Computer Algorithms

Unit-4

Backtracking

The General Method

- Backtracking represents one of the most general techniques
- Problems which deal with searching for a set of solutions or
- which ask for an optimal solution satisfying some constraints
- can be solved using backtracking formulation
- Backtracking was first introduced by D. H. Lehmer in 1950

The General Method

- In many applications of backtrack method, the desire solution is expressible as an n-tuple (x1, x2, ...,xn), where xi 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(x1,x2,…xn)
- Sometimes it seeks all vectors that satisfy P
- E.g. Sorting – Criteria function P is inequality

The General Method

- Suppose mi is the size of set Si.
- Then there are m = m1,m2,...,mn n-tuples
- that are possible candidates for satisfying the function P.
- The brute force approach would be to form all these n-tuples,
- evaluate each one with P, and save those which yield the optimum

The General Method

- The backtrack algorithm has as its virtue
- the ability to yield the same answer with far fewer than m trials.
- Its basic idea is to build up the solution vector one component at a time
- and to use modified criterion functions
- Pi (x1, . . . , xi) (sometimes called bounding functions)

The General Method

- To test whether the vector being formed has any chance of success.
- The major advantage of this method is this:
- if it is realized that the partial vector (x1, x2, . . . , xi) can in no way lead to an optimal solution,
- then mi+1 ... mn possible test vectors can be ignored entirely.

The General Method

- Many of the problems we solve using backtracking require that
- all the solutions satisfy a complex set of constraints.
- For any problem these constraints can be divided into two categories:
- explicit and
- Implicit.

straints can be divided into two categories: *explicit* and *implicit*.

Definition 7.1 Explicit constraints are rules that restrict each x_i to take on values only from a given set. □

Common examples of explicit constraints are

$$\begin{array}{rcrcl} x_i \geq 0 & \text{or} & S_i & = & \{\text{all nonnegative real numbers}\} \\ x_i = 0 \ \text{ or } \ 1 & \text{or} & S_i & = & \{0, 1\} \\ l_i \leq x_i \leq u_i & \text{or} & S_i & = & \{a : l_i \leq a \leq u_i\} \end{array}$$

The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible *solution space* for I.

Definition 7.2 The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the x_i must relate to each other. □

THE 8-QUEENS PROBLEM

- Tackle the 8-queens problem via a backtracking solution.
- Generalize the problem and consider an n x n chessboard
- and try to find all ways to place n nonattacking queens.
- Let (x1,…, Xn) represents a solution in which xi is the column of the ith row where ith queen is placed

THE 8-QUEENS PROBLEM

- The 8-queens problem can be represented as 8-tuples (X1 . . . , X*8),*
- *Si* = {I, 2, 3, 4, 5, 6, 7, 8}, 1<=i<=8.
- The solution space consists of 8^8 8-tuples.
- The implicit constraints for this problem are that no two *Xi's* can be the same
- (i.e. all queens must be on different columns)
- and no two queens can be on the same diagonal.

THE 8-QUEENS PROBLEM

- The first of these two constraints implies that
- all solutions are permutations of the 8-tuple (1, 2, 3, 4, 5, 6, 7, 8).
- This realization reduces the size of the solution space from 8^8 tuples to 8! tuples.

THE 8-QUEENS PROBLEM

- The xi's will all be distinct
- since no two queens can be placed in the same column.
- Now how do we test whether two queens are on the same diagonal?
- Imagine the chessboard squares being numbered as
- the indices of the two-dimensional array a[1 : n, 1 : n],

THE 8-QUEENS PROBLEM

- Then observe that
- every element on the same diagonal that runs from the upper left to the lower right
- has the same row-column value.
- For example, in Figure consider the queen at a[4,2].
- The squares that are diagonal to this queen a[3, 1], a[5,3], a[6,4], a[7,5], and a[8,6].
- All these squares have row-column value of 2

THE 8-QUEENS PROBLEM

column →

row ↓	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4		Q						
5							Q	
6	Q							
7			Q					
8					Q			

THE 8-QUEENS PROBLEM

- Also, every element on the same diagonal
- that goes from the upper right to the lower left has the same row+column value.
- Suppose two queens are placed at positions (i, j) and (k, l).
- Then by the above they are on the same diagonal only if

THE 8-QUEENS PROBLEM

- $i-j=k-l$
- Or $i+j=k+l$
- The first equation implies
- $j-l=i-k$
- The second implies
- $j-l=k-i$
- Therefore two queens lie on the same diagonal if and only if $|j - l| = |i - k|$

THE 8-QUEENS PROBLEM

- Algorithm Place(k, l)
- //Return True if queen can be placed in k^{th} row and l^{th} column.
- //X[ ] is global array whose (k-1) values have been set
- //Abs(r) returns absolute value or r
- {
-   For i=1 to k-1 do
-       if(x[i]=l) //Two queens in same column
-         OR
-       Abs(x[i]-l)=Abs(i-k) //Two queens on same diagonal
-         Then return false;
-   Return True;
- }

THE 8-QUEENS PROBLEM

- Place(k, l) returns a boolean value that is **true** if the kth queen can be placed in column l.
- It tests both whether l is distinct from all previous values x[l],. . . , x[k - 1]
- and whether there is no other queen on the same diagonal.
- Its computing time is O(k - 1).

THE 8-QUEENS PROBLEM

- Algorithm Nqueens(k, n)
- //Using backtracking, this procedure prints all possible placements of n queens on an n x 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);
- }
- }

THE 8-QUEENS PROBLEM

- At this point it is clear, how effective function NQueens is over the brute force approach.
- For an 8 x 8 chessboard there are 16777216 possible ways to place 8 pieces,
- However, by allowing only placements of queens on distinct rows and columns,
- we require the examination of at most 8!, or only 40,320 8-tuples.

THE 8-QUEENS PROBLEM

- Permutation Tree is used to show the solution space consisting of n! permutations of n-tuples (1,2,…n)
- The edges are labeled by possible values of Xi
- Edges from level 1 to level 2 nodes specify the values for x1
- Leftmost subtree contains all solutions with x1=1, x2=2 and so on
- Edges from level i to i+1 are lebeled with values of Xi

THE 8-QUEENS PROBLEM

- The solution space is defined by all paths from the root node to a leaf node
- There are 4!=24 leaf nodes

