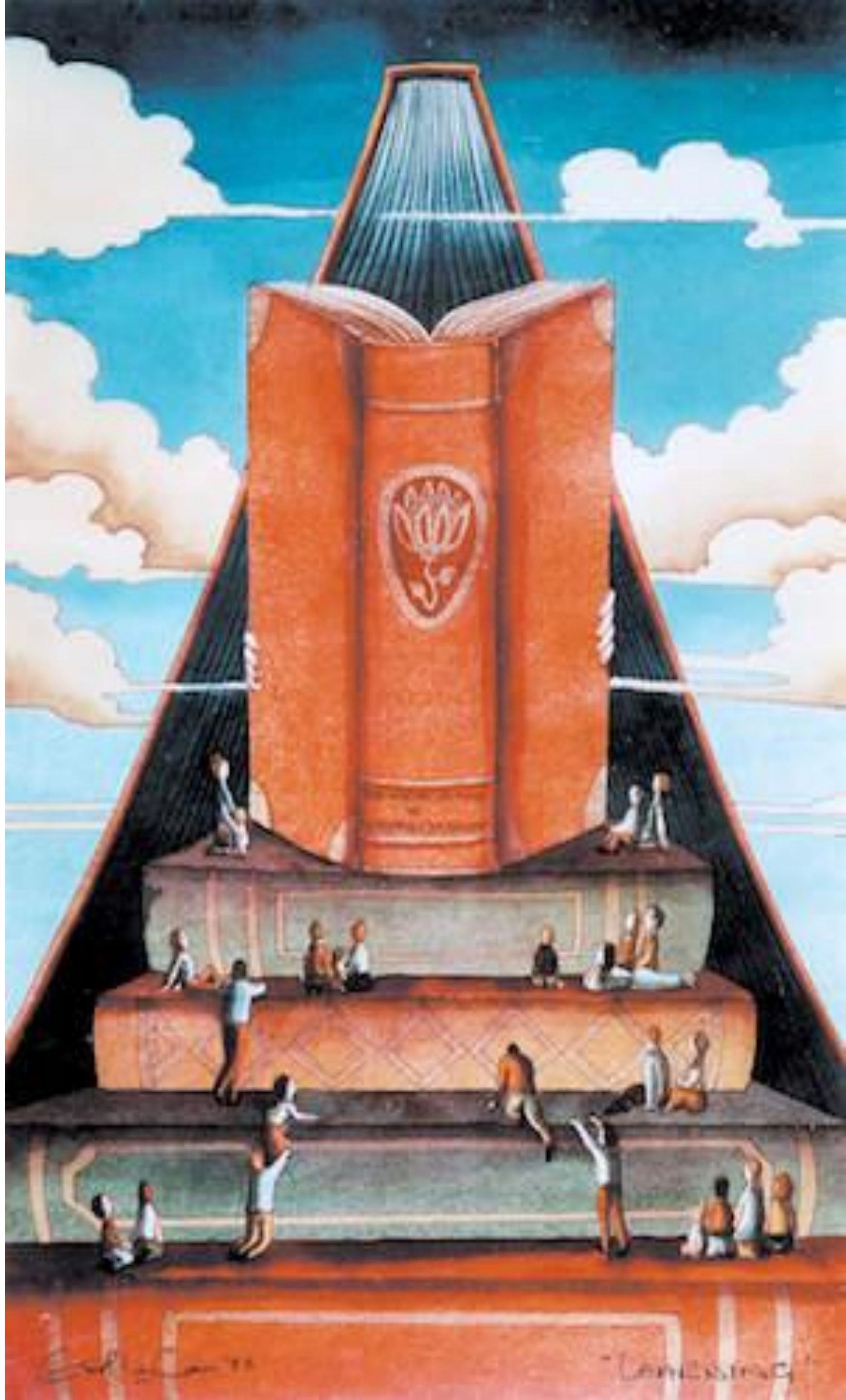
BACKTRACKING



Outlines

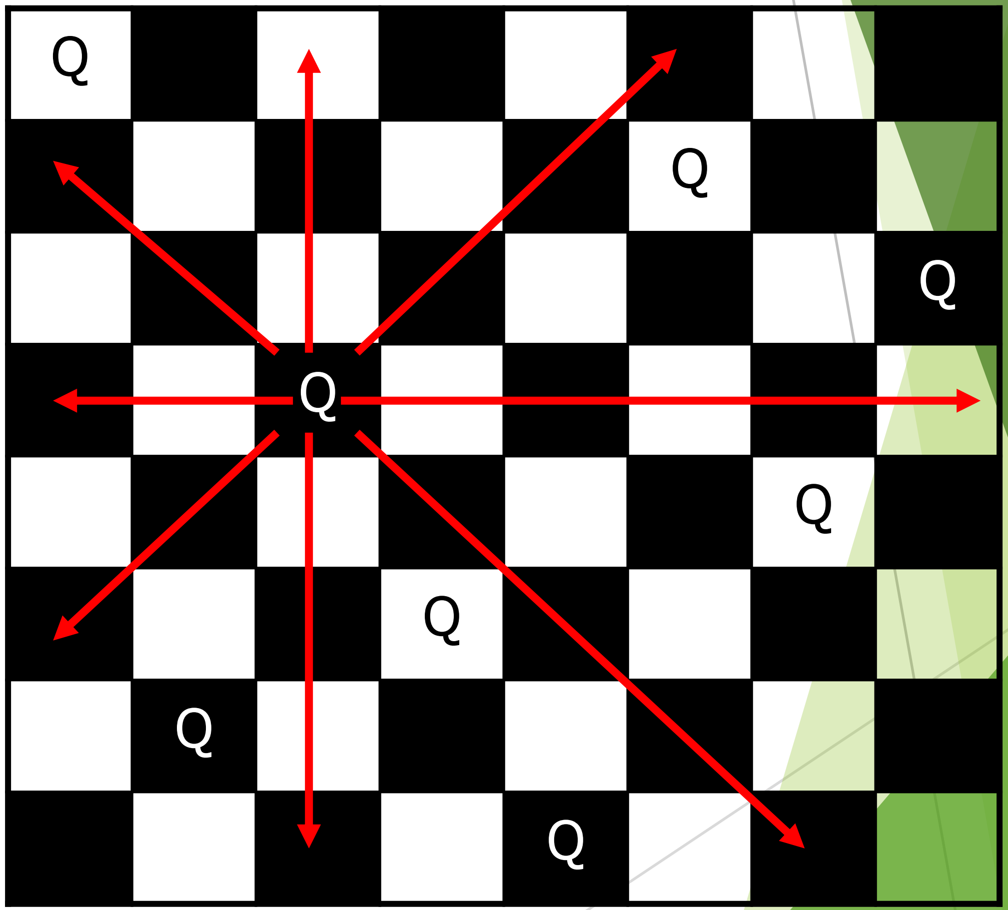
- Introduction
- 8-Queen Problem
- Sum of Subsets Problem
- Graph Coloring Problem
- Hamiltonian Cycle

Introduction

- Suppose you have to make a series of decisions, among various choices, where
  - You don’t have enough information to know what to choose
  - Each decision leads to a new set of choices
  - Some sequence of choices (possibly more than one) may be a solution to your problem
- Backtracking is a methodical way of trying out various sequences of decisions, until you find one that “works”

The 8-Queens Problem

- Place eight queens on the chessboard so that no queen can attack any other queen
  - What are the "choices"?
  - How do we "make" or "un-make" a choice?
  - How do we know when to stop?



