired solution is expressible as an n-tuple $(x_1,\ldots..,x_n)$, where the x_i are chosen from some finite set Si
- Often the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a criterion function $P(x_1,\ldots..,x_n)$
- Sometimes it seeks all vectors that satisfy P
- Suppose m is the size of set S_i, then there are $m = m_1 m_2 \ldots. m_n$, n-tuples that are possible candidates for satisfying the function P
- Its basic idea is to build up the solution vector one component at a time and to use modified criterion function $P_i(x_1, x_2, \ldots, x_i)$ (sometimes called bounding function) to test whether the vector being formed has any chance to of success
- Major advantages of this method is this – if it is realized that the partial vector $(x_1, x_2, \ldots\ldots., x_i)$ can in no way lead to an optimal solution, then $m_{i+1} \ldots. m_n$ possible test vectors can be ignored entirely

Backtracking General Method

- The principle idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows
- If a partially constructed solution can be developed further without violating the problem’s constraints, it is done by taking the first remaining legitimate option for the next component
- If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered

Backtracking General Method

- In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option
- This kind of backtracking is implemented by constructing a tree called as state-space tree(SST)
- In SST, its root represent an initial state before the search for a solution begins
- The nodes of first level in the SST represent the choices made for the first component of a solution
- The nodes of the second level represent the choices for the second component and so on

```
Algorithm Backtrack($k$)
// This schema describes the backtracking process using
// recursion. On entering, the first $k-1$ values
// $x[1], x[2], \ldots, x[k-1]$ of the solution vector
// $x[1:n]$ have been assigned. $x[\,]$ and $n$ are global.
{
    for (each $x[k] \in T(x[1], \ldots, x[k-1])$ do
    {
        if ($B_k(x[1], x[2], \ldots, x[k]) \neq 0$) then
        {
            if ($x[1], x[2], \ldots, x[k]$ is a path to an answer node)
                then write ($x[1:k]$);
            if ($k < n$) then Backtrack($k+1$);
        }
    }
}
```

Algorithm 7.1 Recursive backtracking algorithm

n-Queens Problem

n-Queens Problem

- The problem is to place n queens on an n-by-n chessboard so that no TWO queens attack each other by being in the same row or in the same column or on the same diagonal.
- For n=1, the problem has a trivial solution
- For n=2 and n=3, there is no solution
- So let us consider, n=4, i.e. four-queens problem and solve it by the backtracking technique

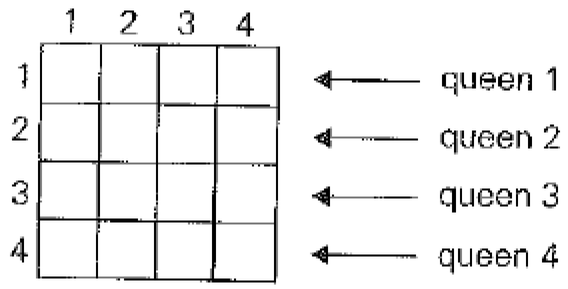


FIGURE 12.1 Board for the four-queens problem

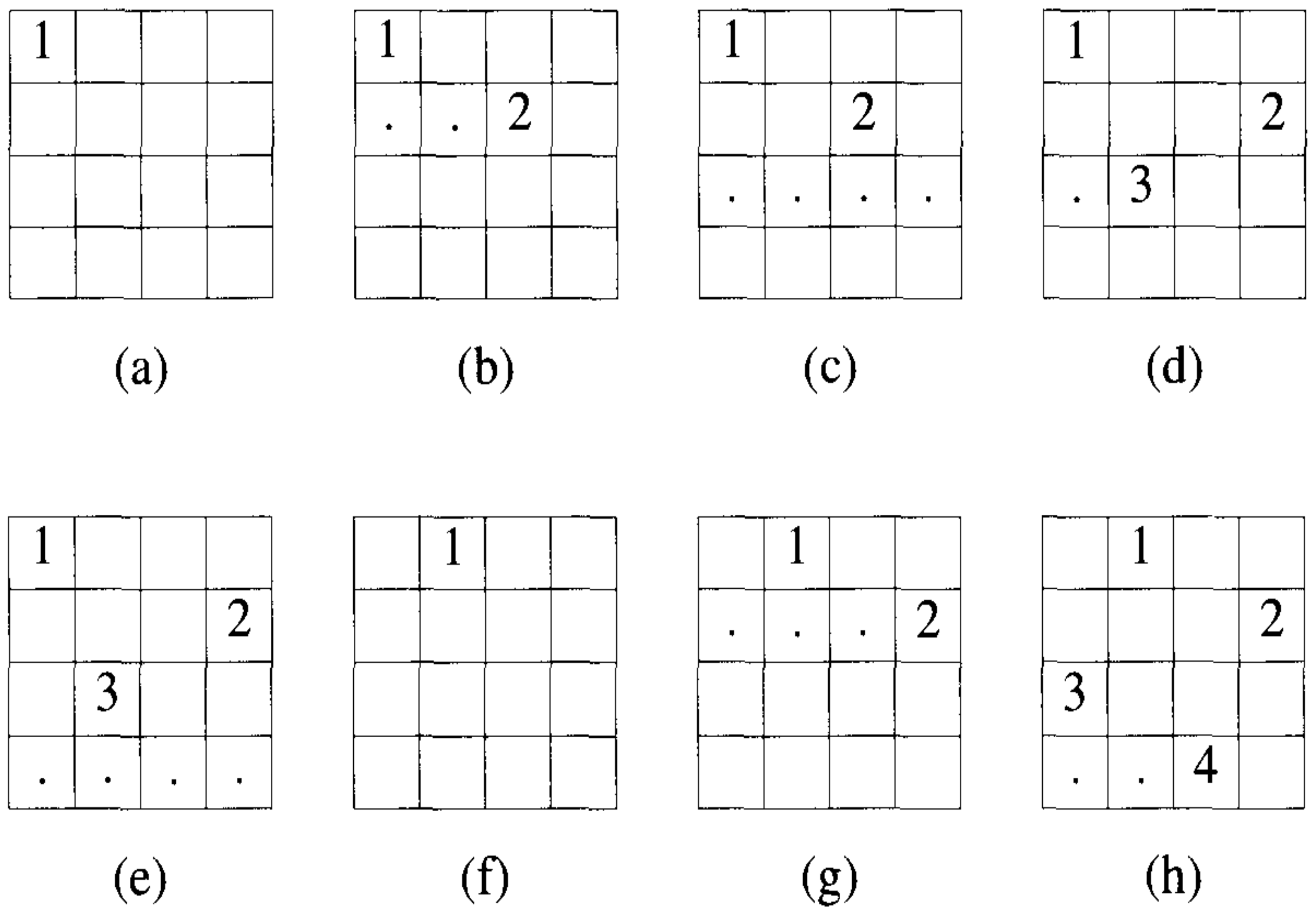


Figure 7.5 Example of a backtrack solution to the 4-queens problem

State-Space Tree for n-Queens Problem

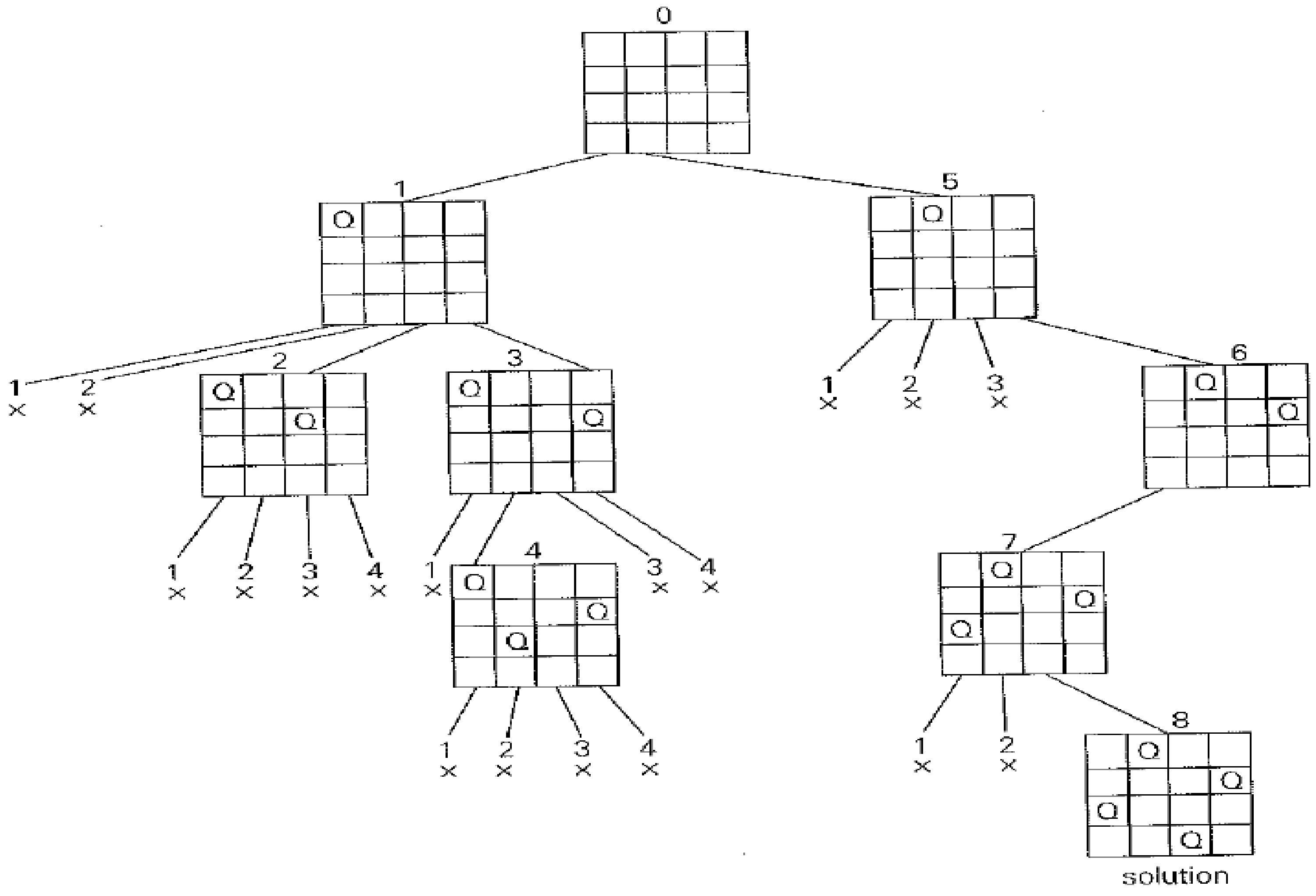


FIGURE 12.2 State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

Figure 7.5 shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen which were tried and rejected because another queen was attacking. In Figure 7.5(b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In Figure 7.5(c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In Figure 7.5(d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure 7.5 (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.

N-Queen Algorithm

```
Algorithm Place($k, i$)
// Returns true if a queen can be placed in $k$th row and
// $i$th column. Otherwise it returns false. $x[\,]$ is a
// global array whose first $(k-1)$ values have been set.
// Abs($r$) returns the absolute value of $r$.
{
    for $j := 1$ to $k-1$ do
        if (($x[j] = i$) // Two in the same column
            or (Abs($x[j] - i$) = Abs($j - k$)))
                // or in the same diagonal
            then return false;
    return true;
}
```

```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of $n$ queens on an $n \times n$
// chessboard so that they are nonattacking.
{
    for $i := 1$ to $n$ do
    {
        if Place(k, i) then
        {
            $x[k] := i$;
            if $(k = n)$ then write $(x[1 : n])$;
            else NQueens($k + 1, n$);
        }
    }
}
```

Sum of subset Problem

Example 7.2 [Sum of subsets] Given positive numbers w_i, $1 \leq i \leq n$, and m, this problem calls for finding all subsets of the w_i whose sums are m. For example, if $n = 4$, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$, and $m = 31$, then the desired subsets are (11, 13, 7) and (24, 7). Rather than represent the solution vector by the w_i which sum to m, we could represent the solution vector by giving the indices of these w_i. Now the two solutions are described by the vectors (1, 2, 4) and (3, 4). In general, all solutions are k-tuples $(x_1, x_2, \ldots, x_k)$, $1 \leq k \leq n$, and different solutions may have different-sized tuples. The explicit constraints require $x_i \in \{j \mid j \text{ is an integer and } 1 \leq j \leq n\}$. The implicit constraints require that no two be the same and that the sum of the corresponding w_i's be m. Since we wish to avoid generating multiple instances of the same subset (e.g., (1, 2, 4) and (1, 4, 2) represent the same subset), another implicit constraint that is imposed is that $x_i < x_{i+1}$, $1 \leq i < k$.

In another formulation of the sum of subsets problem, each solution subset is represented by an n-tuple $(x_1, x_2, \ldots, x_n)$ such that $x_i \in \{0, 1\}$, $1 \leq i \leq n$. Then $x_i = 0$ if w_i is not chosen and $x_i = 1$ if w_i is chosen. The solutions to the above instance are (1, 1, 0, 1) and (0, 0, 1, 1). This formulation expresses all solutions using a fixed-sized tuple. Thus we conclude that there may be several ways to formulate a problem so that all solutions are tuples that satisfy some constraints. One can verify that for both of the above formulations, the solution space consists of 2^n distinct tuples. $\square$

Example 7.4 [Sum of subsets] In Example 7.2 we gave two possible formulations of the solution space for the sum of subsets problem. Figures 7.3 and 7.4 show a possible tree organization for each of these formulations for the case $n = 4$. The tree of Figure 7.3 corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level $i+1$ node represents a value for x_i. At each node, the solution space is partitioned into subsolution spaces. The solution space is defined by all paths from the root node to any node in the tree, since any such path corresponds to a subset satisfying the explicit constraints. The possible paths are () (this corresponds to the empty path from the root to itself), (1), (1,2), (1,2,3), (1,2,3,4), (1,2,4), (1,3,4), (2), (2,3), and so on. Thus, the leftmost subtree defines all subsets containing w_1, the next subtree defines all subsets containing w_2 but not w_1, and so on.