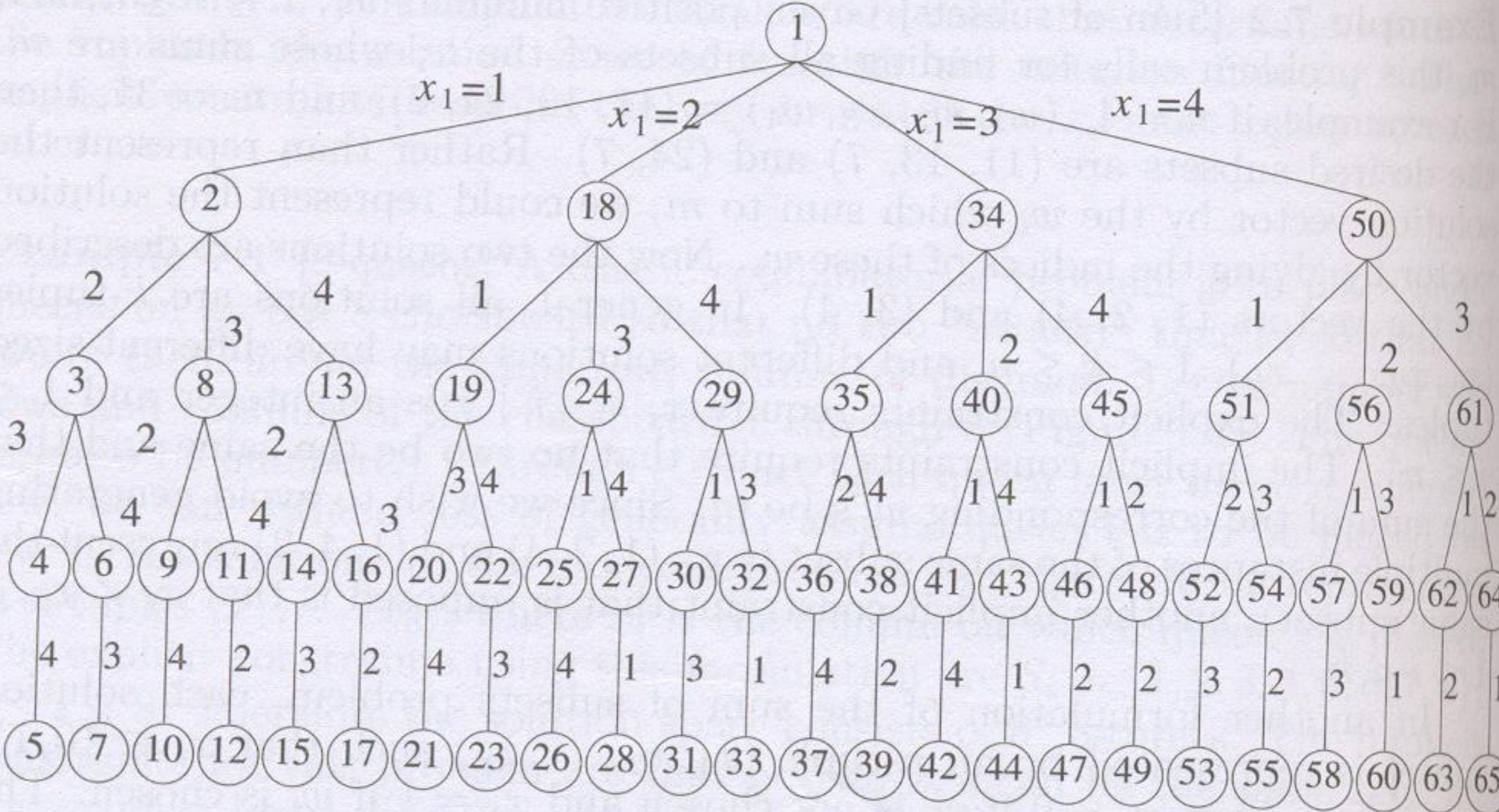
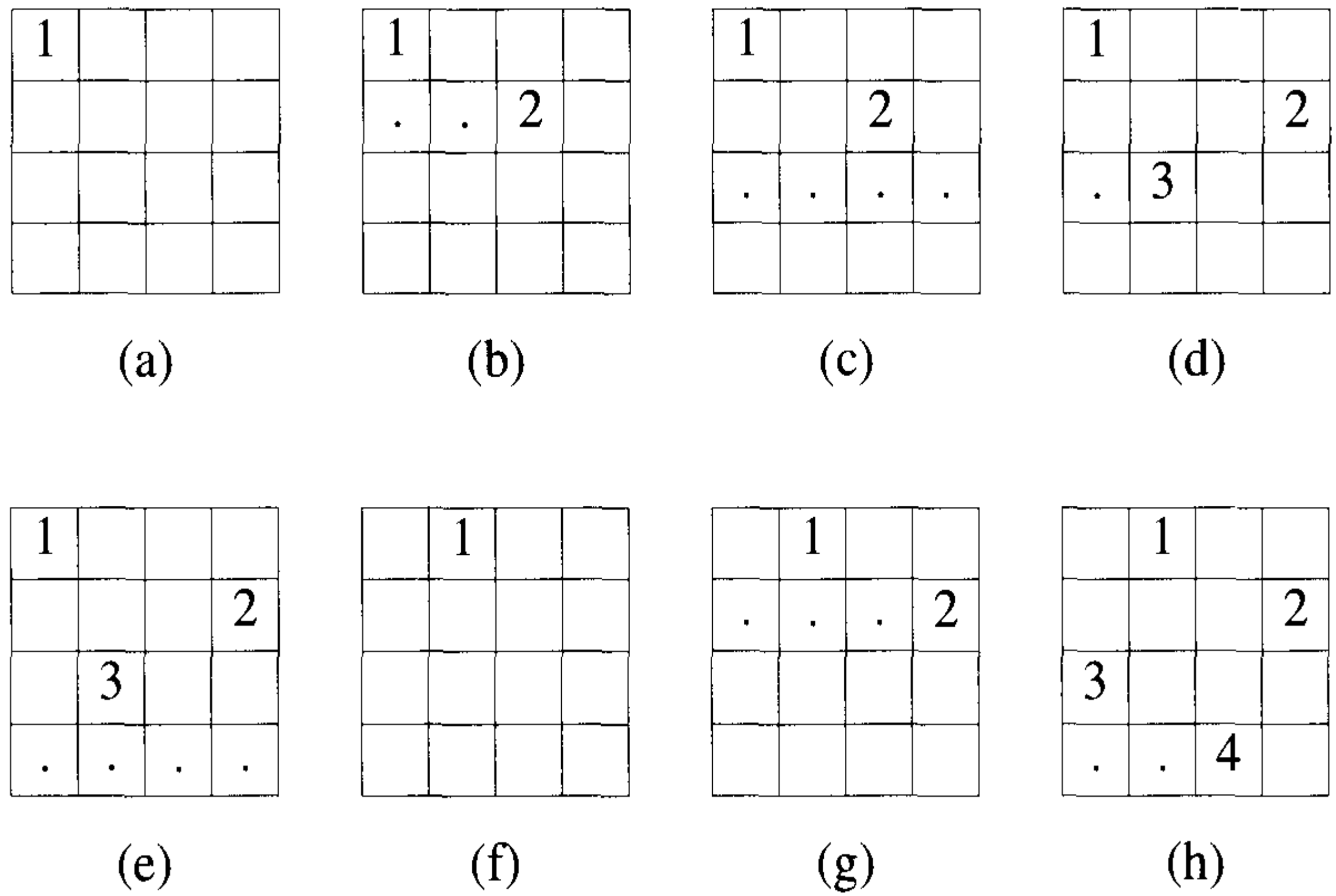


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.