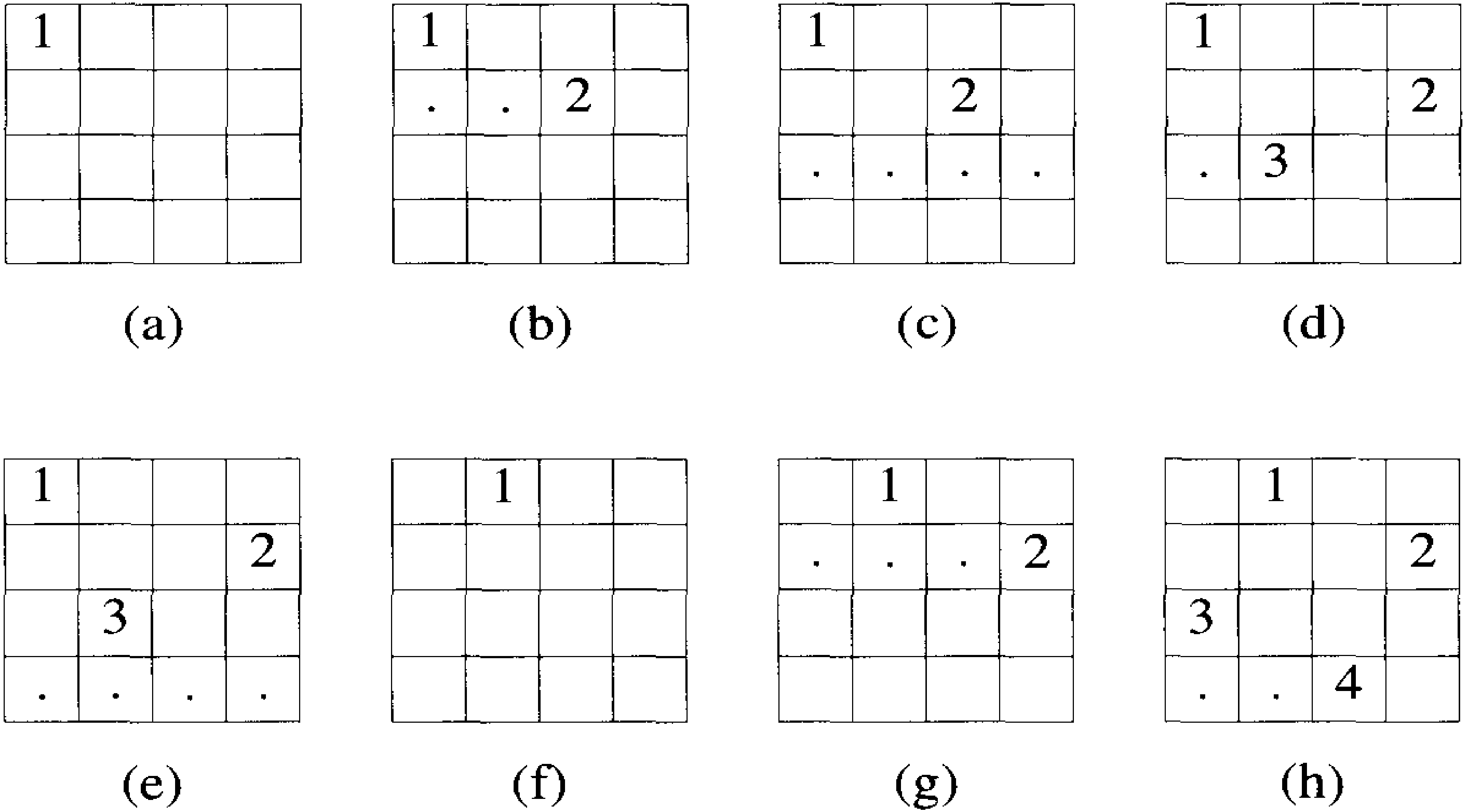
The 8-Queens Problem

- One strategy: guess at a solution
  - There are 4,426,165,368 ways to arrange 8 queens on a chessboard of 64 squares
- An observation that eliminates many arrangements from consideration
  - No queen can reside in a row or a column that contains another queen
  - Now: only 40,320 (8!) arrangements of queens to be checked for attacks along diagonals