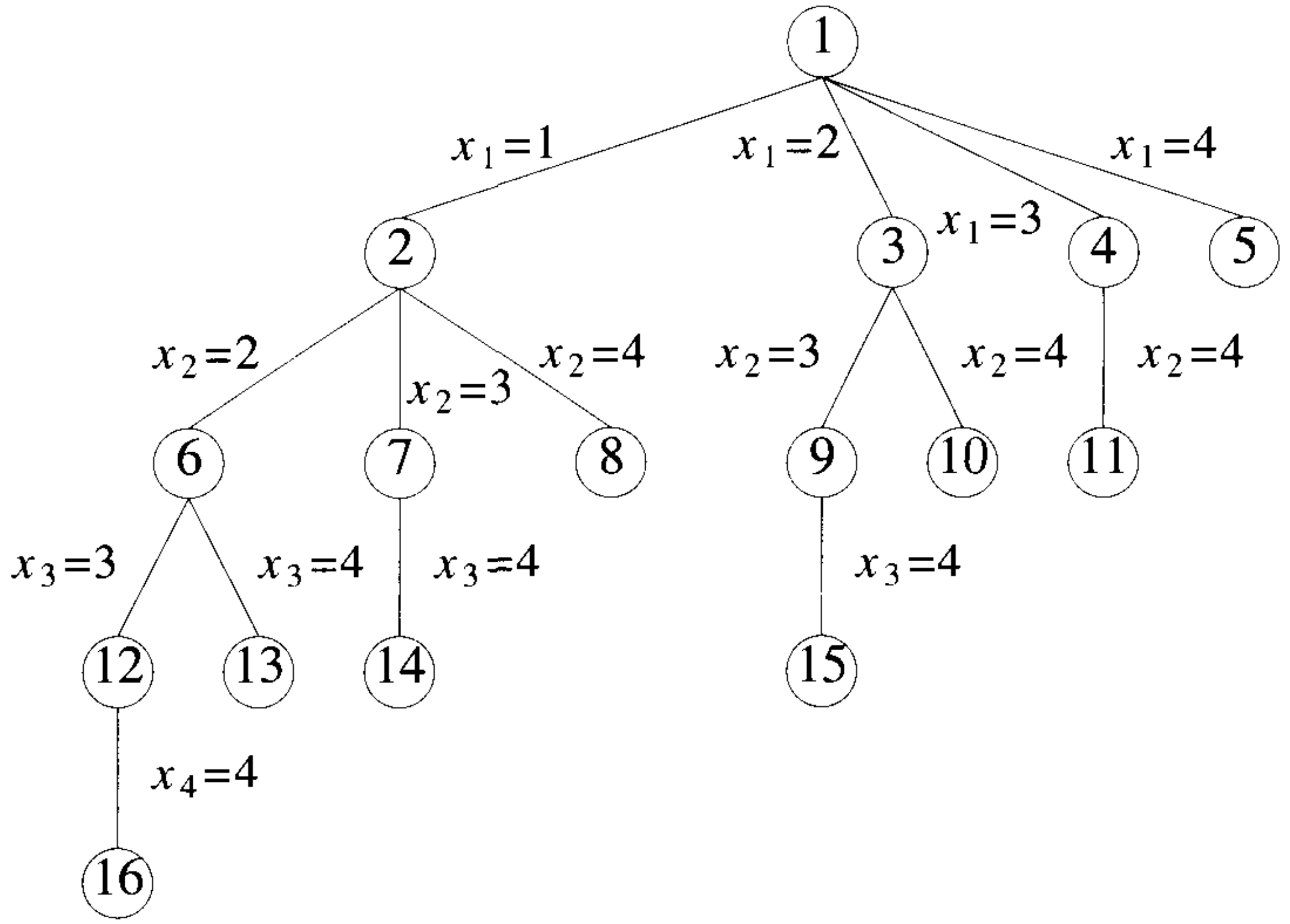


Figure 7.3 A possible solution space organization for the sum of subsets problem. Nodes are numbered as in breadth-first search.

The tree of Figure 7.4 corresponds to the fixed tuple size formulation. Edges from level i nodes to level $i+1$ nodes are labeled with the value of x_i, which is either zero or one. All paths from the root to a leaf node define the solution space. The left subtree of the root defines all subsets containing w_1, the right subtree defines all subsets not containing w_1, and so on. Now there are 2^4 leaf nodes which represent 16 possible tuples. $\square$

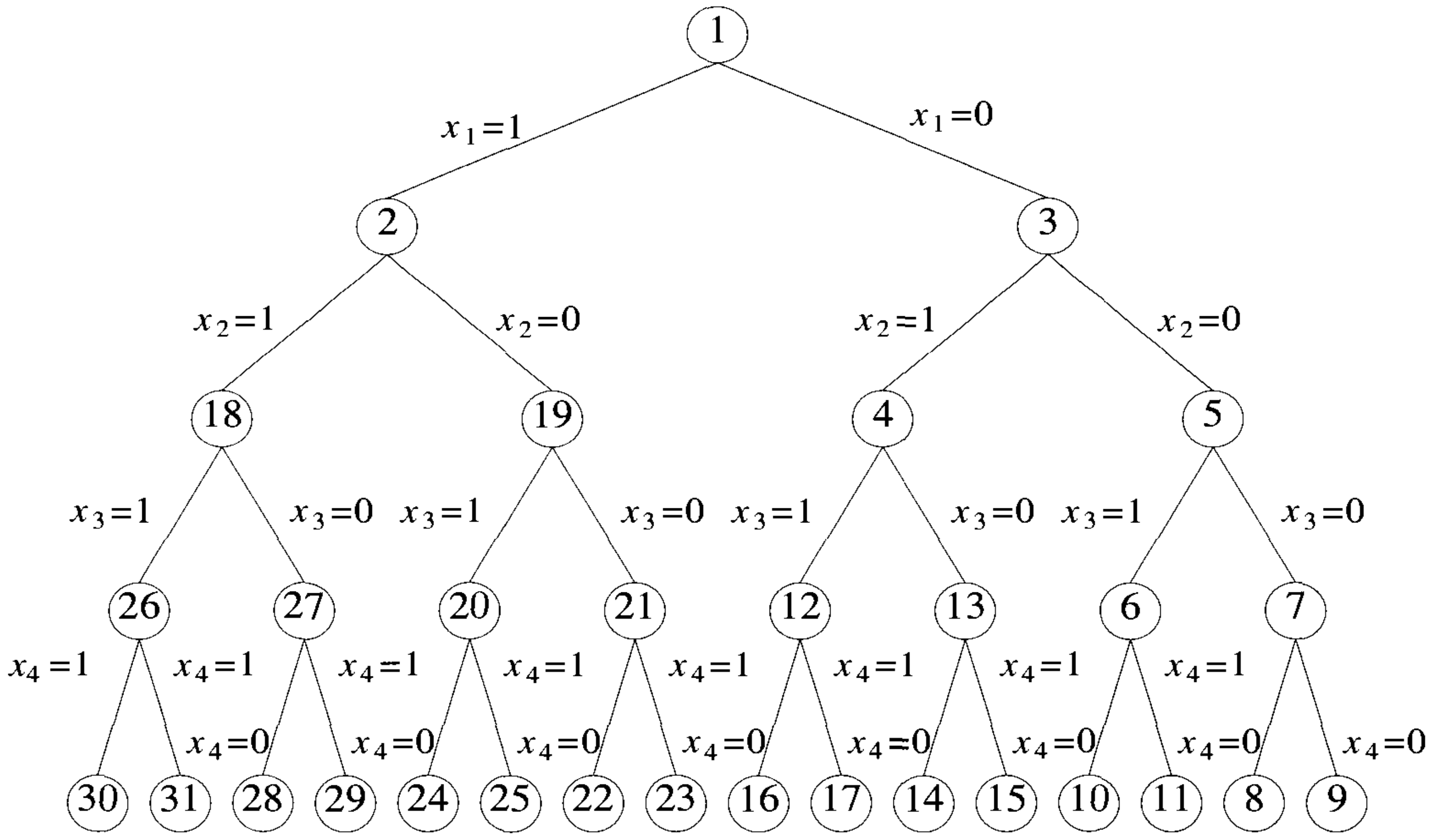


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in *D*-search.

Sum of Subset Problem

- Find a subset of a given set S = { s_1, s_2, …. s_n } of n positive integers whose sum is equal to a given positive integer d
- For example, for S = { 1, 2, 5, 6, 8 } and d=9
- Then there are two solutions:-
- Subset1 = { 1, 2, 6 } = 9
- Subset2 = { 1, 8 } = 9
- Of course, some instances of such problem is not possible i.e. if d=23
- For convenient, all the set elements are sorted in increasing order as shown below-

$$s_1 \le s_2 \le ..... \le s_n$$

Sum of subset example
S = { 3, 5, 6, 7 } and d = 15

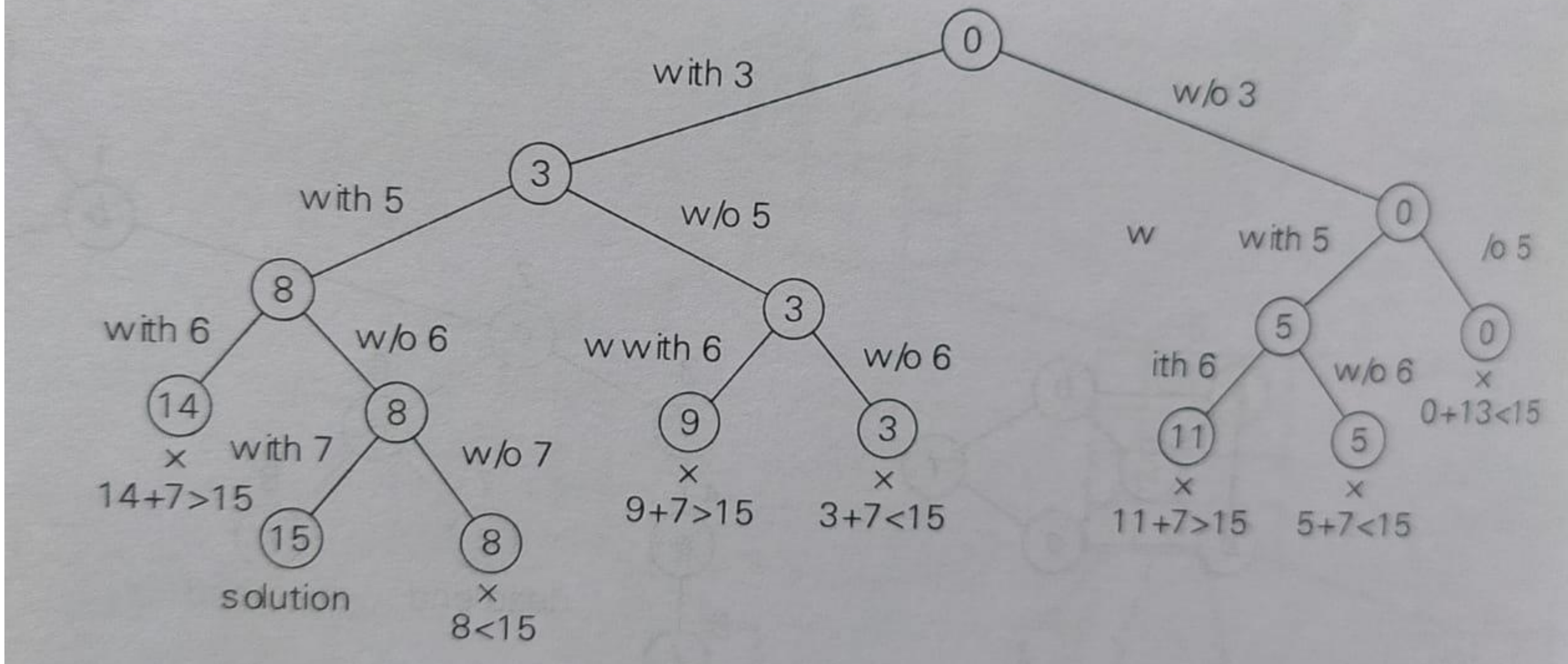


FIGURE 12.4 Complete state-space tree of the backtracking algorithm applied to the instance $S = \{3, 5, 6, 7\}$ and $d = 15$ of the subset-sum problem. The number inside a node is the sum of the elements already included in subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

Sum of Subsets

Suppose we are given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m. This is called the *sum of subsets* problem. Examples 7.2 and 7.4 showed how we could formulate this problem using either fixed- or variable-sized tuples. We consider a backtracking solution using the fixed tuple size strategy. In this case the element x_i of the solution vector is either one or zero depending on whether the weight w_i is included or not.

The children of any node in Figure 7.4 are easily generated. For a node at level i the left child corresponds to $x_i = 1$ and the right to $x_i = 0$.