Example of a backtrack solution to the 4-queens problem

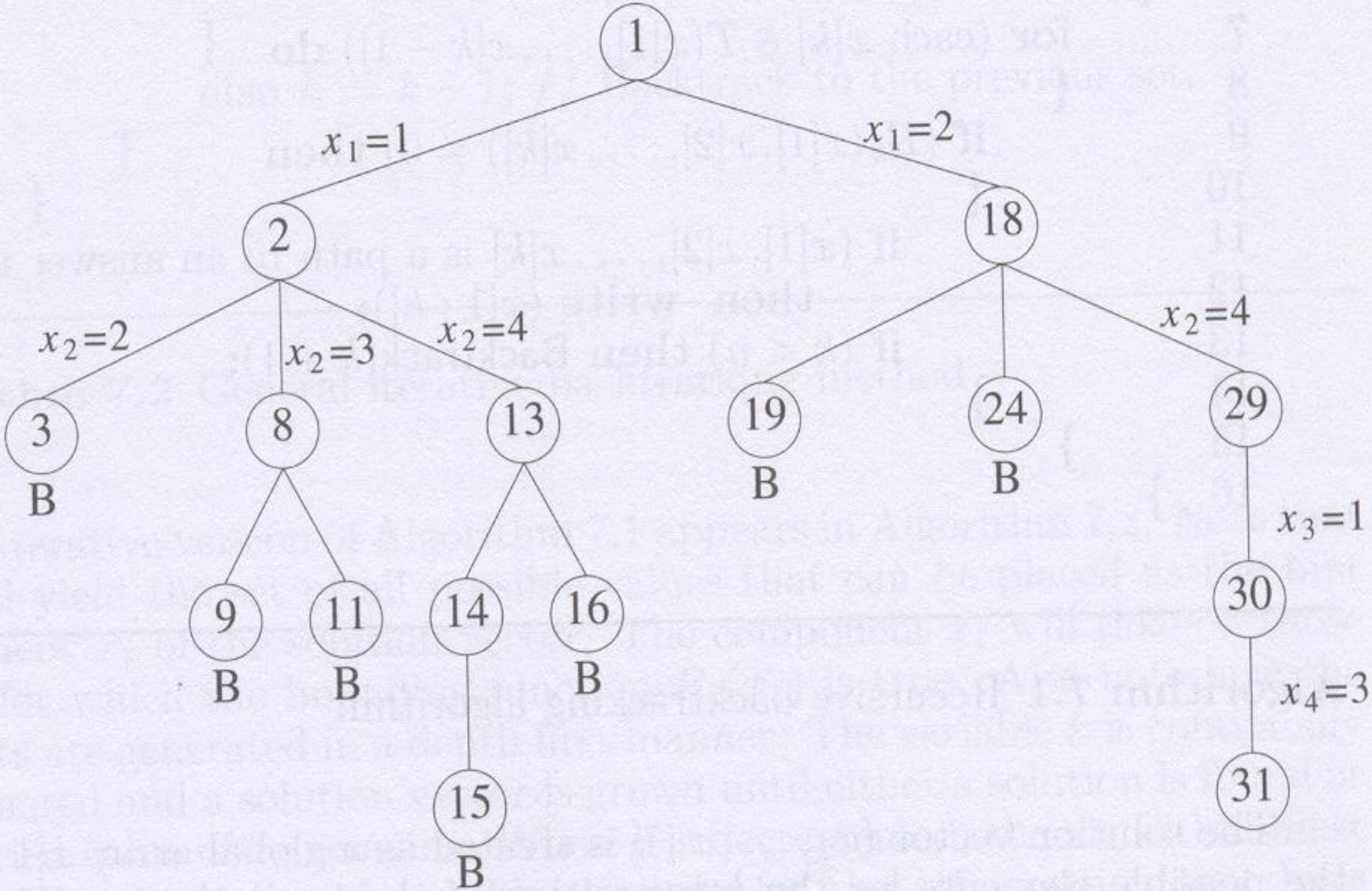


Figure 7.6 Portion of the tree of Figure 7.2 that is generated during backtracking

THE 8-QUEENS PROBLEM

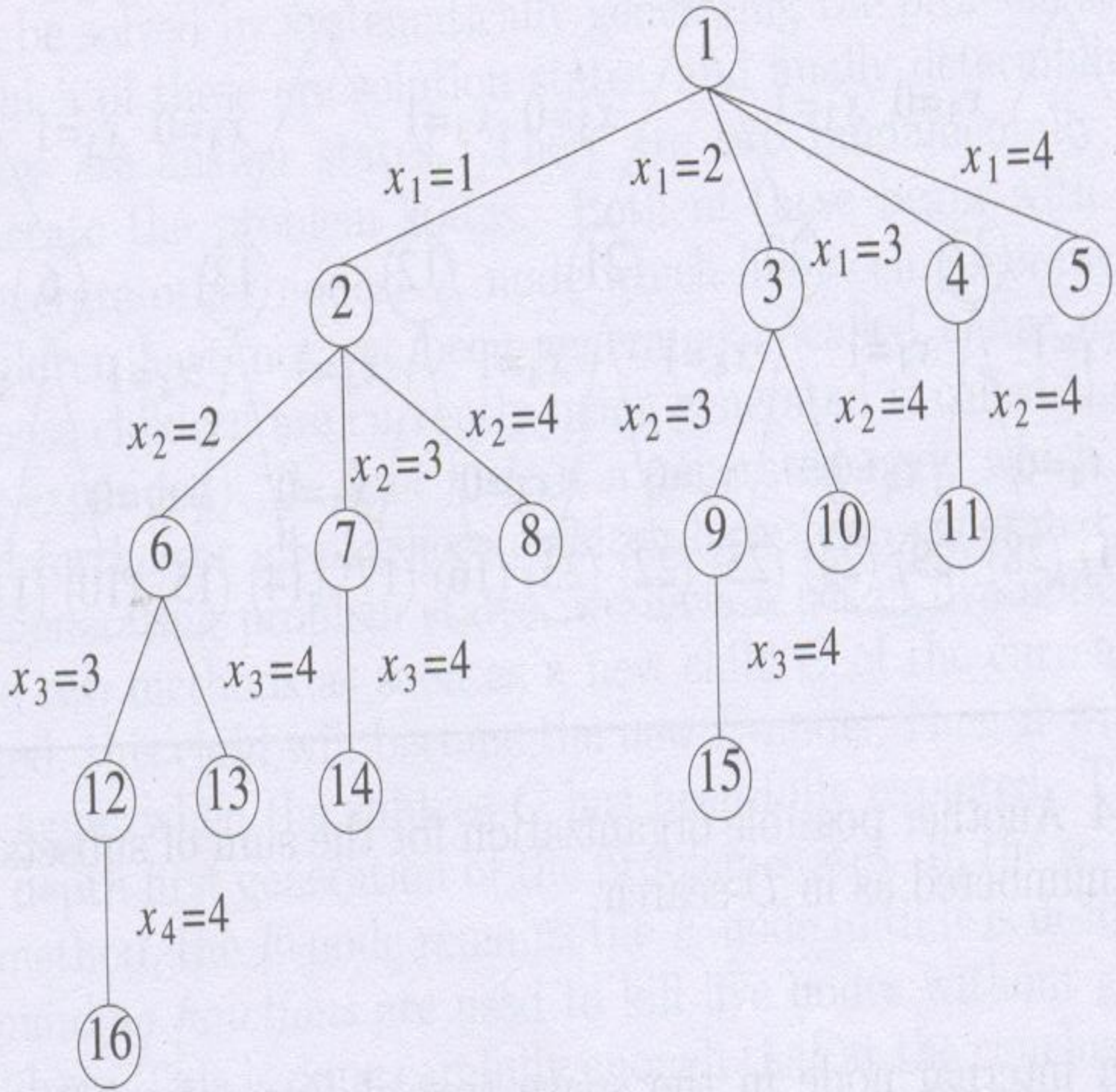
- As required, the placement of each queen on the chessboard was chosen randomly.
- With each choice kept track of the number of columns a queen could legitimately be placed on.
- These numbers are listed in the vector
- The estimated number of unbounded nodes is only about 2.34% of the total number of nodes in the 8-queens state space tree.

SUM OF SUBSETS

- Suppose we are given *n* distinct positive numbers (usually called weights)
- we desire to find all combinations of these numbers whose sums are m.
- This is called the sum of subsets problem.
- consider a backtracking solution using the fixed tuple size strategy.

SUM OF SUBSETS

- Given positive numbers Wi, 1<=i<=n and m,
- this problem calls for finding all subsets of the Wi whose sums are m.
- For example, if n = 4, (WI, W2, W3, W4) = (7, 11, 13, 24), and m = 31,
- then the desired subsets are (11, 13, 7) and (24, 7). Rather than represent the solution vector by the Wi which sum to m,
- we could represent the solution vector by giving the indices of these Wi. Now the two solutions are described by the vectors (1, 2, 3) and (1, 4).


1
$x_1=1$
$x_1=2$
$x_1=3$
$x_1=4$
2
3
4
5
$x_2=2$
$x_2=3$
$x_2=4$
$x_2=3$
$x_2=4$
$x_2=4$
6
7
8
9
10
11
$x_3=3$
$x_3=4$
$x_3=4$
$x_3=4$
12
13
14
15
$x_4=4$
16