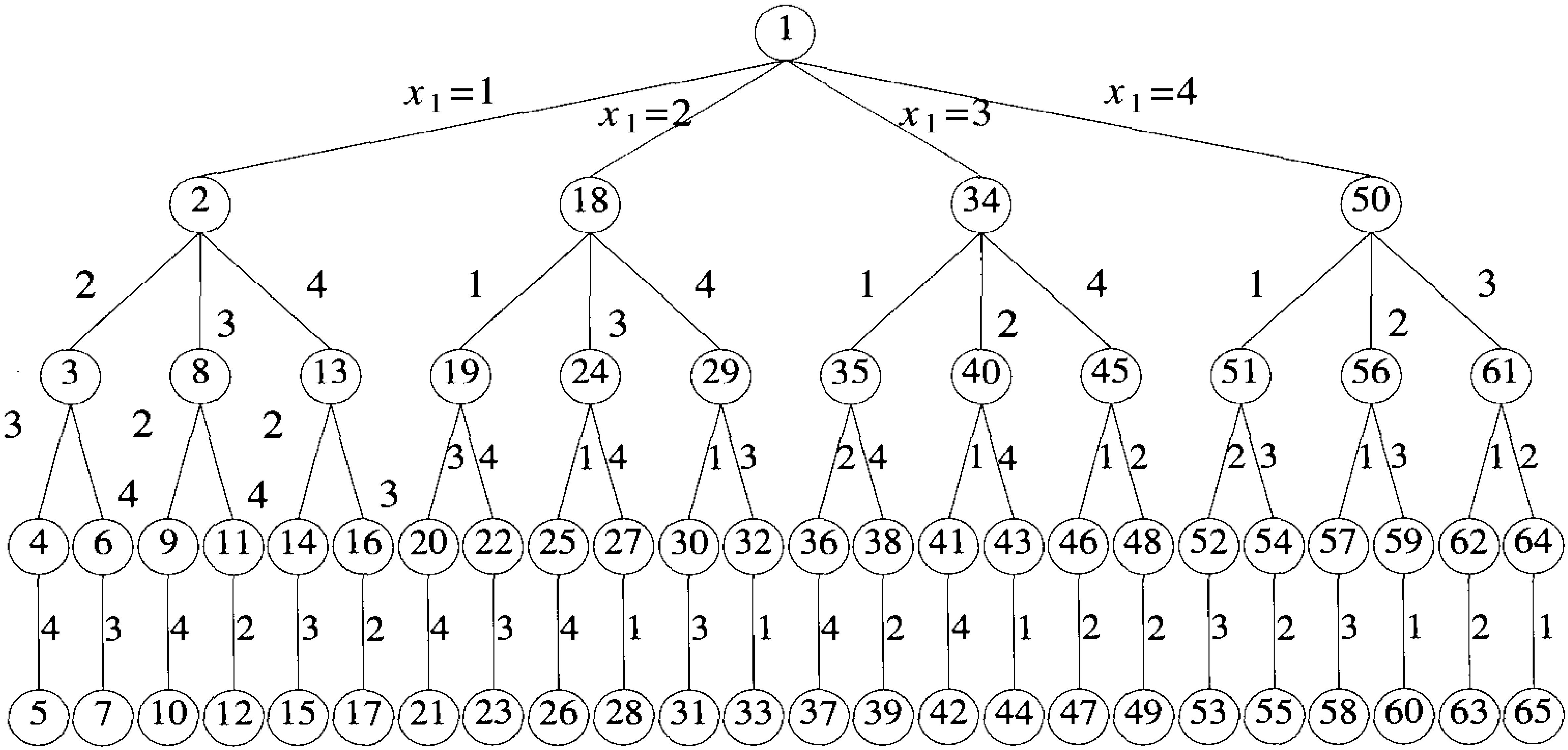
The 8-Queens Problem

- A recursive algorithm that places a queen in a column
  - Base case
    - If there are no more columns to consider
      - You are finished
  - Recursive step
    - If you successfully place a queen in the current column
      - Consider the next column
    - If you cannot place a queen in the current column
      - You need to backtrack

The 8-Queens Problem



The 8-Queens Problem



Solution Space of 4-Queens Problem

The 8-Queens Problem

- The total number of nodes in the N-Queens state space tree is:
$$1+\sum_{j=0}^{N-1}\left[\prod_{i=0}^{j}(N-i)\right]$$
- Every element on the same diagonal (1st) has the same row - column value
- Again, every element on the same diagonal (2nd) has the same row + column value
- If two queens are placed at position (i, j) and (k, l) then

  $i - j = k - l$ or $i + j = k + l$
- Therefore, two queens lie on the same diagonal iff $|j-l| = |i-k|$