A simple choice for the bounding functions is $B_k(x_1, \ldots, x_k) =$ true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

$$B_k(x_1, \ldots, x_k) = \mathit{true} \text{ iff } \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

$$\text{and } \sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m \tag{7.1}$$

```
Algorithm SumOfSub($s, k, r$)
// Find all subsets of $w[1:n]$ that sum to $m$. The values of $x[j]$,
// $1 \leq j < k$, have already been determined. $s = \sum_{j=1}^{k-1} w[j] * x[j]$
// and $r = \sum_{j=k}^{n} w[j]$. The $w[j]$'s are in nondecreasing order.
// It is assumed that $w[1] \leq m$ and $\sum_{i=1}^{n} w[i] \geq m$.
{
    // Generate left child. Note: $s + w[k] \leq m$ since $B_{k-1}$ is true.
    $x[k] := 1$;
    if ($s + w[k] = m$) then write ($x[1:k]$); // Subset found
        // There is no recursive call here as $w[j] > 0,\ 1 \leq j \leq n$.
    else  if ($s + w[k] + w[k+1] \leq m$)
          then SumOfSub($s + w[k], k+1, r - w[k]$);
    // Generate right child and evaluate $B_k$.
    if (($s + r - w[k] \geq m$) and ($s + w[k+1] \leq m$)) then
    {
        $x[k] := 0$;
        SumOfSub($s, k+1, r - w[k]$);
    }
}
```

Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

Algorithm SumOfSub avoids computing $\sum_{i=1}^{k} w_i x_i$ and $\sum_{i=k+1}^{n} w_i$ each time by keeping these values in variables s and r respectively. *The algorithm assumes $w_1 \leq m$ and $\sum_{i=1}^{n} w_i \geq m$.* The initial call is SumOfSub$(0, 1, \sum_{i=1}^{n} w_i)$. It is interesting to note that the algorithm does not explicitly use the test $k > n$ to terminate the recursion. This test is not needed as on entry to the algorithm, $s \neq m$ and $s + r \geq m$. Hence, $r \neq 0$ and so k can be no greater than n. Also note that in the **else if** statement (line 11), since $s + w_k < m$ and $s + r \geq m$, it follows that $r \neq w_k$ and hence $k + 1 \leq n$. Observe also that if $s + w_k = m$ (line 9), then $x_{k+1}, \ldots, x_n$ must be zero. These zeros are omitted from the output of line 9. In line 11 we do not test for $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$, as we already know $s + r \geq m$ and $x_k = 1$.

Example 7.6 Figure 7.10 shows the portion of the state space tree generated by function SumOfSub while working on the instance $n = 6$, $m = 30$, and $w[1:6] = \{5, 10, 12, 13, 15, 18\}$. The rectangular nodes list the values of s, k, and r on each of the calls to SumOfSub. Circular nodes represent points at which subsets with sums m are printed out. At nodes A, B, and C the output is respectively (1, 1, 0, 0, 1), (1, 0, 1, 1), and (0, 0, 1, 0, 0, 1). Note that the tree of Figure 7.10 contains only 23 rectangular nodes. The full state space tree for $n = 6$ contains $2^6 - 1 = 63$ nodes from which calls could be made (this count excludes the 64 leaf nodes as no call need be made from a leaf). □

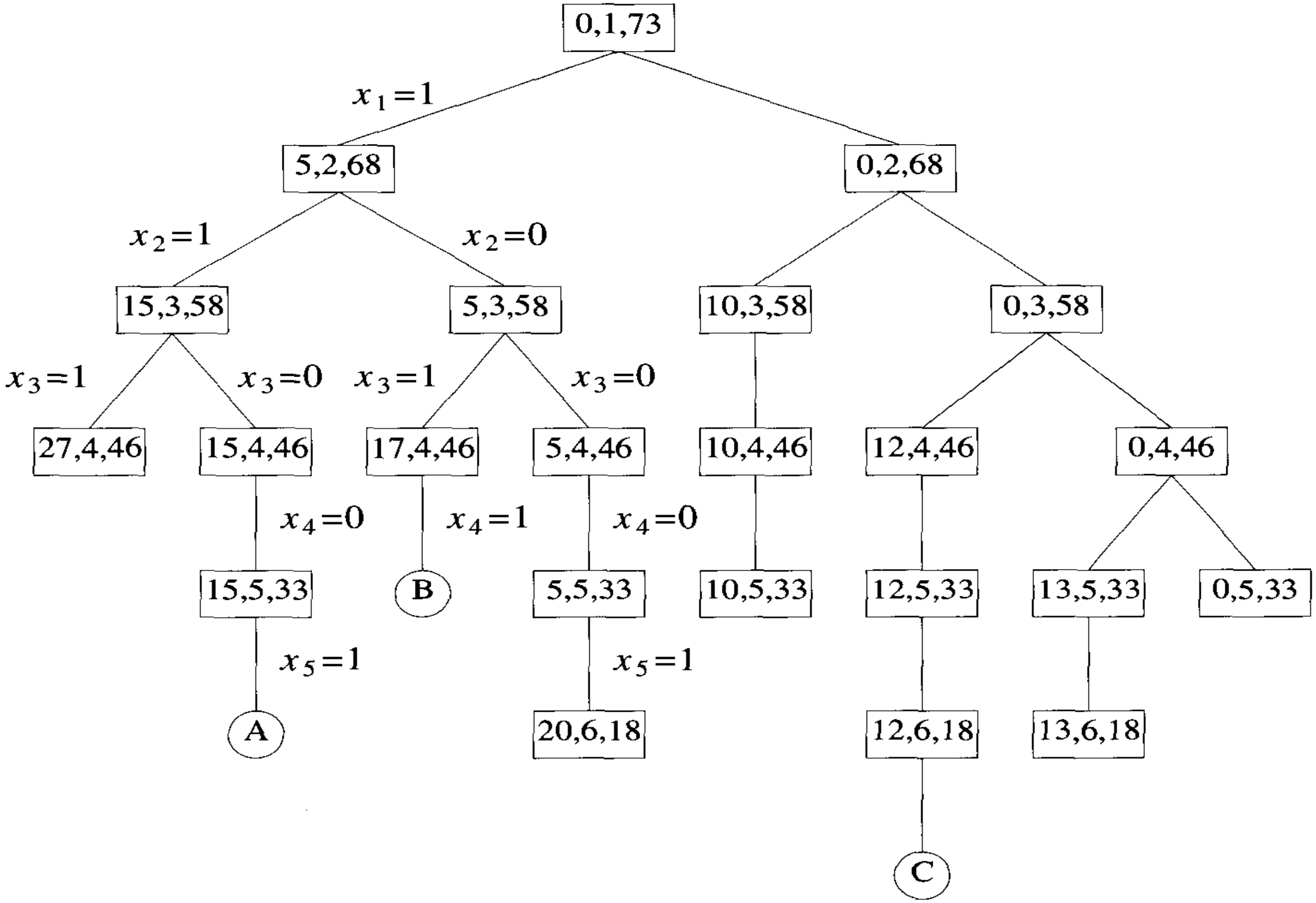
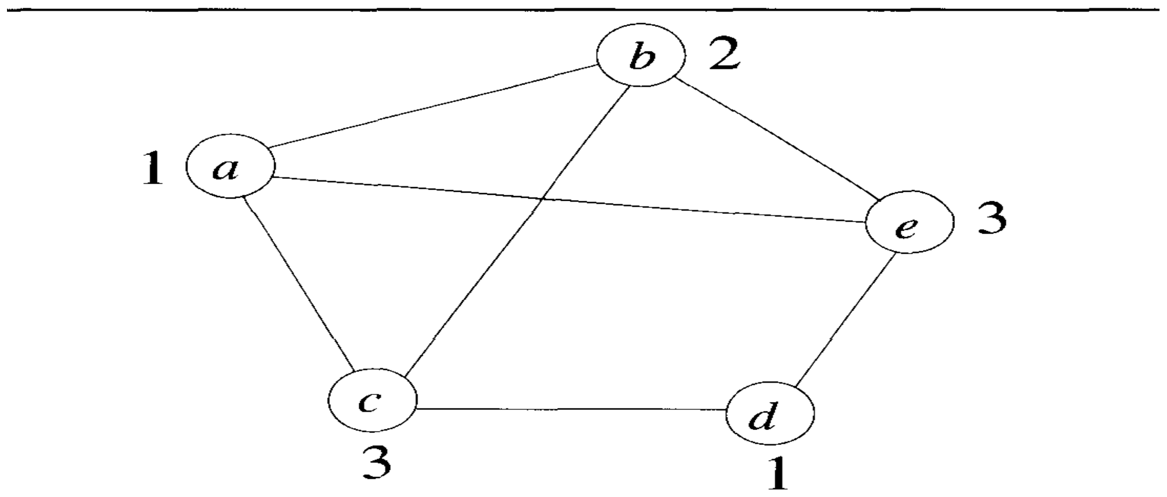


Figure 7.10 Portion of state space tree generated by SumOfSub

Graph Coloring Problem

Graph Coloring

- Let G be a graph and m be a given positive integer
- Then the nodes of graph G can be colored in such a way that no TWO adjacent nodes have the same color yet only m colors are used
- This is termed the m-colorability decision problem
- If d is the degree of graph, then it can be colored with d+1 colors
- The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored
- This integer is referred to as the chromatic number of the graph



- The above graph can be colored with three colors 1, 2 and 3
- The color of each node is indicated next to it
- Three colors are needed to color this graph and hence this graph's chromatic number is 3

Suppose we represent a graph by its adjacency matrix $G[1:n,1:n]$, where $G[i,j]=1$ if (i,j) is an edge of G, and $G[i,j]=0$ otherwise. The colors are represented by the integers $1,2,\ldots,m$ and the solutions are given by the n-tuple $(x_1,\ldots,x_n)$, where x_i is the color of node i. Using the recursive backtracking formulation as given in Algorithm 7.1, the resulting algorithm is mColoring (Algorithm 7.7). The underlying state space tree used is a tree of degree m and height $n+1$. Each node at level i has m children corresponding to the m possible assignments to x_i, $1 \leq i \leq n$. Nodes at

level $n+1$ are leaf nodes. Figure 7.13 shows the state space tree when $n = 3$ and $m = 3$.

Function mColoring is begun by first assigning the graph to its adjacency matrix, *setting the array $x[\,]$ to zero*, and then invoking the statement mColoring(1);.

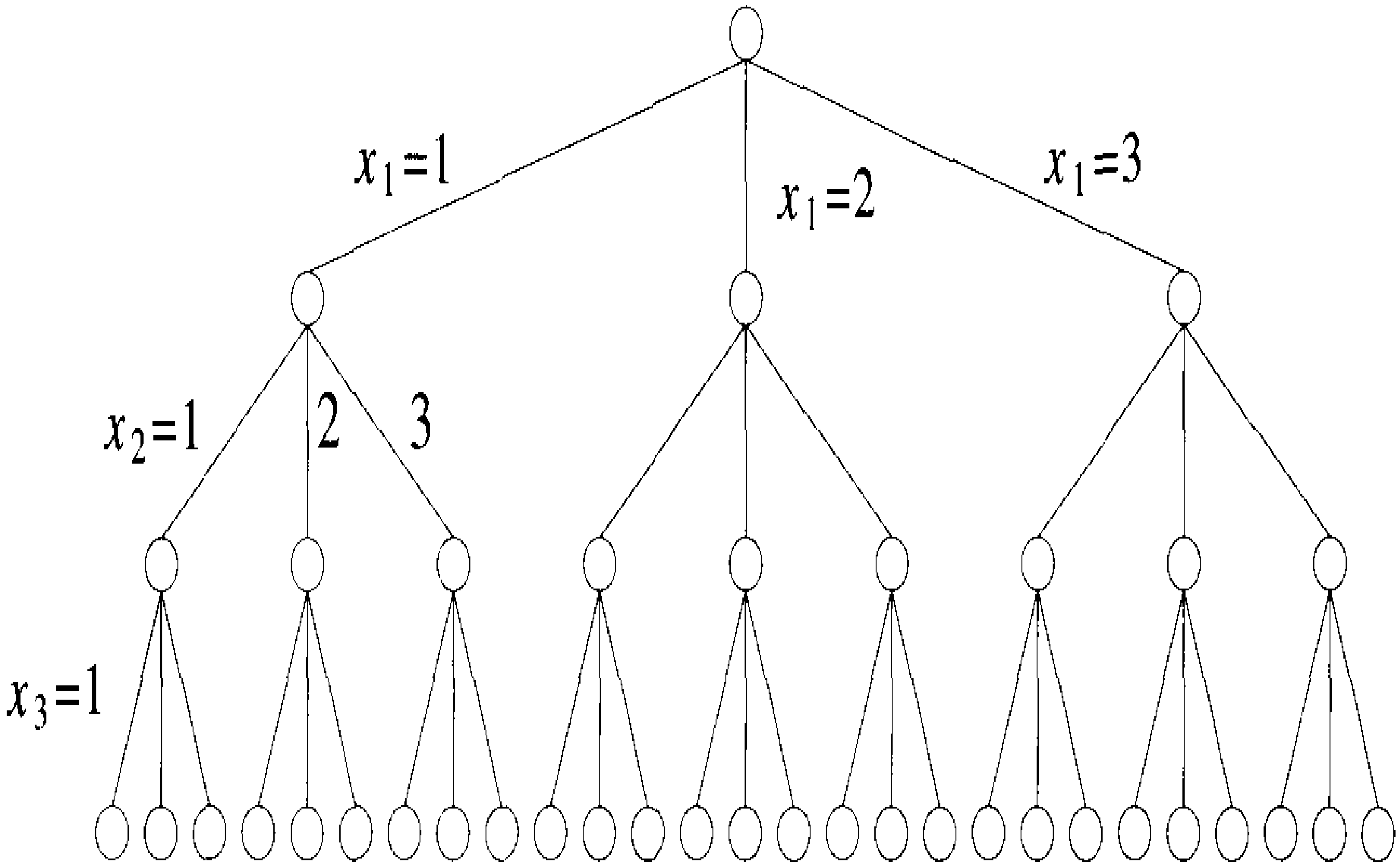


Figure 7.13 State space tree for mColoring when n =3 and m = 3

```
Algorithm mColoring($k$)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix $G[1:n, 1:n]$. All assignments of $1, 2, \ldots, m$ to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. $k$ is the index
// of the next vertex to color.
{
    repeat
    {// Generate all legal assignments for $x[k]$.
        NextValue($k$); // Assign to $x[k]$ a legal color.
        if ($x[k] = 0$) then return; // No new color possible
        if ($k = n$) then     // At most $m$ colors have been
                              // used to color the $n$ vertices.
            write ($x[1:n]$);
        else mColoring($k+1$);
    } until (false);
}
```