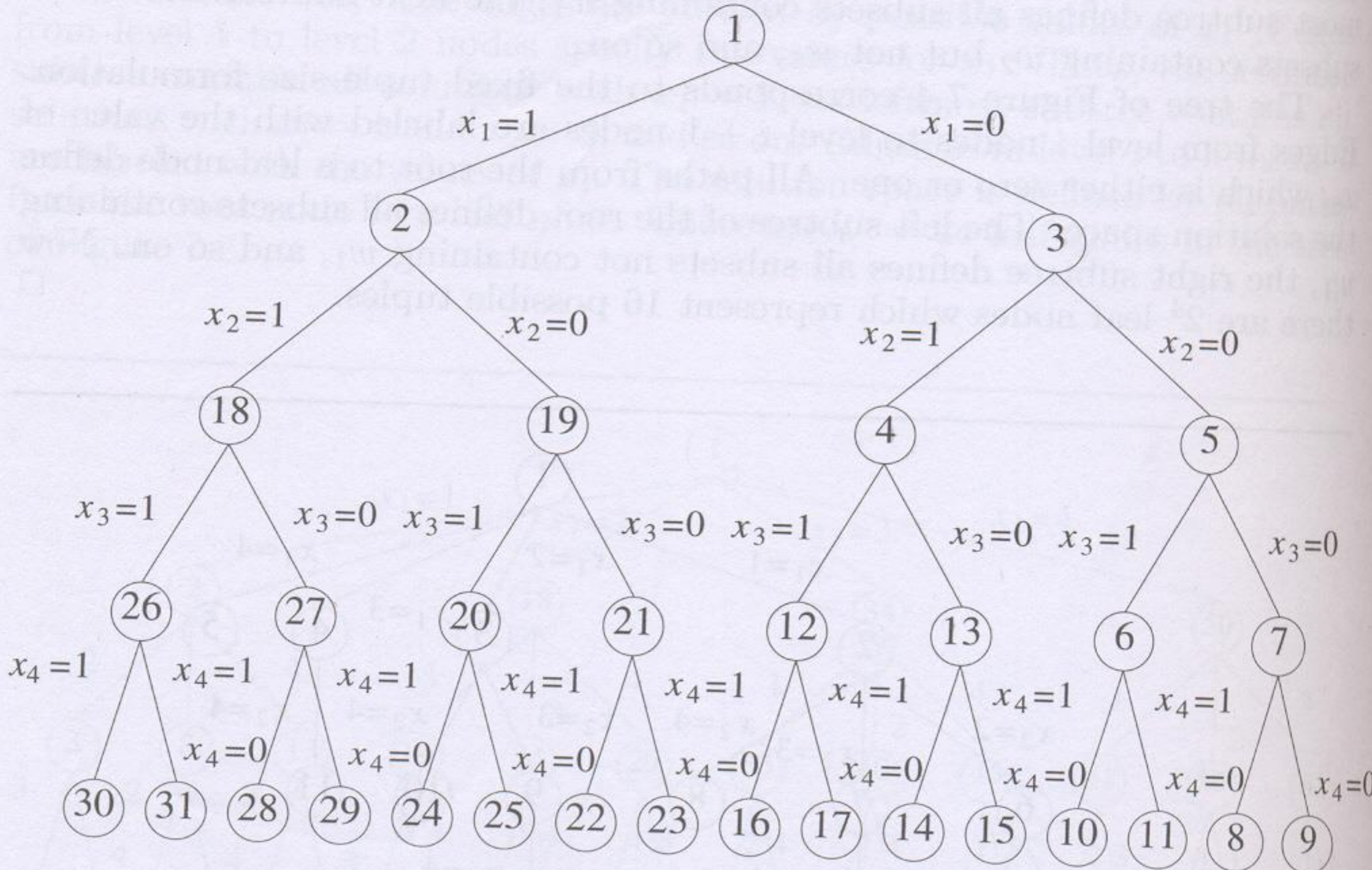


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in D-search.

SUM OF SUBSETS

- In this case the element *Xi* of the solution vector is either one or zero
- depending on whether the weight *Wi* is included or not.

- S= sum upto k-1 element
- K=current element
- R=sum from k to n

```
Algorithm SumOfSub(s, k, r)
// Find all subsets of w[1 : n] that sum to m. The values of x[j],
// 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
// and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
// It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
{
    // Generate left child. Note: s + w[k] ≤ 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 ≤ j ≤ n.
    else  if (s + w[k] + w[k + 1] ≤ m)
          then SumOfSub(s + w[k], k + 1, r - w[k]);
    // Generate right child and evaluate B_k.
    if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
    {
        x[k] := 0;
        SumOfSub(s, k + 1, r - w[k]);
    }
}
```

Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

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). □

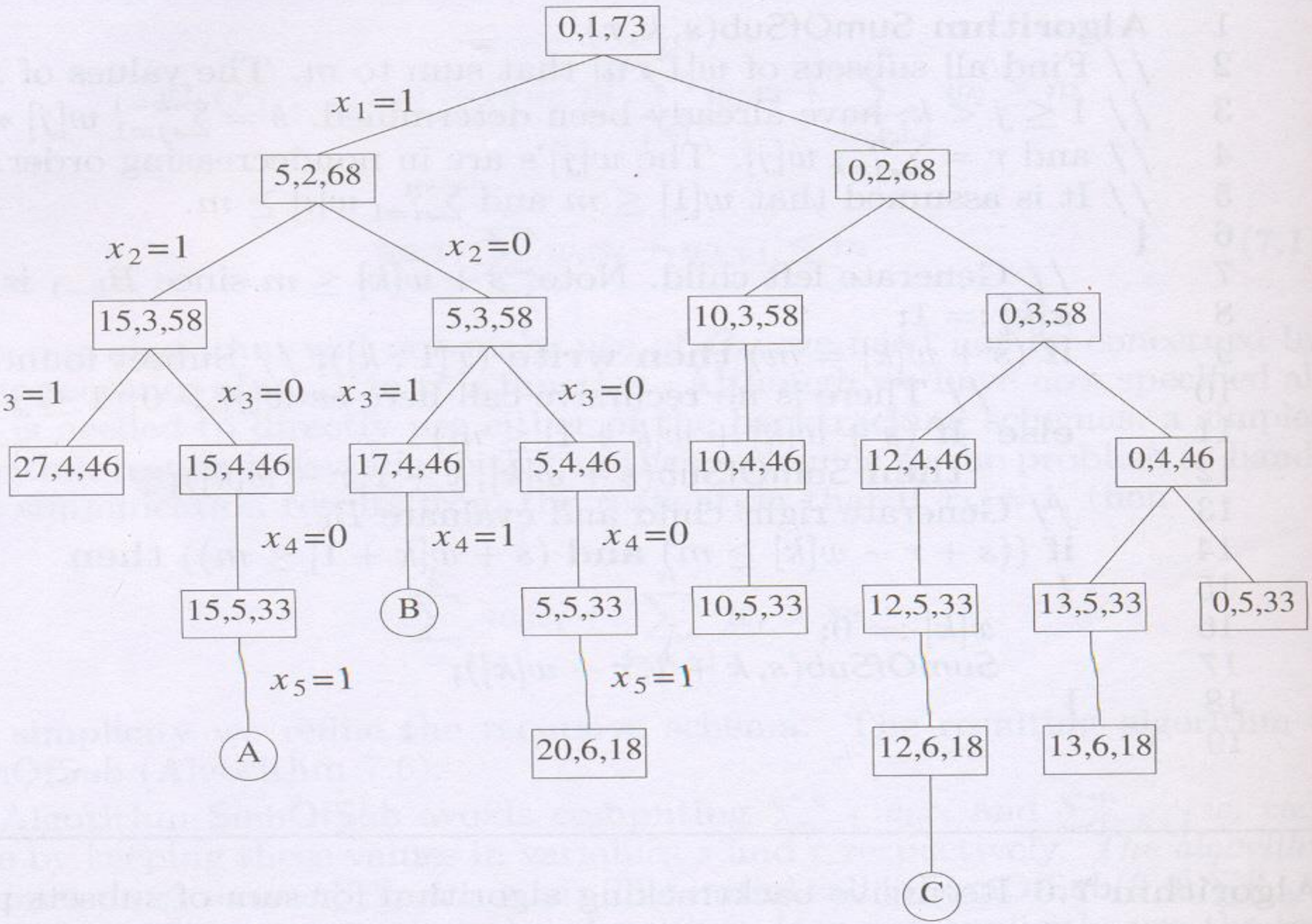


Figure 7.10 Portion of state space tree generated by SumOfSub

GRAPH COLORING

- Let G be a graph and m be a given positive integer.
- We want to discover whether the nodes of 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*

GRAPH COLORING

- if d is the degree of the given 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 coloreds
- This integer is referred to as the chromatic number of the graph.

GRAPH COLORING

- For example, the graph of Figure can be colored with three colors 1,2, and 3.
- The color of each node is indicated next to it.
- It can also be seen that three colors are needed to color this graph
- and hence this graph's chromatic number is 3.