The 8-Queens Problem

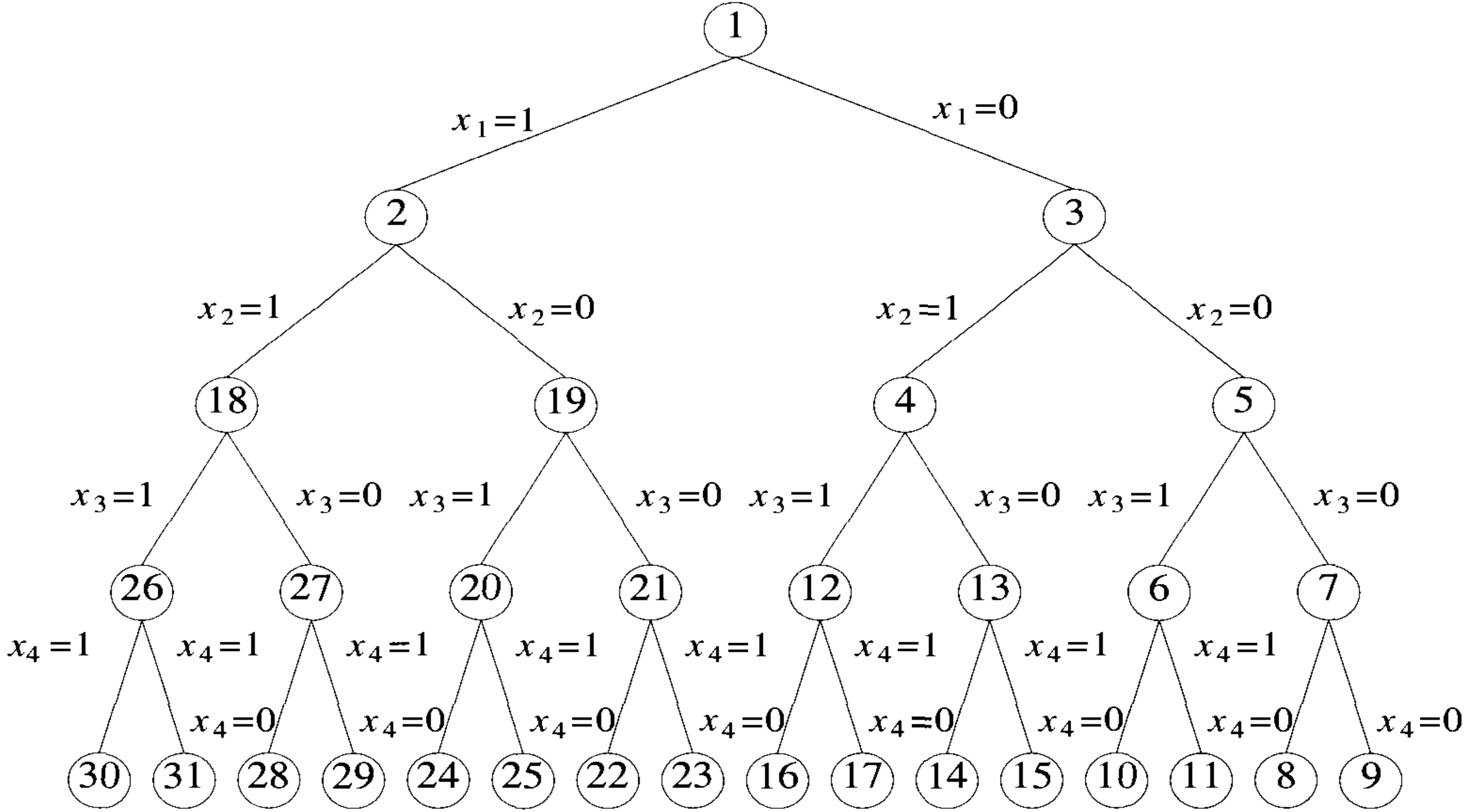
```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 :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