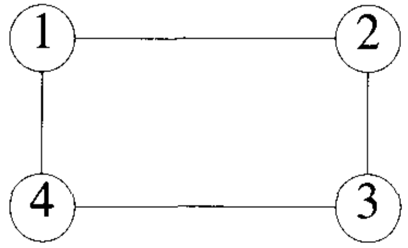
Algorithm 7.7 Finding all m-colorings of a graph

```
Algorithm NextValue($k$)
// $x[1], \ldots, x[k-1]$ have been assigned integer values in
// the range $[1, m]$ such that adjacent vertices have distinct
// integers. A value for $x[k]$ is determined in the range
// $[0, m]$. $x[k]$ is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex $k$. If no such color exists, then $x[k]$ is 0.
{
    repeat
    {
        $x[k] := (x[k] + 1)$ mod $(m + 1)$; // Next highest color.
        if $(x[k] = 0)$ then return; // All colors have been used.
        for $j := 1$ to $n$ do
        {   // Check if this color is
            // distinct from adjacent colors.
            if $((G[k, j] \neq 0)$ and $(x[k] = x[j]))$
            // If $(k, j)$ is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if $(j = n + 1)$ then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

Algorithm 7.8 Generating a next color

For instance, Figure 7.14 shows a simple graph containing four nodes. Below that is the tree that is generated by mColoring. Each path to a leaf represents a coloring using at most three colors. Note that only 12 solutions exist with *exactly* three colors. In this tree, after choosing $x_1 = 2$ and $x_2 = 1$, the possible choices for x_3 are 2 and 3. After choosing $x_1 = 2$, $x_2 = 1$, and $x_3 = 2$, possible values for x_4 are 1 and 3. And so on.

An upper bound on the computing time of mColoring can be arrived at by noticing that the number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$. At each internal node, $O(mn)$ time is spent by NextValue to determine the children corresponding to legal colorings. Hence the total time is bounded by $\sum_{i=0}^{n-1} m^{i+1}n = \sum_{i=1}^{n} m^i n = n(m^{n+1} - 2)/(m-1) = O(nm^n)$.



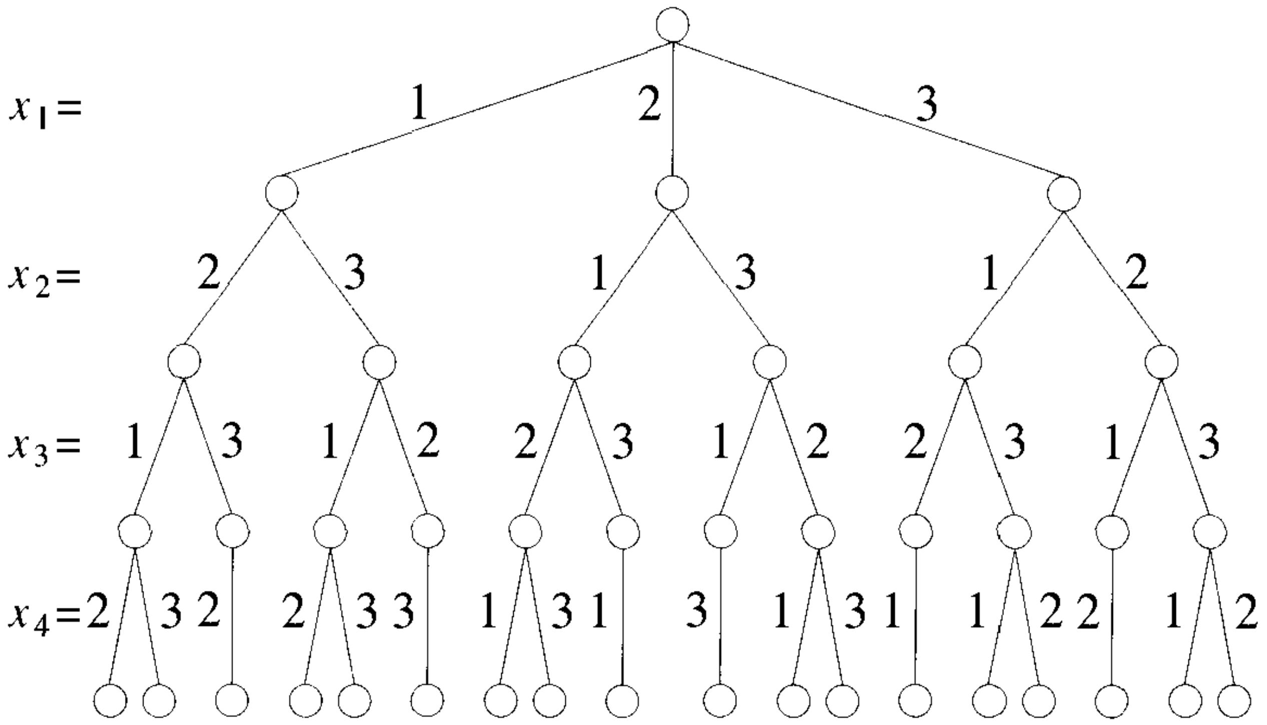


Figure 7.14 A 4-node graph and all possible 3-colorings

Hamiltonian Cycle

Hamiltonian Cycle

- Let G = (V, E) be a connected graph with n vertices
- A Hamiltonian cycle is a round-trip path along n edges of G that visits every vertex once and returns to its starting position
- In other words if a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices of G are visited in the order $v_1, v_2, \ldots, V_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \le i \le n$, and the v_i are distinct except for v_1 and v_{n+1}, which are equal.

The graph $G1$ of Figure 7.15 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph $G2$ of Figure 7.15 contains no Hamiltonian cycle. There is no known easy way to determine whether a given graph contains a Hamiltonian cycle. We now look at a backtracking algorithm that finds all the Hamiltonian cycles in a graph. The graph may be directed or undirected. Only distinct cycles are output.

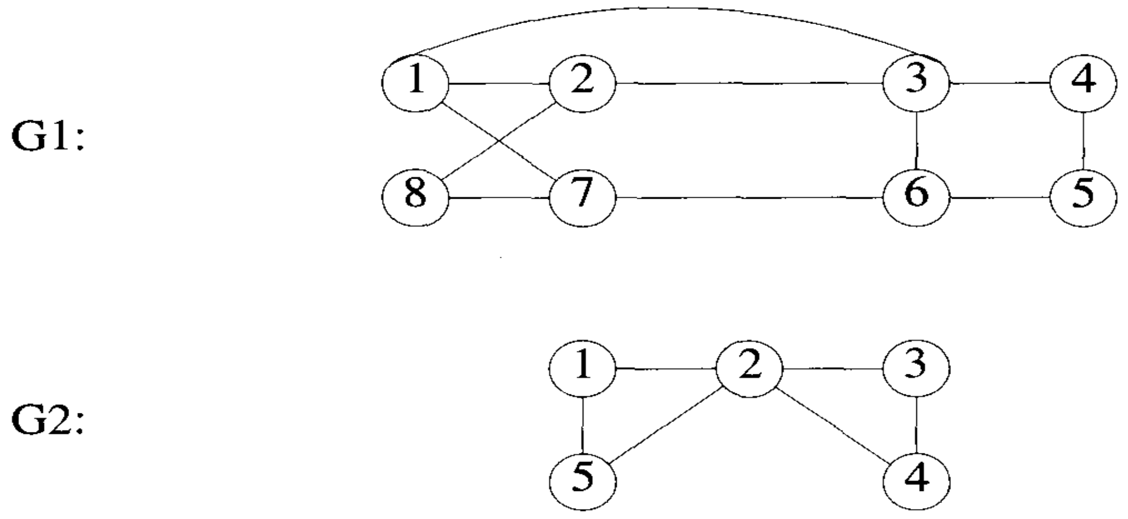


Figure 7.15 Two graphs, one containing a Hamiltonian cycle

The backtracking solution vector $(x_1, \ldots, x_n)$ is defined so that x_i represents the ith visited vertex of the proposed cycle. Now all we need do is determine how to compute the set of possible vertices for x_k if $x_1, \ldots, x_{k-1}$ have already been chosen. If $k = 1$, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to x_{k-1}. The vertex x_n can only be the one remaining vertex and it must be connected to both x_{n-1} and x_1. We begin by presenting function NextValue(k) (Algorithm 7.9), which determines a possible next

vertex for the proposed cycle.

Using NextValue we can particularize the recursive backtracking schema to find all Hamiltonian cycles (Algorithm 7.10). This algorithm is started by first initializing the adjacency matrix $G[1:n, 1:n]$, then setting $x[2:n]$ to zero and $x[1]$ to 1, and then executing Hamiltonian(2).

```
Algorithm NextValue($k$)
// $x[1:k-1]$ is a path of $k-1$ distinct vertices. If $x[k]=0$, then
// no vertex has as yet been assigned to $x[k]$. After execution,
// $x[k]$ is assigned to the next highest numbered vertex which
// does not already appear in $x[1:k-1]$ and is connected by
// an edge to $x[k-1]$. Otherwise $x[k]=0$. If $k=n$, then
// in addition $x[k]$ is connected to $x[1]$.
{
    repeat
    {
        $x[k] := (x[k]+1)$ mod $(n+1)$; // Next vertex.
        if $(x[k]=0)$ then return;
        if $(G[x[k-1], x[k]] \neq 0)$ then
        { // Is there an edge?
            for $j := 1$ to $k-1$ do if $(x[j]=x[k])$ then break;
                                  // Check for distinctness.
            if $(j=k)$ then // If true, then the vertex is distinct.
                if $((k<n)$ or $((k=n)$ and $G[x[n], x[1]] \neq 0))$
                    then  return;
        }
    } until (false);
}
```

Algorithm 7.9 Generating a next vertex

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1 : n, 1 : n]. All cycles begin at node 1.
{
    repeat
    { // Generate values for x[k].
        NextValue(k); // Assign a legal next value to x[k].
        if (x[k] = 0) then return;
        if (k = n) then write (x[1 : n]);
        else Hamiltonian(k + 1);
    } until (false);
}
```

Algorithm 7.10 Finding all Hamiltonian cycles

Programme & Bound

Programme-and-Bound

- Branch-and-Bound is similar to backtracking, but it cut off a branch of the problem’s state-space tree as soon as we can deduce that it cannot lead to a solution
- This idea is useful to find an optimization problem, one that seeks to minimize or maximize an objective function, usually subject to some constraints.
- A feasible solution is a point in the problem’s search space that satisfies all the problem’s constraints
- While, an optimal solution is a feasible solution with the best value of the objective function

- Compared to backtracking, branch-and-bound requires two additional items:
  - a way to provide, for every node of a state-space tree, a bound on the best value of the objective function1 on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
  - the value of the best solution seen so far

Branch-and-Bound Algorithm

- Three reasons to terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm-

1. The value of the node's bound is not better than the value of the best solution seen so far
2. The node represents no feasible solutions because the constraints of the problem are already violated
3. The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)

Assignment Problem Statement

- Assignment problem is a problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible
- An instance of the assignment problem is specified by an n-by-n cost matrix C so that we can state the problem as follows - select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible

$$C = \begin{array}{c} \\ \\ \\ \\ \end{array}\begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \\ \end{array}$$

$$\begin{array}{c|cccc|l} & \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} & \\ \hline & 9 & 2 & 7 & 8 & \text{person } a \\ C = & 6 & 4 & 3 & 7 & \text{person } b \\ & 5 & 8 & 1 & 8 & \text{person } c \\ & 7 & 6 & 9 & 4 & \text{person } d \end{array}$$

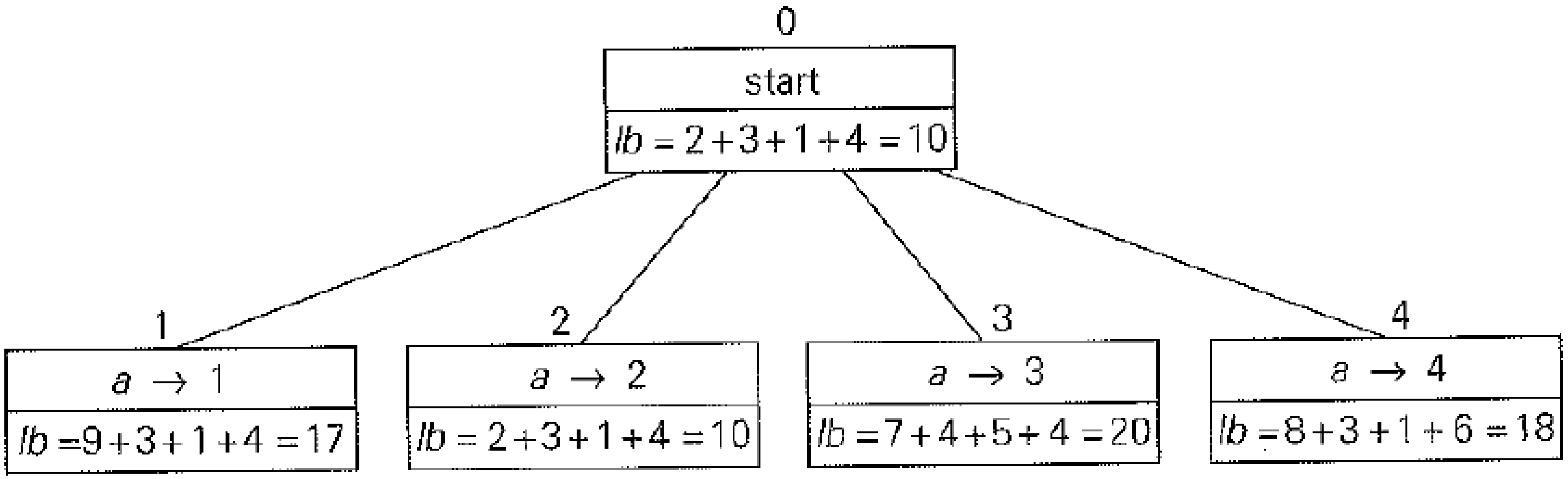


FIGURE 12.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person *a* and the lower bound value, *lb*, for this node.