GRAPH COLORING

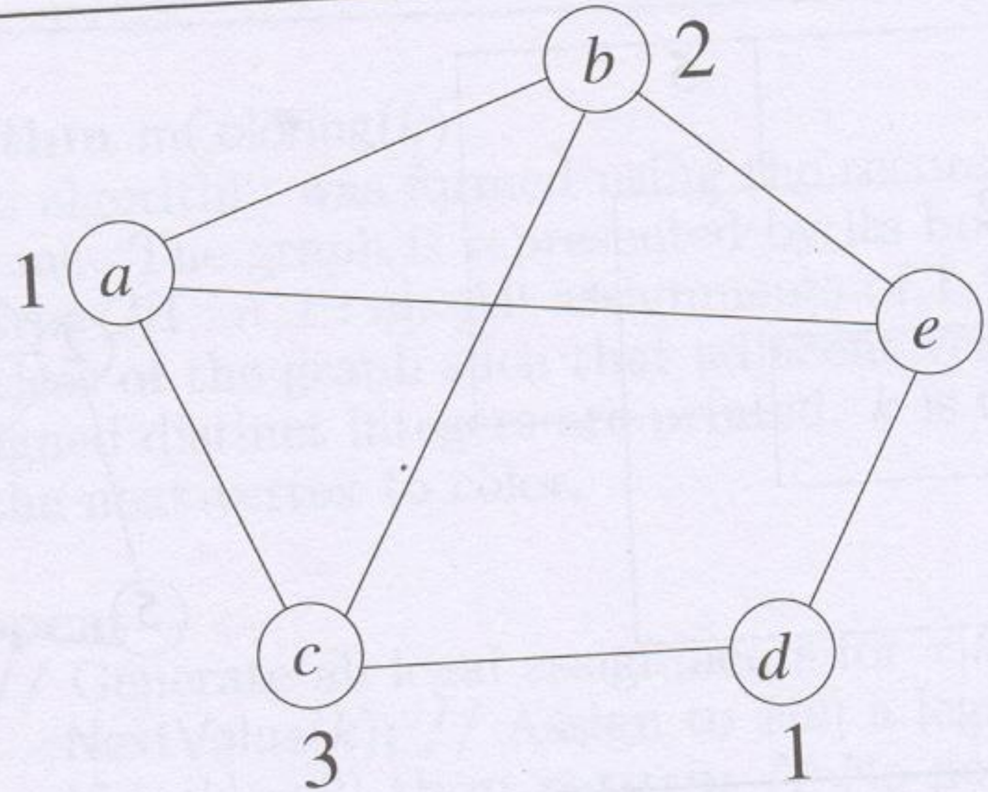


Figure 7.11 An example graph and its coloring

GRAPH COLORING

- A graph is said to be *planar* *iff*
- it can be drawn in a plane in such a way that no two edges cross each other.
- A famous special case of the m-colorability decision problem is the 4-color problem for planar graphs.
- This problem asks the following question

GRAPH COLORING

- Given any map, can the regions be colored in such a way that
- no two adjacent regions have the same color yet only four colors are needed?
- This turns out to be a problem for which graphs are very useful,
- because a map can easily be transformed into a graph.

GRAPH COLORING

- Each region of the map becomes a node,
- and if two regions are adjacent, then the corresponding nodes are joined by an edge.
- Figure shows a map with five regions and its corresponding graph.
- This map requires four colors.
- For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found.

GRAPH COLORING

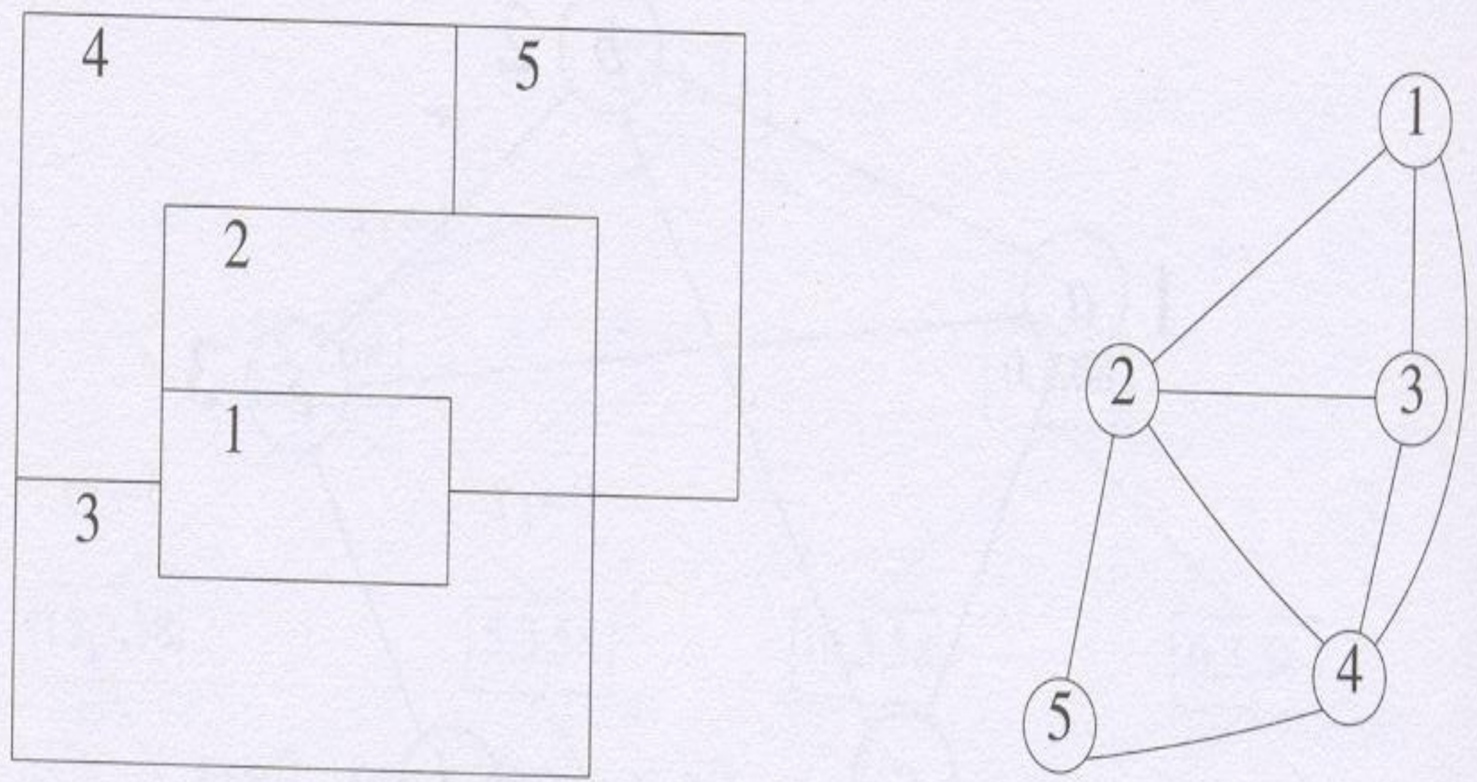


Figure 7.12 A map and its planar graph representation

GRAPH COLORING

- We are interested in determining
- all the different ways in which a given graph
- can be colored using at most m colors.
- 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