```
Algorithm Place(k, i)
// Returns true if a queen can be placed in kth row and
// ith 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;
}
```

Sum of Subsets Problem

- We are given n distinct positive numbers (weights) and we desire to find all combinations of these numbers whose sums are m.
- Element x_i of the solution vector is either one or zero depending on whether the weight w_i is included or not
- For a node at level i the left child corresponds to $x_i=1$ and the right to $x_i=0$

Sum of Subsets Problem



State Space Tree for SSP with n=4

Sum of Subsets Problem

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

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

- Clearly $x_1,\ldots,x_k$ cannot lead to an answer node if this condition is not satisfied
- The bounding function we use are therefore,

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

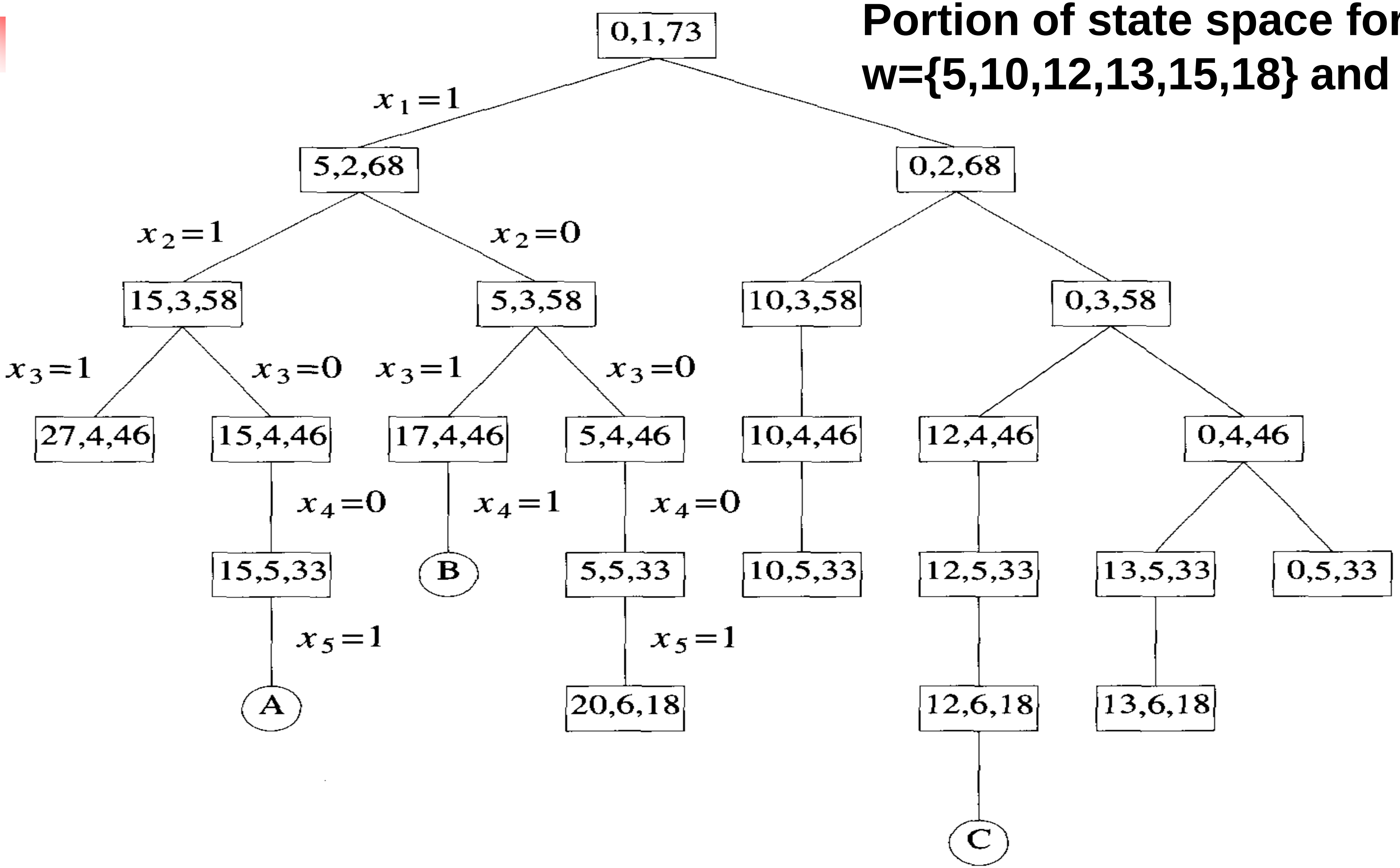
$$\text{and} \quad \sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

Sum of Subsets Problem

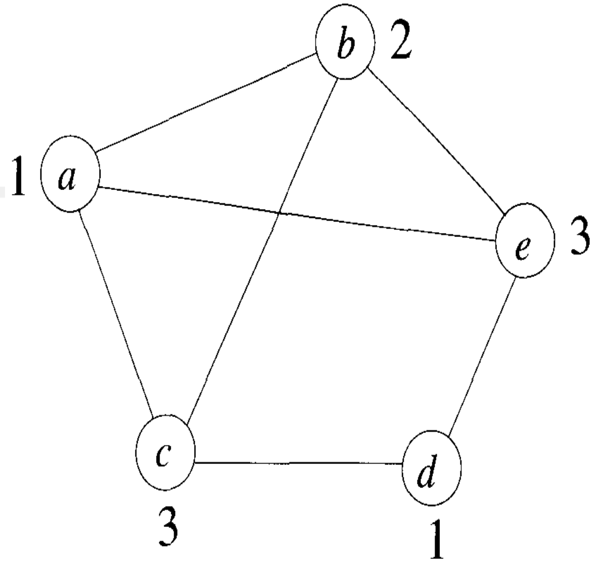
```
Algorithm SumOfSub(s, k, r)
// Find all subsets of w[1 : n] that sum to m. The values of x[j],
// 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
// and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
// It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
{
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