Assignment Problem Solution

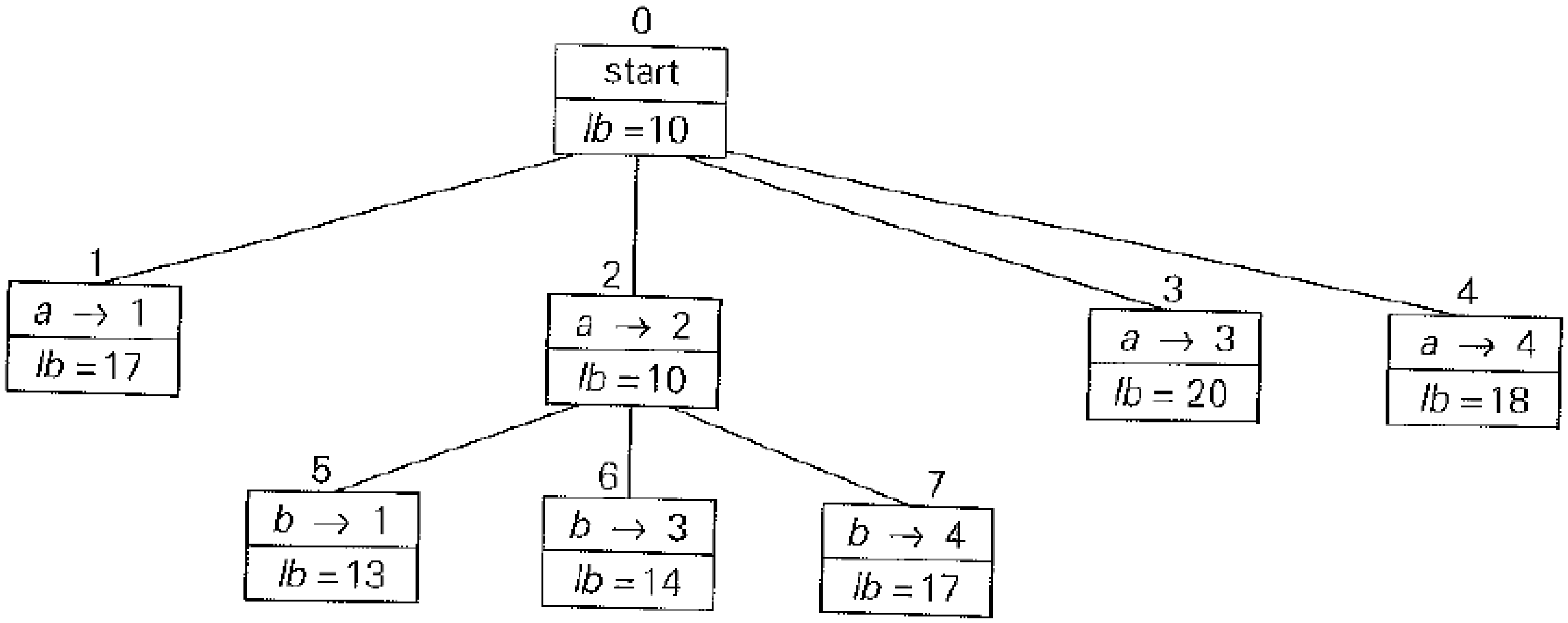


FIGURE 12.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

Assignment Problem- Optimal(Minimum) Solution

a->2, b->1, c->3, d->4 = 2 + 6 + 1 + 4 = 13

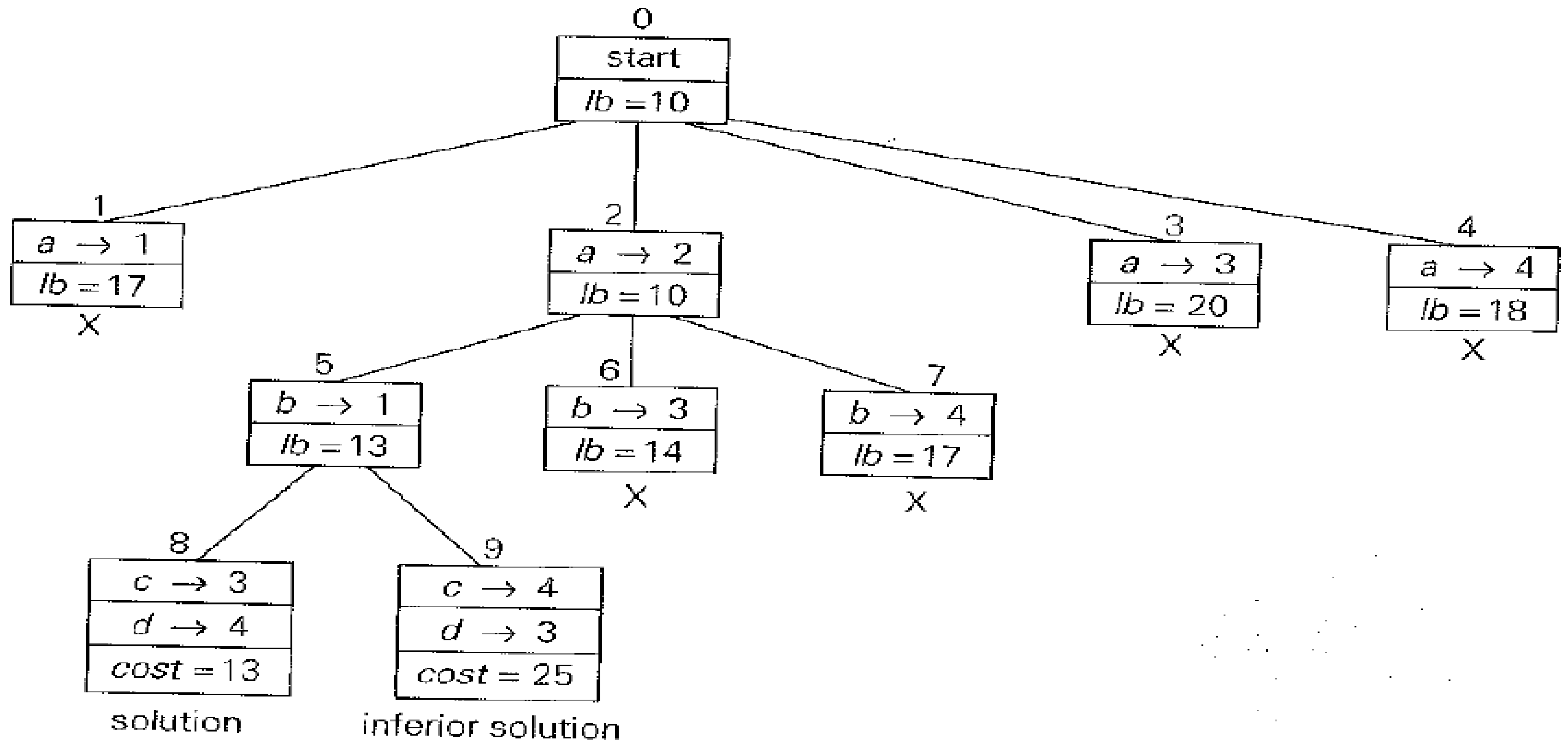
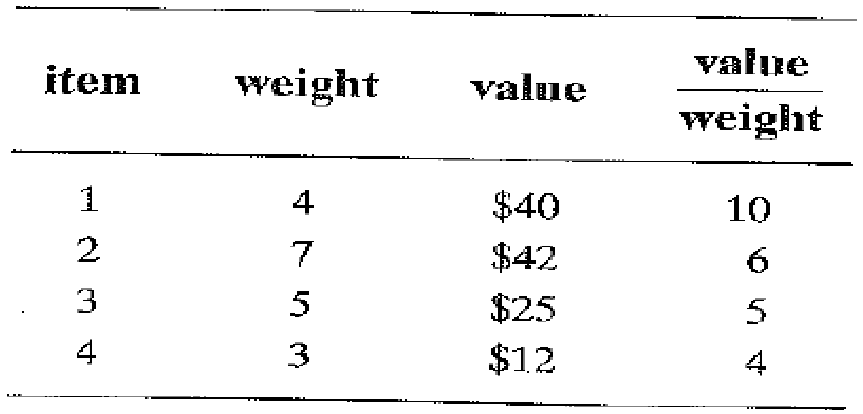


FIGURE 12.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

BRANCH-AND-BOUND Knapsack Problem

- Given n items of known weights w_i and the values v_i, where i = 1, 2, 3, …, n, and a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack
- For convenient, must arrange items in descending order by their value-to-weight ratios as shown in below example-



item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity *W* is 10.

LC(Least Cost) Branch and Bound Solution

- To use branch and bound technique to solve any problem we need to construct state space tree for given problem
- As we know that 0/1 Knapsack problem is maximization problem(to get maximum profit), here we will solve it using minimization problem
- Clearly, $\Sigma p_i x_i$ is maximized iff $-\Sigma p_i x_i$ is minimized
- This modified 0/1 knapsack problem is stated as below-

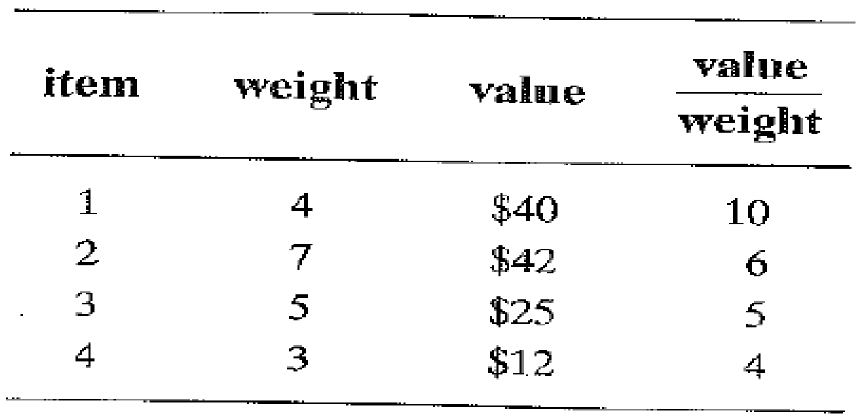
$$\text{minimize } -\sum_{i=1}^{n} p_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} w_i x_i \le m \qquad (8.1)$$

$$x_i = 0 \text{ or } 1, \quad 1 \le i \le n$$

- Each node on the i^{th} level of this tree, $0 \le i \le 1$, represents all the subsets of n items that include a particular selection made from the first i ordered items.
- This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, while a branch going to the right indicates its exclusion.
- We record the total weight w and the total value v of this selection in the node, along with some upper bound ub on the value of any subset that can be obtained by adding zero or more items to this selection.
- A simple way to compute the upper bound ub is to add to v, the total value of the items already selected, the product of the remaining capacity of the knapsack W - w and the best per unit payoff among the remaining items, which is v_{i+1}/w_{i+1}:

$$ub = v + (W - w)(v_{i+1}/w_{i+1}). \qquad \textbf{(12.1)}$$



item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The knapsack's capacity W is 10.