GRAPH COLORING

- The colors are represented by the integers 1,2,... , m
- and the solutions are given by the n-tuple (X1,…, xn),
- where Xi is the color of node i
- Using the recursive backtracking formulation the resulting algorithm is mColoring

```
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, ..., 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], ..., 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] ≠ 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

GRAPH COLORING

- 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 Xi, 1 <= i <= n.
- Nodes at level n + 1 are leaf nodes.
- Figure shows the state space tree when n =3 and m = 3.
- Computing Times is=$O(n\ m^n)$

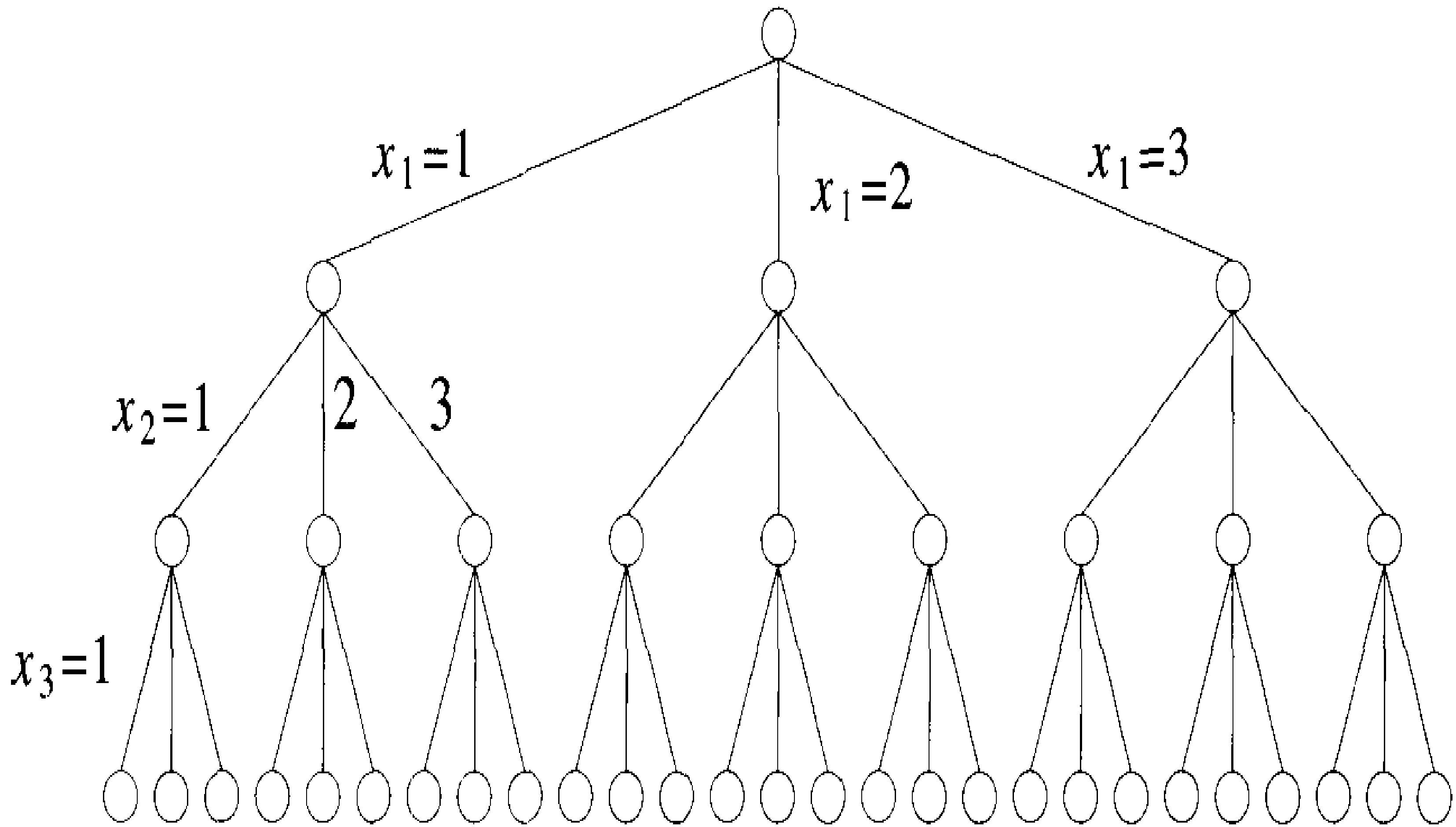


Figure 7.13 State space tree for mColoring when n =3 and $m = 3$

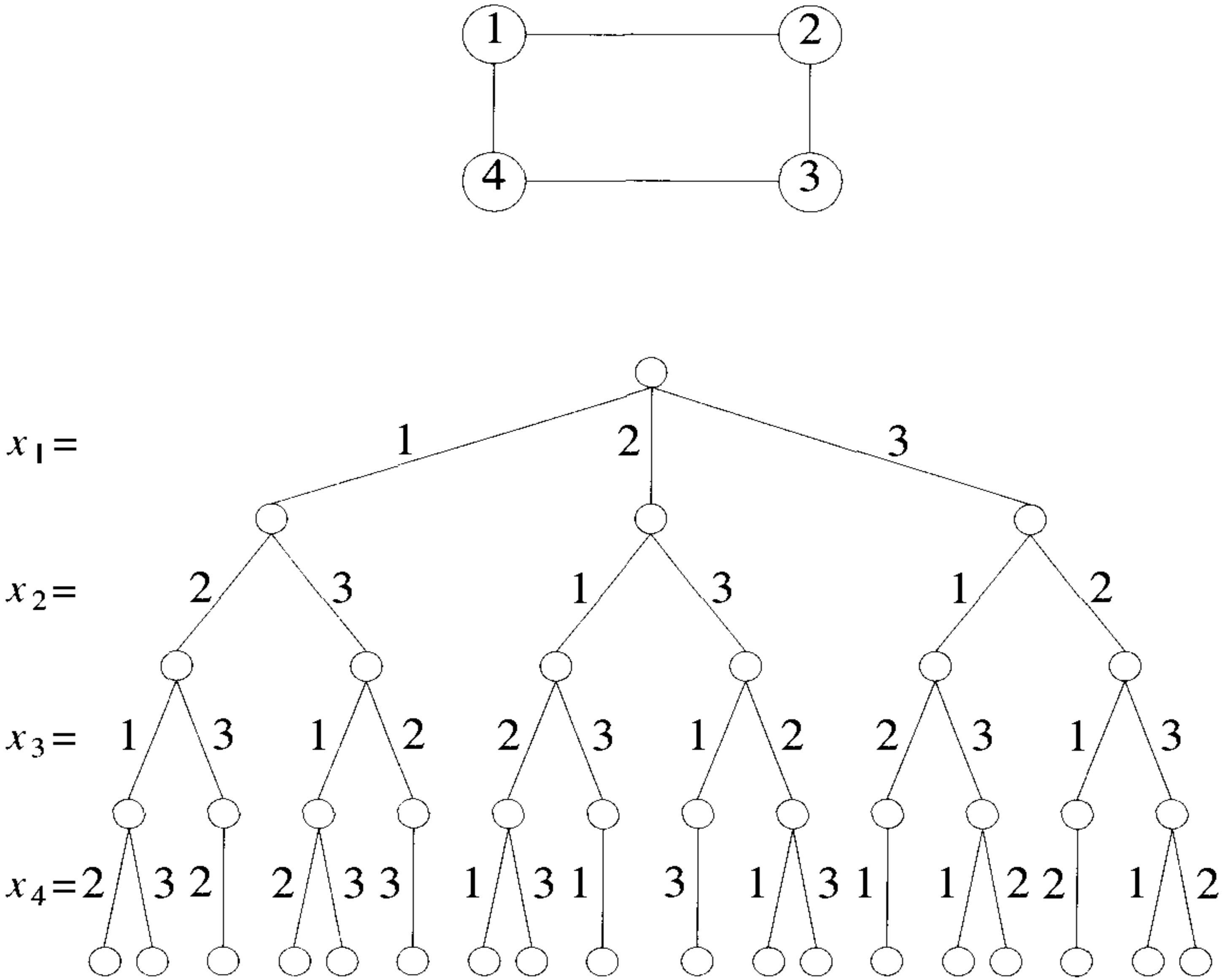


Figure 7.14 A 4-node graph and all possible 3-colorings

HAMILTONIAN CYCLES

- Let G = (V, E) be a connected graph with n vertices.
- A Hamiltonian cycle (suggested by Sir William Hamilton) is a
- round-trip path along n edges of G
- that visits every vertex once and returns to its starting position.

HAMILTONIAN CYCLES

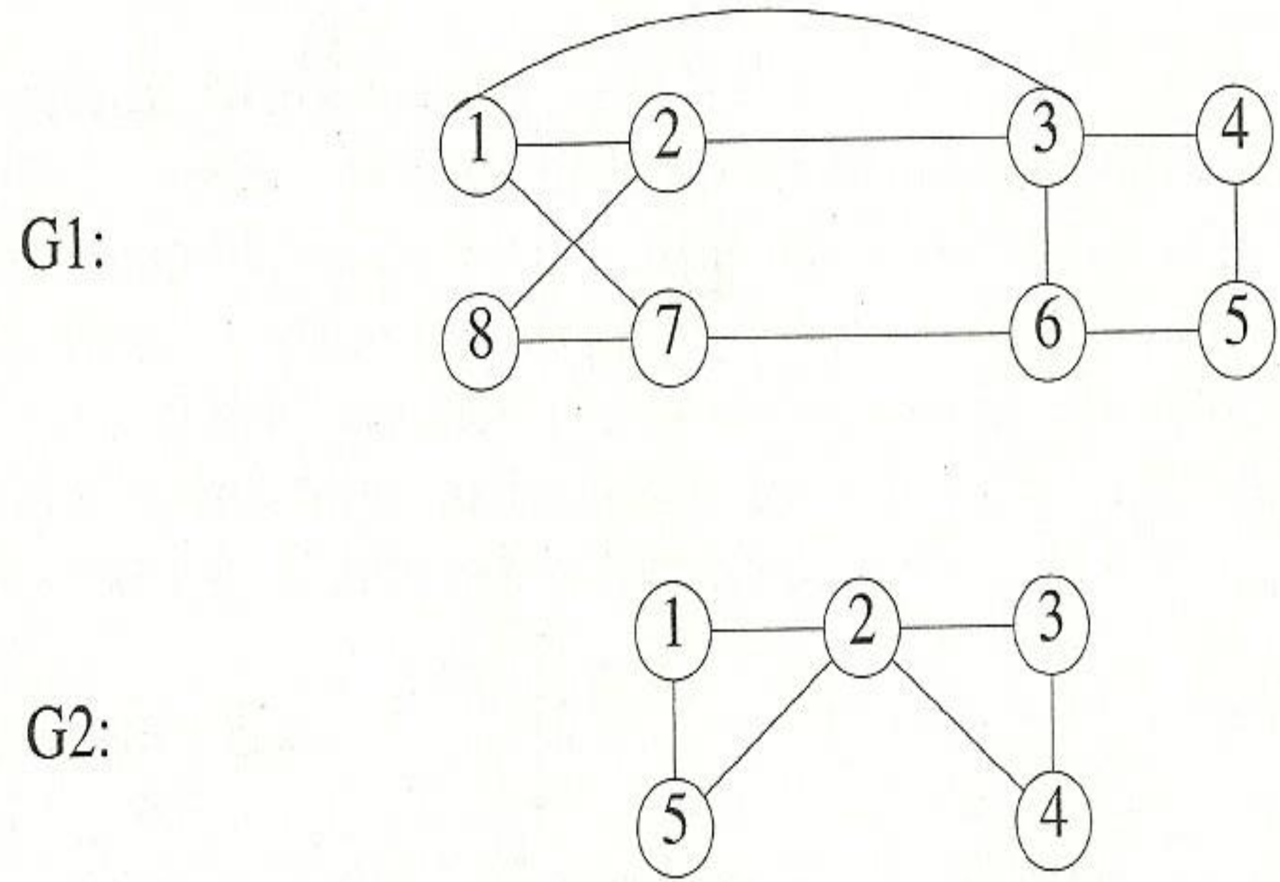
- In other words
- if a Hamiltonian cycle begins at some vertex

  $v_1 \in G$

- 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+i}) are in E, $1 < i < n$,
- and the vi are distinct except for v_1 and v_{n+1}
- which are equal.



- The graph G1 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1.
- The graph G2 contains no Hamiltonian cycle.

HAMILTONIAN CYCLES

- There is no known easy way
- to determine whether a given graph contains a Hamiltonian cycle.
- backtracking algorithm can find all the Hamiltonian cycles in a graph.
- The graph may be directed or undirected.
- Only distinct cycles are output.