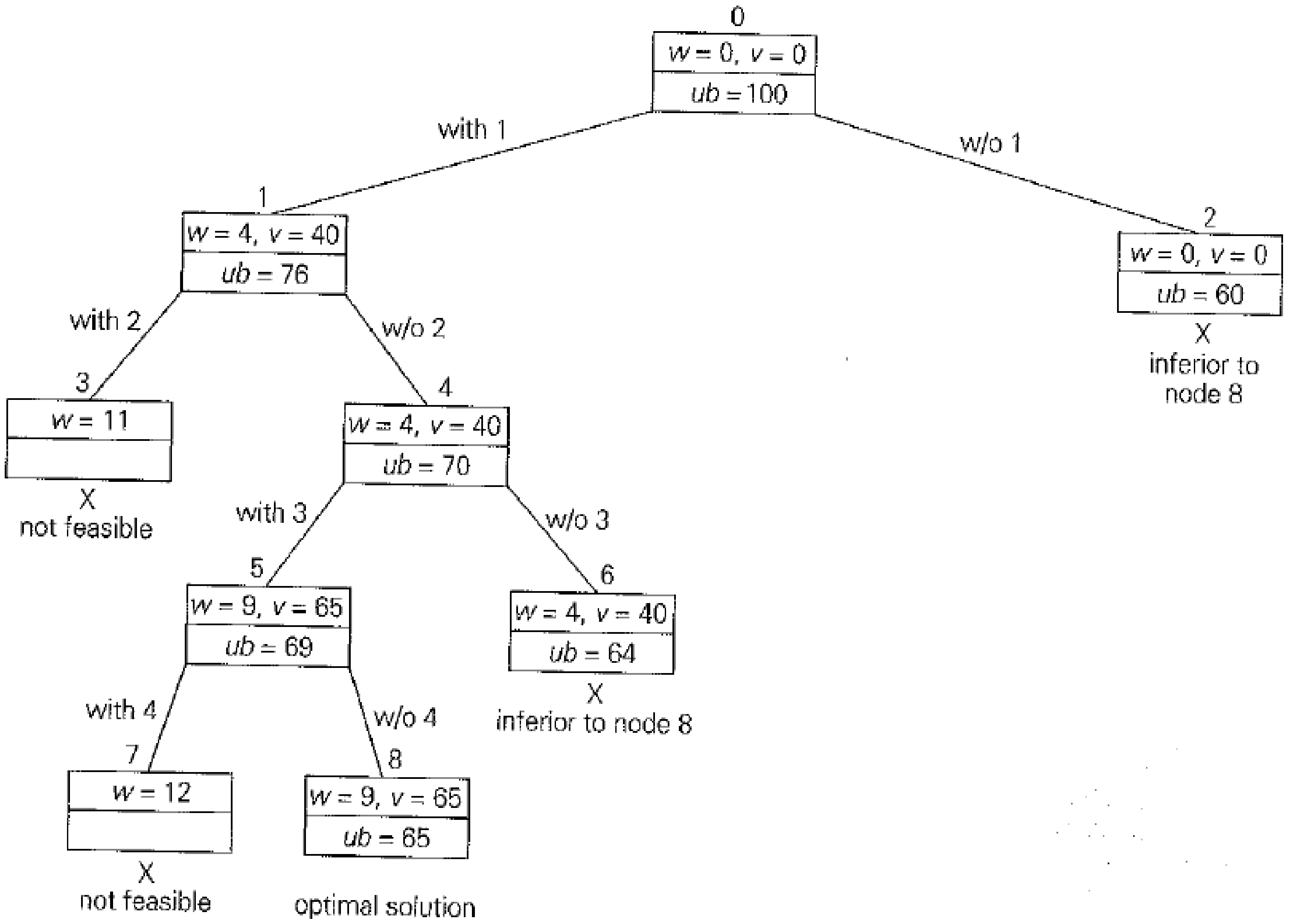
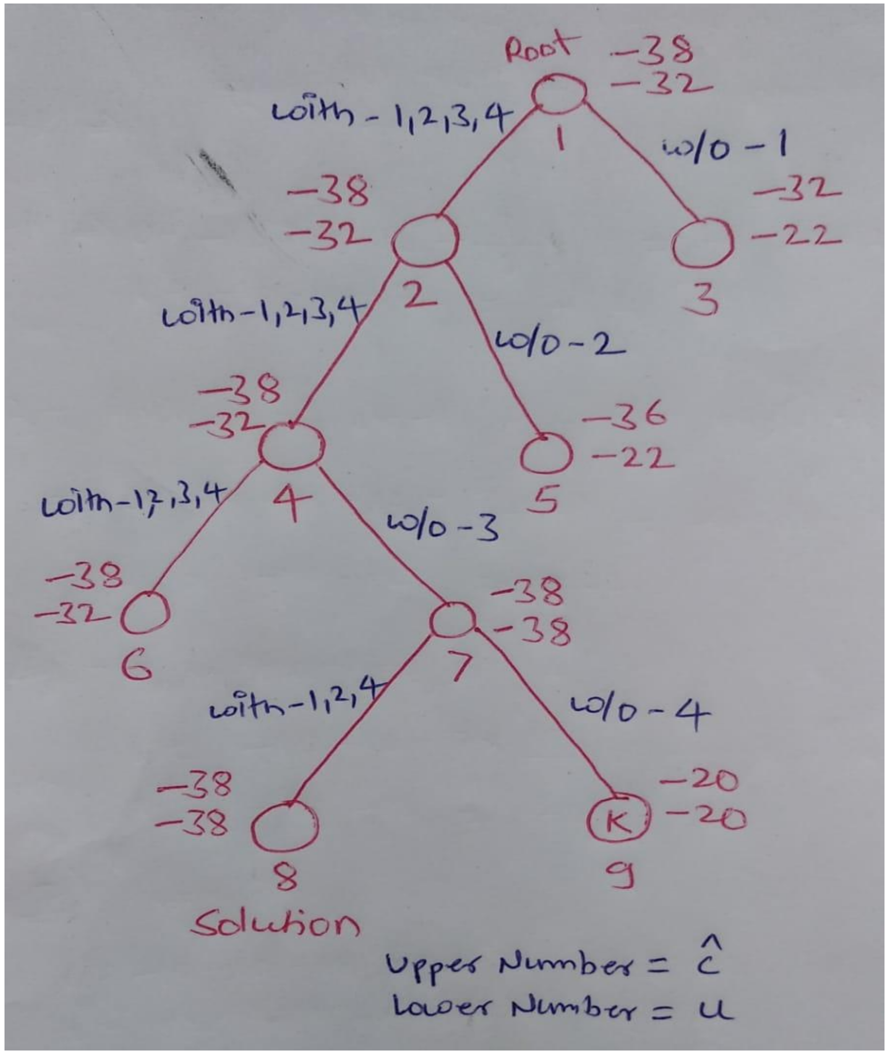
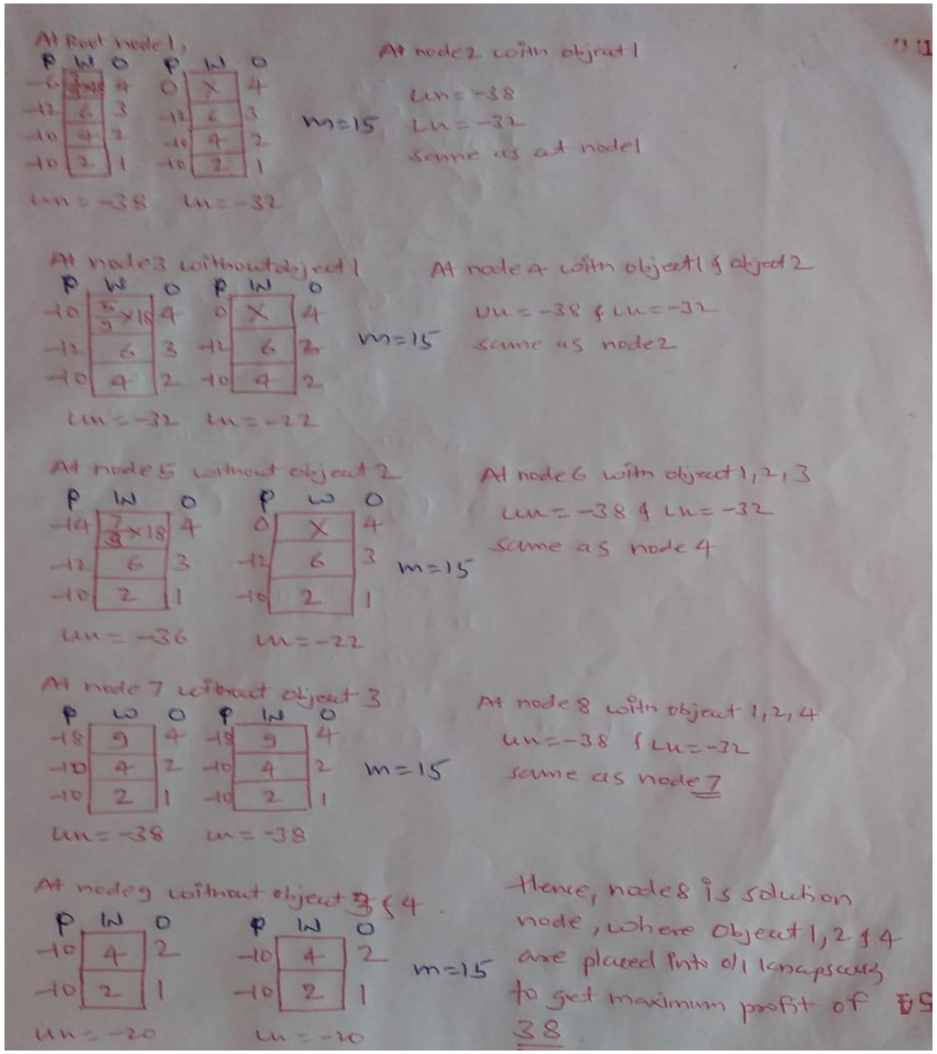


FIGURE 12.8 State-space tree of the branch-and-bound algorithm for the instance of the knapsack problem

For LCBB example - Consider the 0/1 Knapsack instance n=4,(p_1,p_2,p_3,p_4)=(10,10,12,18), (w_1,w_2,w_3,w_4)=(2,4,6,9) and m=15




At Root node 1,

P	W	O
-6	3/9×18	4
-12	6	3
-10	4	2
-10	2	1

un = -38

P	W	O
0	X	4
-12	6	3
-10	4	2
-10	2	1

Ln = -32

m=15

At node 2 with object 1

un = -38

Ln = -32

Same as at node 1

At node 3 without object 1

P	W	O
-10	5/9×18	4
-12	6	3
-10	4	2

un = -32

P	W	O
0	X	4
-12	6	3
-10	4	2

Ln = -22

m=15

At node 4 with object 1 & object 2

Un = -38 & Ln = -32

same as node 2

At node 5 without object 2

P	W	O
-14	7/9×18	4
-12	6	3
-10	2	1

un = -36

P	W	O
0	X	4
-12	6	3
-10	2	1

Ln = -22

m=15

At node 6 with object 1,2,3

un = -38 & Ln = -32

Same as node 4

At node 7 without object 3

P	W	O
-18	9	4
-10	4	2
-10	2	1

un = -38

P	W	O
-18	9	4
-10	4	2
-10	2	1

Ln = -38

m=15

At node 8 with object 1,2,4

un = -38 & Ln = -32

same as node 7

At node 9 without object 3 & 4.