HAMILTONIAN CYCLES

- The backtracking solution vector (x1,….,xn)
- is defined so that xi represent the ith visited vertex of the proposed cycle.
- Now all we need to do is determine how to compute the set of possible vertices
- for xk if x1,..,xk-1 have already been chosen.

HAMILTONIAN CYCLES

- To avoid printing the same cycle n times,
- we require that x1=1. If $1 < k < n$,
- Then xk can be any vertex v that is distinct from x1,x2,... ,xk-1
- and v is connected by an edge to xk-1.
- The vertex xn can only be the one remaining vertex
- and it must be connected to both xn-1 and x1

HAMILTONIAN CYCLESs

- function NextValue(k) determines a possible next vertex for the proposed cycle.
- Using NextValue particularize the recursive backtracking schema
- to find all Hamiltonian cycles
- 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]] ≠ 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]] ≠ 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

KNAPSACK PROBLEM

- Reconsider 0/1 knapsack problem solved using dynamic programming approach
- Given n positive weights wi,
- n positive profits pi, and
- positive number m that is a knapsack capacity
- This problem calls for choosing a subset of weights such that

$$\sum_{1 \le i \le n} w_i x_i \le m \quad \text{and} \quad \sum_{1 \le i \le n} p_i x_i \text{ is maximized}$$

KNAPSACK PROBLEM

- The xi’s constitute a zero-one-valued vector.
- The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the xi’s.
- Thus the solution space is the same as that for the sum of subsets problem.
- Two possible tree organizations are possible.

KNAPSACK PROBLEM

Generate the sets S^i , $0 \le i \le 4$, when

$(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$ and

$(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$

$m=25$

KNAPSACK PROBLEM