P	W	O
-10	4	2
-10	2	1

un = -20

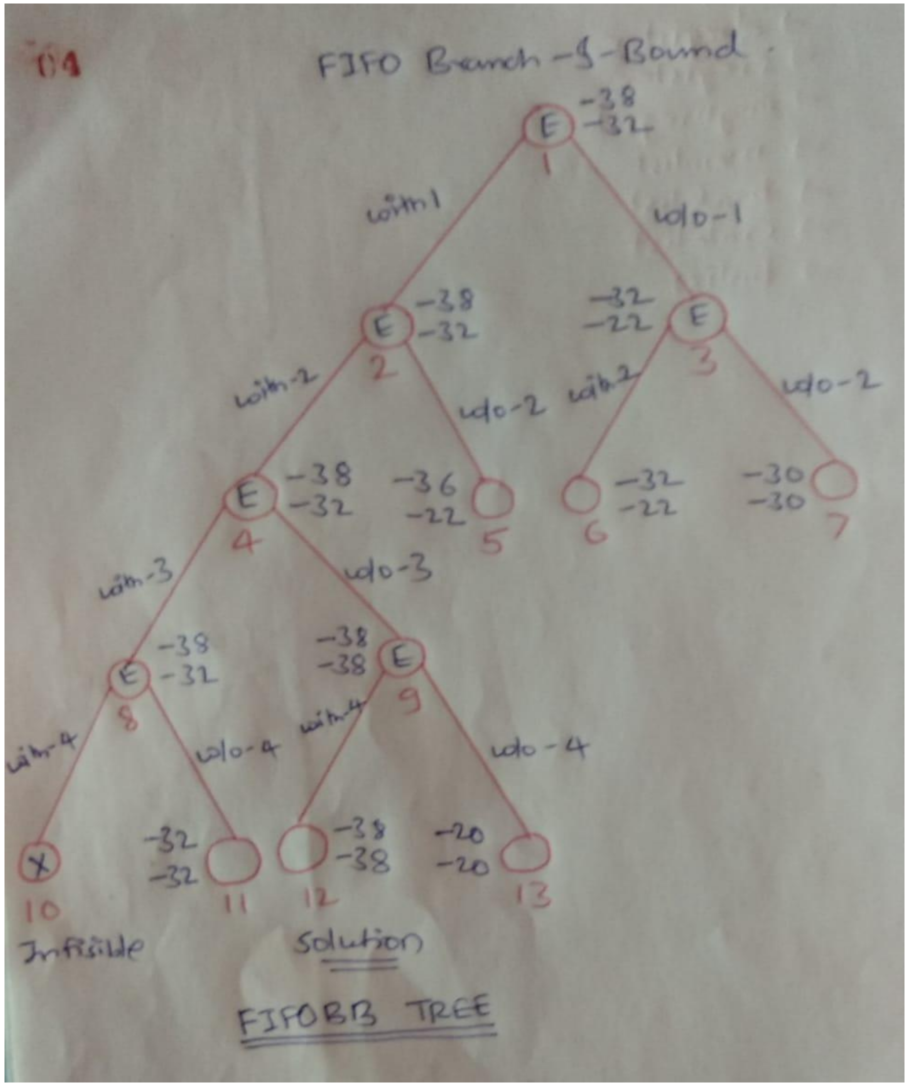
P	W	O
-10	4	2
-10	2	1

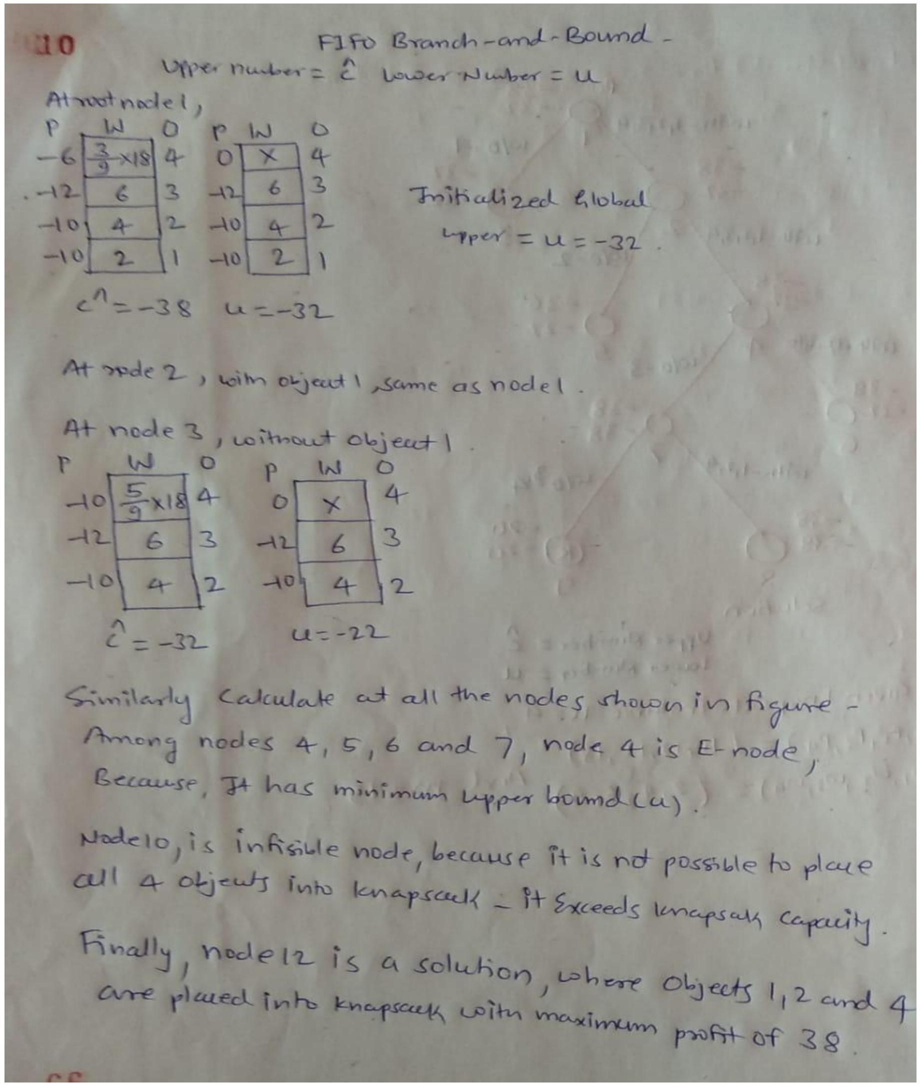
Ln = -20

m=15

Hence, node 8 is solution node, where Object 1, 2 & 4 are placed into 0/1 knapsack to get maximum profit of 38

For FIFOBB Example - Consider the 0/1 Knapsack instance $n=4, (p_1,p_2,p_3,p_4)=(10,10,12,18), (w_1,w_2,w_3,w_4)=(2,4,6,9)$ and $m=15$





10 FIFO Branch-and-Bound -

Upper number = $\hat{c}$ Lower Number = u

At root node 1,

P	W	O
-6	$\frac{3}{9} \times 18$	4
-12	6	3
-10	4	2
-10	2	1

P	W	O
0	X	4
-12	6	3
-10	4	2
-10	2	1

$\hat{c} = -38$ u = -32

Initialized Global upper = u = -32.

At node 2, with object 1, same as node 1.

At node 3, without object 1.

P	W	O
-10	$\frac{5}{9} \times 18$	4
-12	6	3
-10	4	2

P	W	O
0	X	4
-12	6	3
-10	4	2

$\hat{c} = -32$ u = -22

Similarly Calculate at all the nodes shown in figure - Among nodes 4, 5, 6 and 7, node 4 is E-node, Because, It has minimum upper bound (u).

Node 10, is infisible node, because it is not possible to place all 4 objects into knapsack - it Exceeds knapsack capacity.

Finally, node 12 is a solution, where Objects 1, 2 and 4 are placed into knapsack with maximum profit of 38.

Traveling Salesman Problem

- We will be able to apply the branch-and-bound technique to instances of the traveling salesman problem if we come up with a reasonable lower bound on tour lengths.
- One very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix D and multiplying it by the number of cities n.
- For each city i, $1 \le i \le n$, find the sum s_i of the distances from city i to the two nearest cities; compute the sum s of these n numbers; divide the result by 2; and, if all the distances are integers, round up the result to the nearest integer:

$$lb = s/2 \qquad - (12.2)$$

- For example, for the instance in Figure 12.9a, formula (12.2) yields

$$lb = \lceil[(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2\rceil = 14.$$

- Moreover, for any subset of tours that must include particular edges of a given graph, we can modify lower bound (12.2) accordingly.
- For example, for all the Hamiltonian circuits of the graph in Figure 12.9a that must include edge (a, *d), we* get the following lower bound by summing the lengths of the two shortest edges incident with each of the vertices, with the required inclusion of edges *(a, d) and* (d, a):

$$\lceil[(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2\rceil = 16.$$

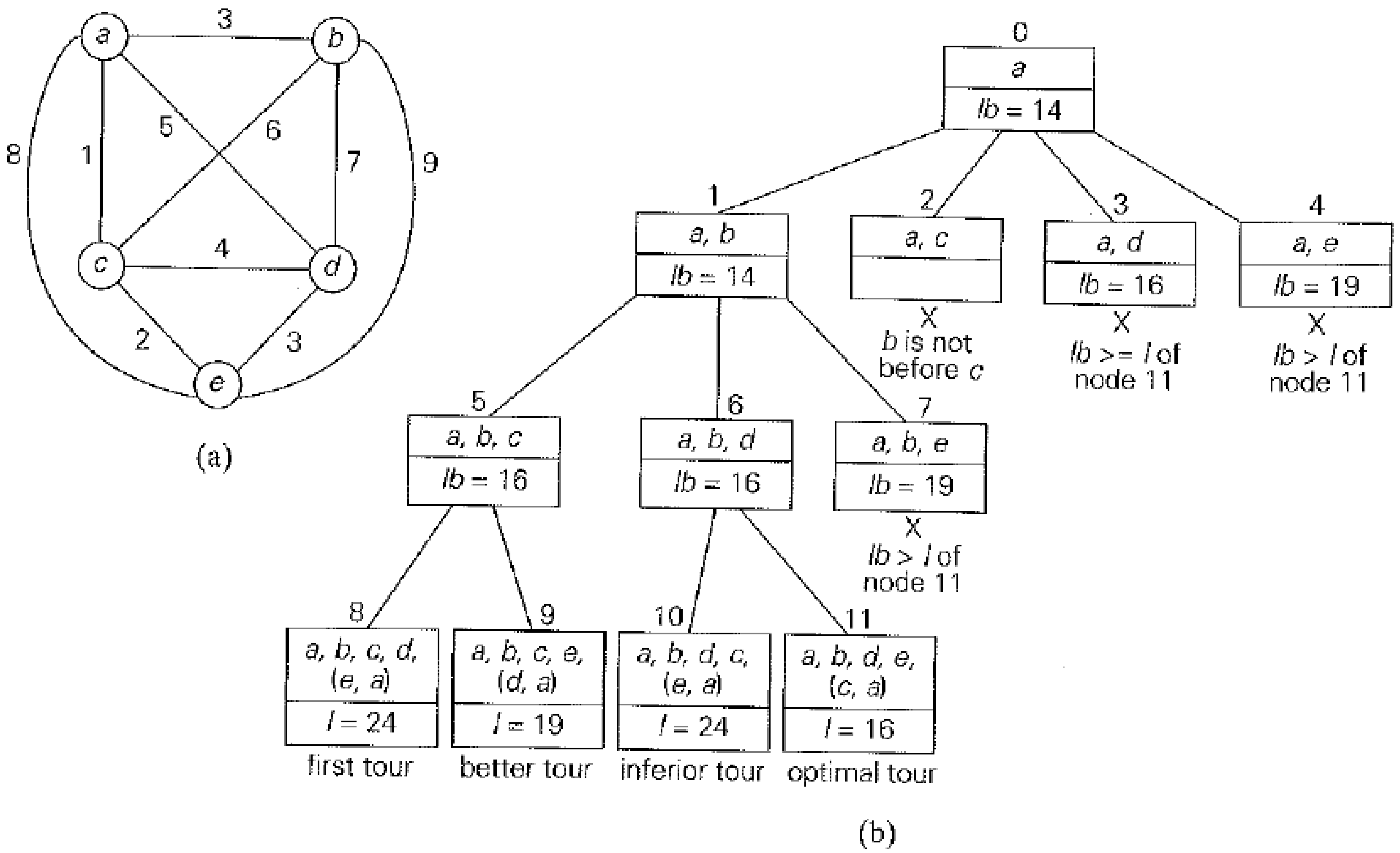


FIGURE 12.9 (a) Weighted graph. (b) State-space tree of the the branch-and-bound algorithm to find the shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

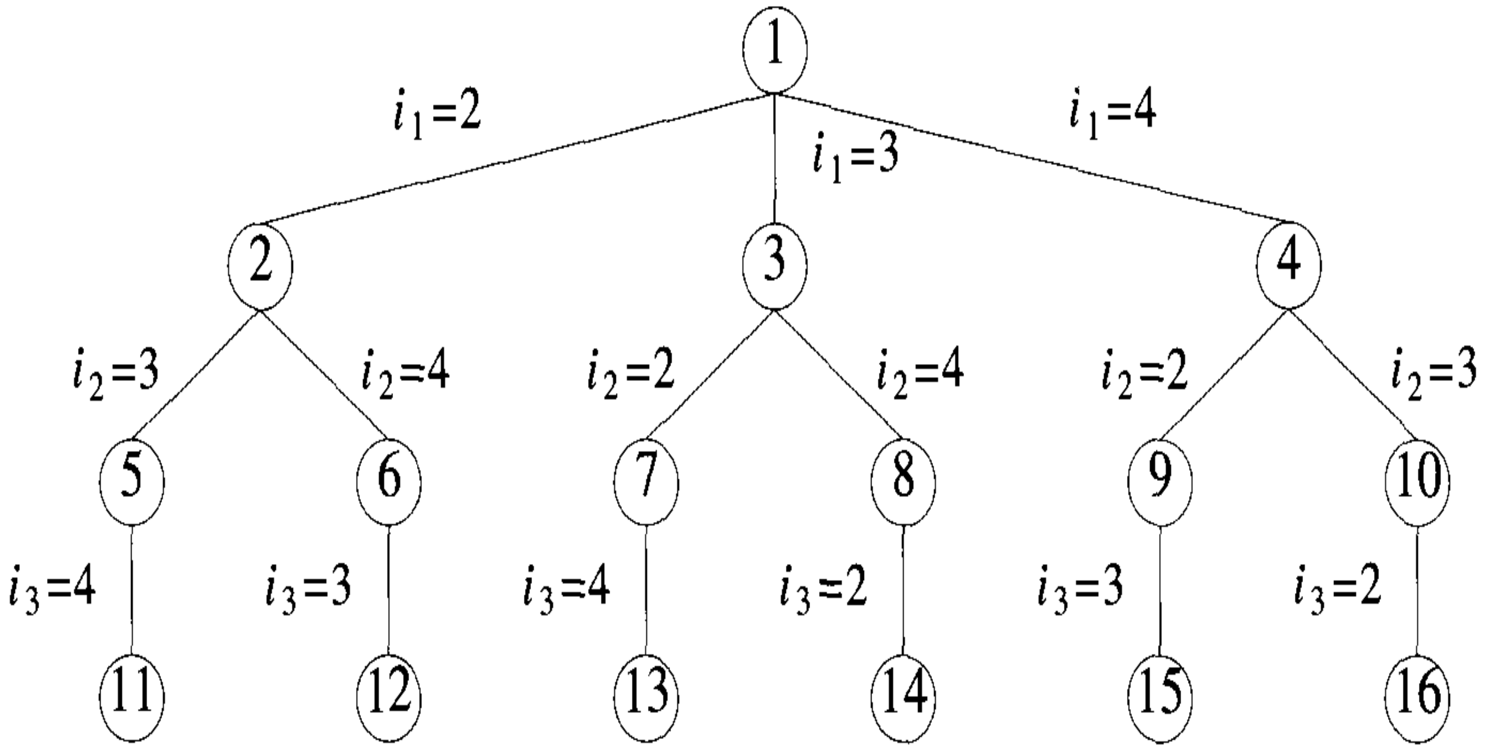


Figure 8.10 State space tree for the traveling salesperson problem with $n = 4$ and $i_0 = i_4 = 1$