```
Algorithm Bound(cp, cw, k)
// cp is the current profit total, cw is the current
// weight total; k is the index of the last removed
// item; and m is the knapsack size.
{
    b := cp; c := cw;
    for i := k + 1 to n do
    {
        c := c + w[i];
        if (c < m) then b := b + p[i];
        else return b + (1 − (c − m)/w[i]) * p[i];
    }
    return b;
}
```


Algorithm A bounding function

```
Algorithm BKnap(k, cp, cw)
// m is the size of the knapsack; n is the number of weights
// and profits. w[ ] and p[ ] are the weights and profits.
// p[i]/w[i] ≥ p[i + 1]/w[i + 1]. fw is the final weight of
// knapsack; fp is the final maximum profit. x[k] = 0 if w[k]
// is not in the knapsack; else x[k] = 1.
{
    // Generate left child.
    if (cw + w[k] ≤ m) then
    {
        y[k] := 1;
        if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
        if ((cp + p[k] > fp) and (k = n)) then
        {
            fp := cp + p[k]; fw := cw + w[k];
            for j := 1 to k do x[j] := y[j];
        }
    }
    // Generate right child.
    if (Bound(cp, cw, k) ≥ fp) then
    {
        y[k] := 0; if (k < n) then BKnap(k + 1, cp, cw);
        if ((cp > fp) and (k = n)) then
        {
            fp := cp; fw := cw;
            for j := 1 to k do x[j] := y[j];
        }
    }
}
```


Algorithm   Backtracking solution to the 0/1 knapsack problem

KNAPSACK PROBLEM

- One corresponds to the fixed tuple size formulation

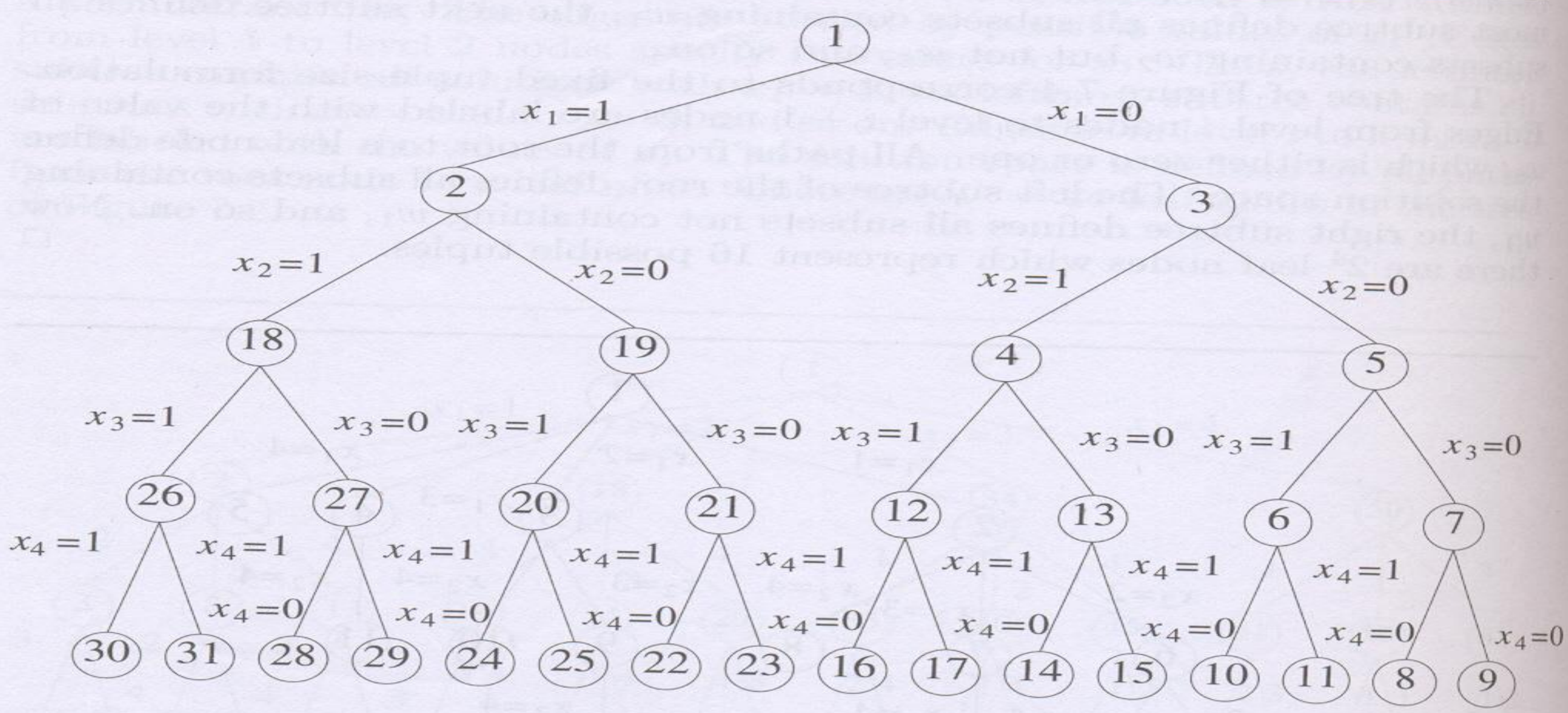
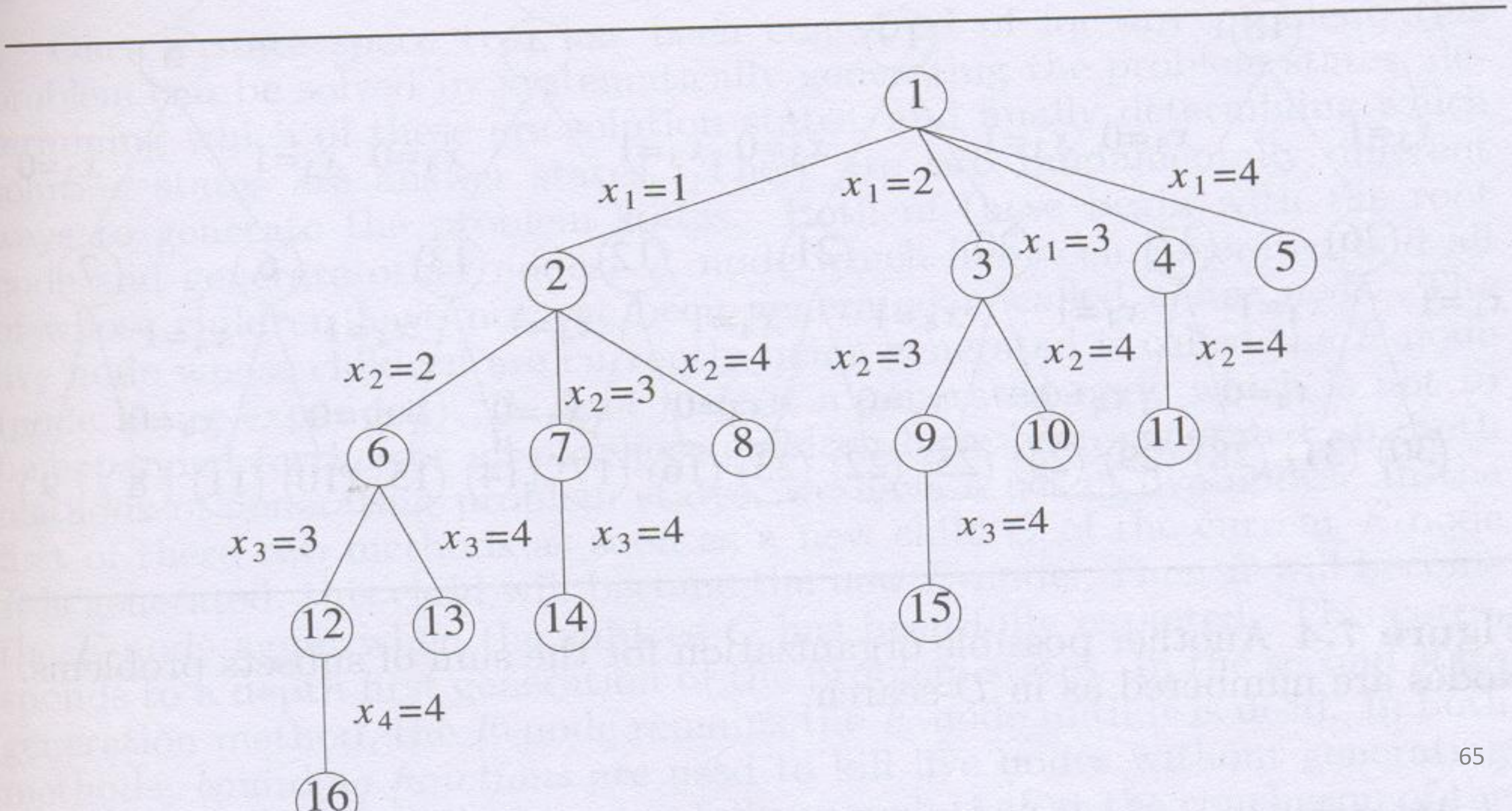


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in D-search.

KNAPSACK PROBLEM

- and the other to the variable tuple size formulation



- Backtracking algorithms for the knapsack problem can be arrived at using either of these two state space trees.
- Regardless of which is used,
- bounding functions are needed to help kill some live nodes without expanding them.

Terminology

- Bounding Function – modified Criteria Function
- E.g. Sorting – comparison
- Problem state is each node in the depth-first search tree
- State space is the set of all paths from root node to other nodes

Terminology

- Solution states are the problem states s for which the path from the root node to s defines a tuple in the solution space
- In variable tuple size formulation tree, all nodes are solution states
- In fixed tuple size formulation tree, only the leaf nodes are solution states
- Partitioned into disjoint sub-solution spaces at each internal node

- Answer states are those solution states s for which the path from root node to s defines a tuple that is a member of the set of solutions
- – These states satisfy implicit constraints
- State space tree is the tree organization of the solution space

- Static trees are ones for which tree organizations are independent of the problem instance being solved
- – Fixed tuple size formulation
- – Tree organization is independent of the problem instance being solved

- Dynamic trees are ones for which organization is dependent on problem instance
- Live node is a generated node for which all of the children have not been generated yet
- E-node is a live node whose children are currently being generated or explored

- Dead node is a generated node that is not to be expanded any further
- – All the children of a dead node are already generated
- – Live nodes are killed using a bounding function to make them dead nodes
- Backtracking is depth-first node generation with bounding functions

Thank You