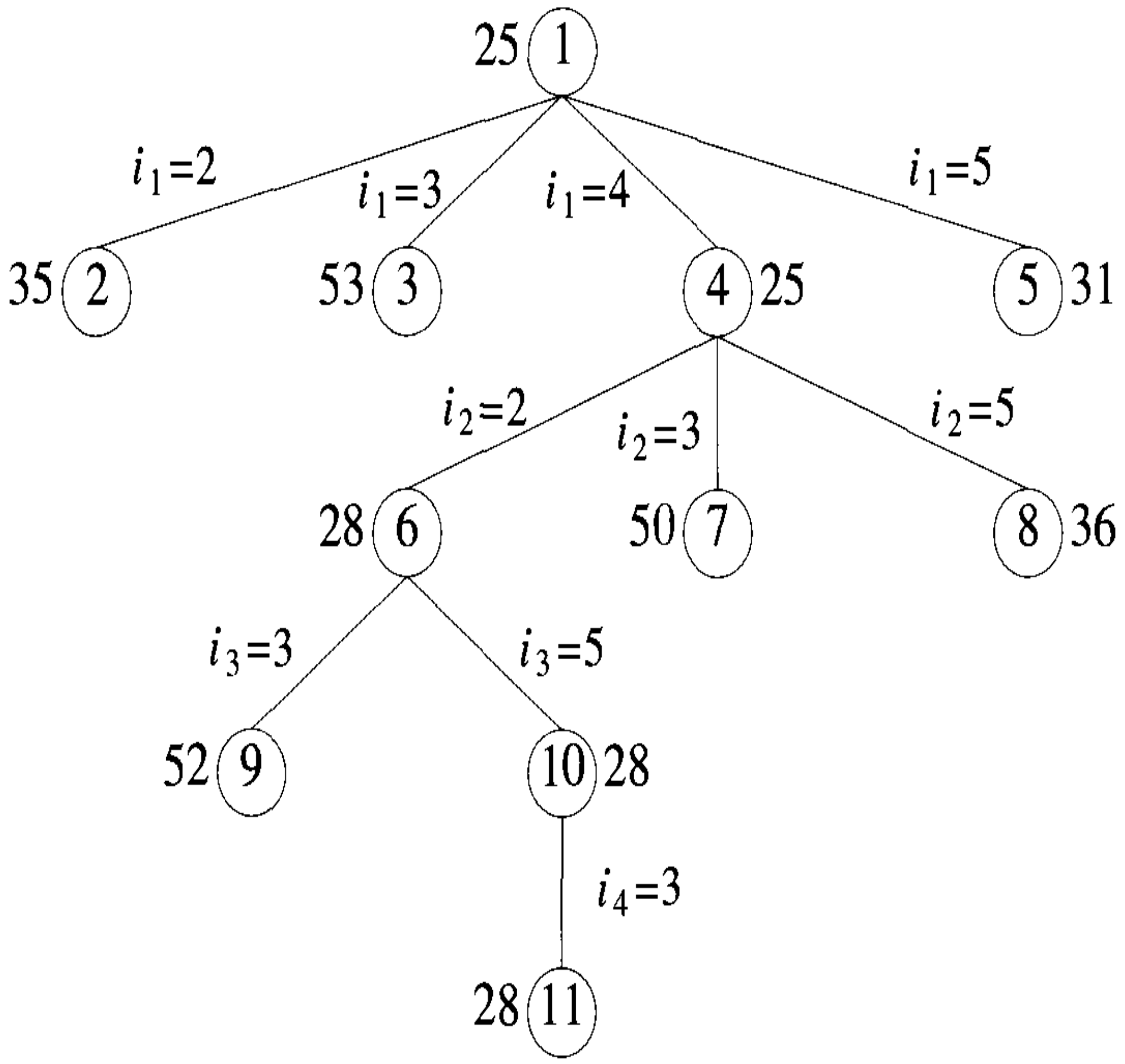
$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost matrix

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced cost matrix
$L = 25$

Figure 8.11 An example



Numbers outside the node are $\hat{c}$ values

Figure 8.13 State space tree generated by procedure LCBB

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1,4,5; node 8

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

(h) Path 1,4,2,3; node 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

(i) Path 1,4,2,5; node 10

Figure 8.13 Reduced cost matrices corresponding to nodes in Figure 8.12

Let us now trace the progress of the LCBB algorithm on the problem instance of Figure 8.11(a). We use $\hat{c}$ and u as above. The initial reduced matrix is that of Figure 8.11(b) and $upper = \infty$. The portion of the state space tree that gets generated is shown in Figure 8.12. Starting with the root node as the E-node, nodes 2, 3, 4, and 5 are generated (in that order). The reduced matrices corresponding to these nodes are shown in Figure 8.13. The matrix of Figure 8.13(b) is obtained from that of 8.11(b) by (1) setting all entries in row 1 and column 3 to ∞, (2) setting the element at position (3, 1) to ∞, and (3) reducing column 1 by subtracting by 11. The $\hat{c}$ for node 3 is therefore 25 + 17 (the cost of edge $\langle 1, 3\rangle$ in the reduced matrix) + 11 = 53. The matrices and $\hat{c}$ value for nodes 2, 4, and 5 are obtained similarly. The value of *upper* is unchanged and node 4 becomes the next E-node. Its children 6, 7, and 8 are generated. The live nodes at this time are nodes 2, 3, 5, 6, 7, and 8. Node 6 has least $\hat{c}$ value and becomes the next E-node. Nodes 9 and 10 are generated. Node 10 is the next E-node. The solution node, node 11, is generated. The tour length for this node is $\hat{c}(11) = 28$ and *upper* is updated to 28. For the next E-node, node 5, $\hat{c}(5) = 31 > upper$. Hence, LCBB terminates with 1, 4, 2, 5, 3, 1 as the shortest length tour.

NP Complete and NP Hard Problems Basic Concept

- In this chapter we are concerned with the distinction between problems that can be solved by a polynomial time algorithm and problems for which no polynomial time algorithm is known
- For many of the problems we know and study, the best algorithms for their solutions have computing times that cluster into two groups
- First group consists of problems whose solution times are bounded by polynomial of small degrees. Examples- Searching(O(logn)), Sorting(O(nlogn))
- Second group made up of problems whose best-known algorithms are non-polynomials. Examples- Travelling Salesman Problem($O(n^2 2^n)$), Knapsack Problems($O(2^{n/2})$)

NP Complete and NP Hard Problems

Basic Concept

- The theory of NP-completeness which we present here does not provide a method of obtaining polynomial time algorithms for problems in the second group
- Nor does it say that algorithms of this complexity do not exist
- Instead, what we do is show that many of the problems for which there are no known polynomial time algorithms are computationally related
- In fact, we establish two classes of problems
- These are given names NP-hard and NP-complete

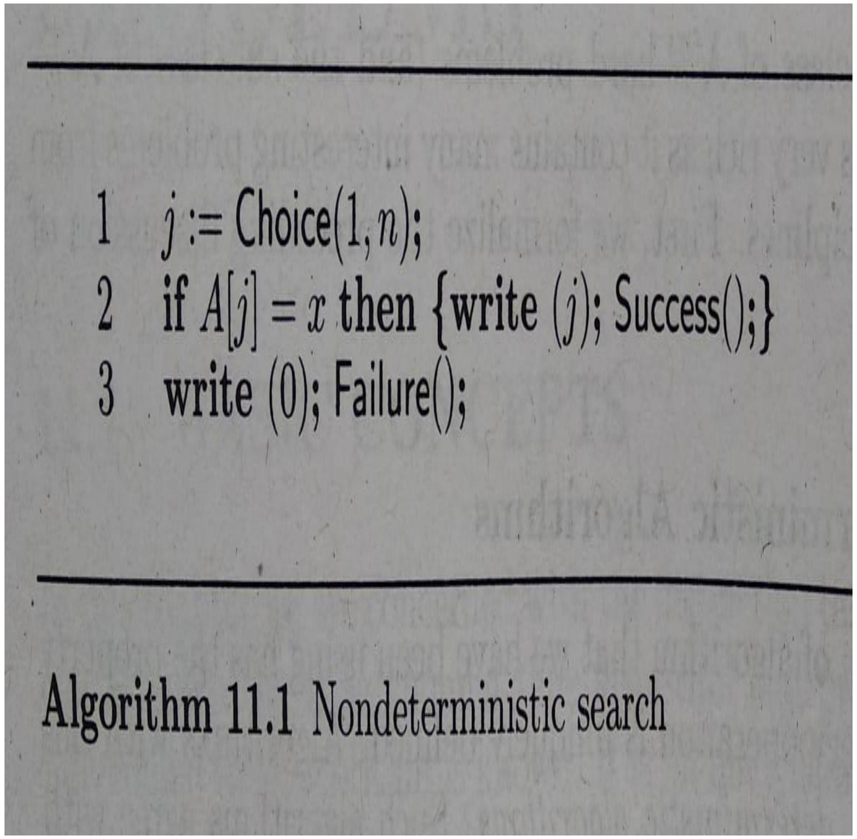
NP Complete and NP Hard Problems
Basic Concept

- A problem that is NP-complete has the property that it can be solved in polynomial time if and only if all other NP-complete problems can also be solved in polynomial time
- If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time
- All NP-complete problems are NP-hard, but some NP-hard problems are not known to be NP-complete

NP Complete and NP Hard Problems
Non-Deterministic Algorithm

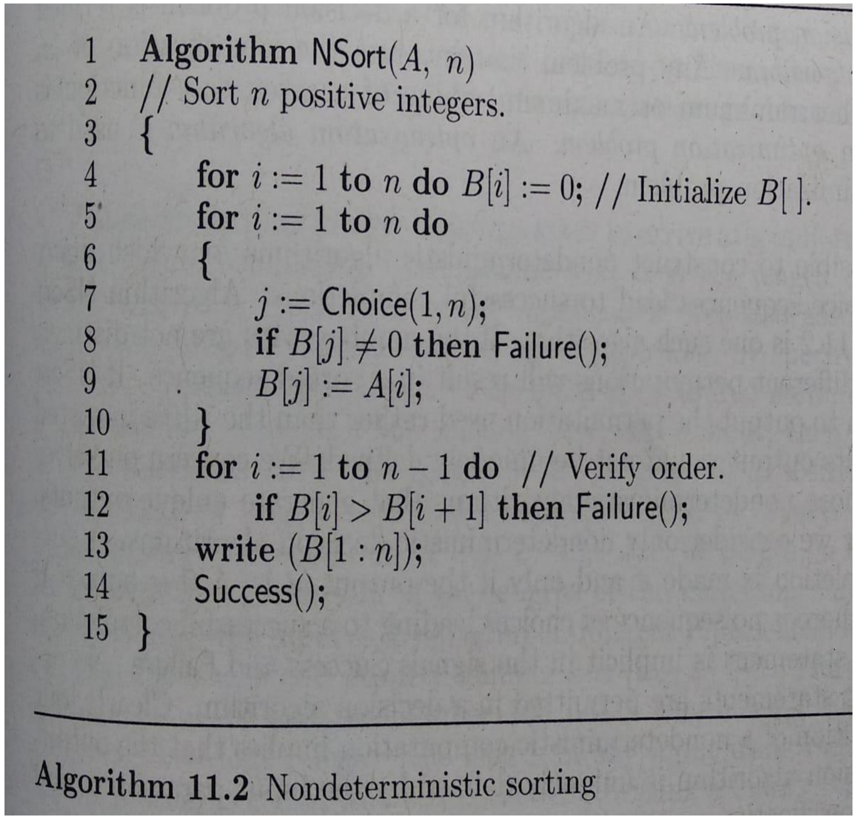
- Deterministic algorithm has the property that the result of every operation is uniquely defined
- Deterministic algorithms agrees with the way programs are executed on a computer
- Non-deterministic algorithm remove restriction on the outcome of every operation
- Also non-deterministic algorithm allow to contain operations whose outcomes are not uniquely defined but are limited to specified sets of possibilities
- Non-deterministic function introduces three new functions-
    1. Choice(S) arbitrarily chooses one of the element of set S
    2. Failure() signals an unsuccessful completion
    3. Success() signals a successful completion

Non-Deterministic Algorithms



```
j := Choice(1, n);
if A[j] = x then {write (j); Success();}
write (0); Failure();
```

Algorithm 11.1 Nondeterministic search



```
Algorithm NSort(A, n)
// Sort n positive integers.
{
    for i := 1 to n do B[i] := 0; // Initialize B[ ].
    for i := 1 to n do
    {
        j := Choice(1, n);
        if B[j] ≠ 0 then Failure();
        B[j] := A[i];
    }
    for i := 1 to n - 1 do  // Verify order.
        if B[i] > B[i + 1] then Failure();
    write (B[1 : n]);
    Success();
}
```

Algorithm 11.2 Nondeterministic sorting

The Classes NP-hard and NP-complete

- P is the set of all decision problems solvable by deterministic algorithms in polynomial time
- NP is the set of all decision problems solvable by non-deterministic algorithms in polynomial time

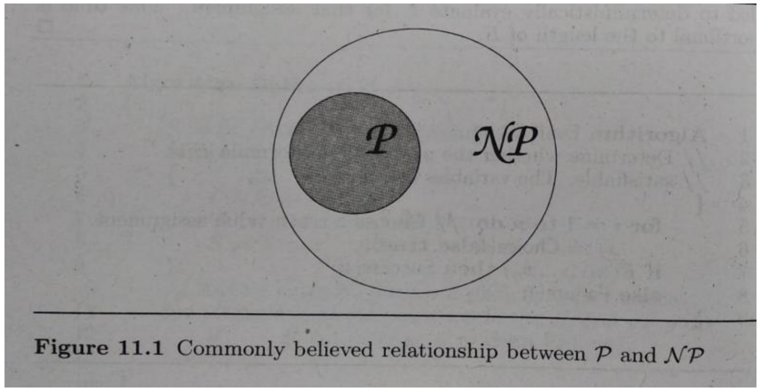


Figure 11.1 Commonly believed relationship between $\mathcal{P}$ and $\mathcal{NP}$

- From below figure, its clear that NP-hard problems that are not NP-complete
- Only a decision problem can be NP-complete
- However, an optimization problem may be NP-hard
- Optimization problems cannot be NP-complete whereas decision problems can
- There also exist NP-hard decision problems that are not NP-complete

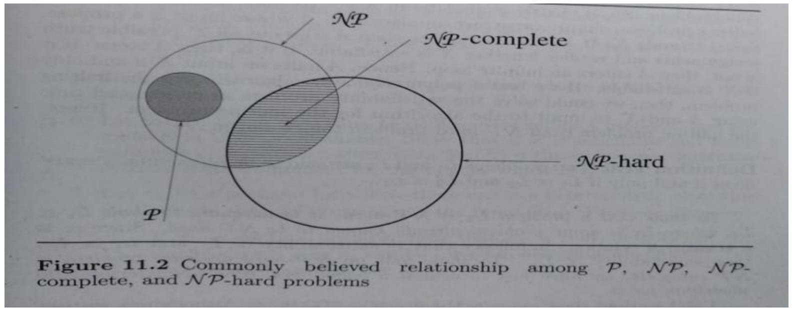


Figure 11.2 Commonly believed relationship among $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-complete, and $\mathcal{NP}$-hard problems