Sum of Subsets Problem

Portion of state space for w={5,10,12,13,15,18} and m=30



Graph Coloring



- Let G be a graph and m be a given positive integer indicating the number of colors
- 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 m-colorability decision problem
- 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

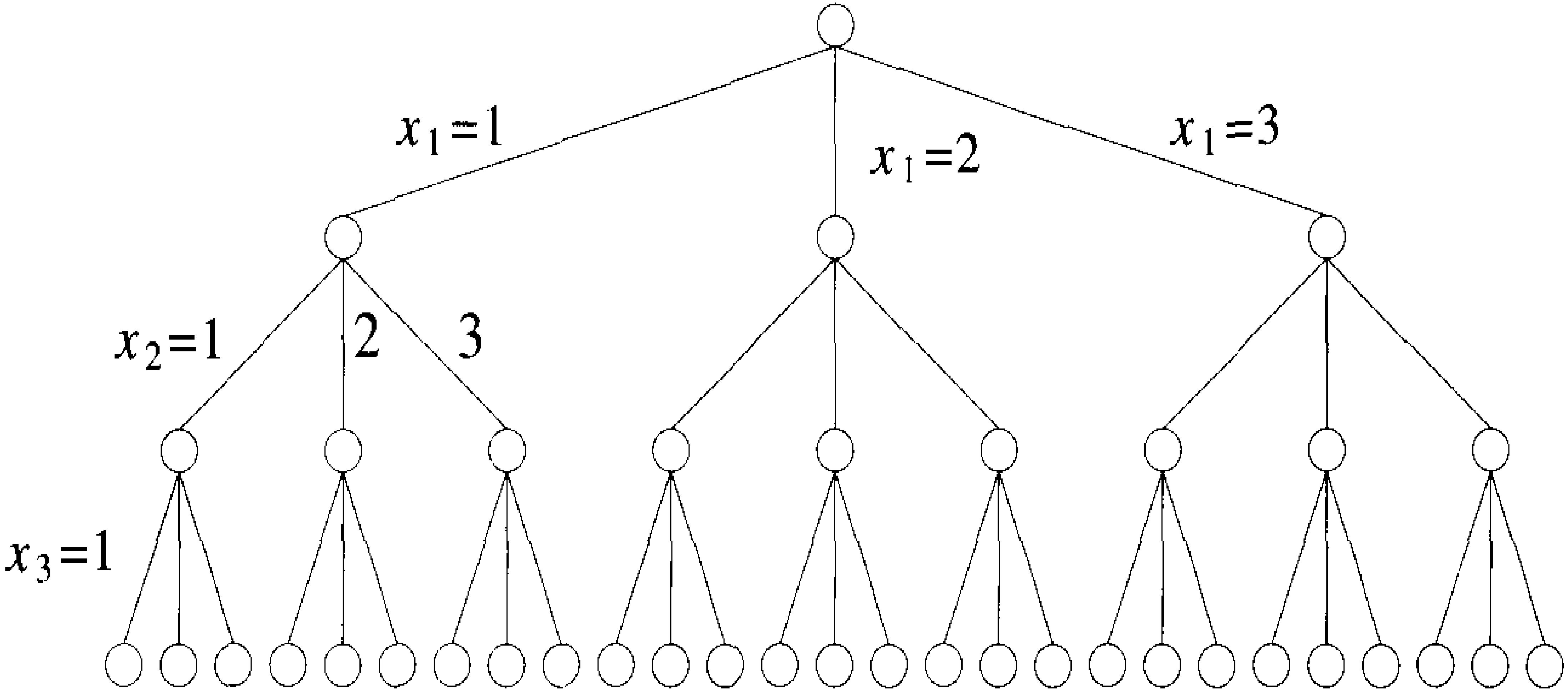
Graph Coloring

```
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);
}
```

Graph Coloring

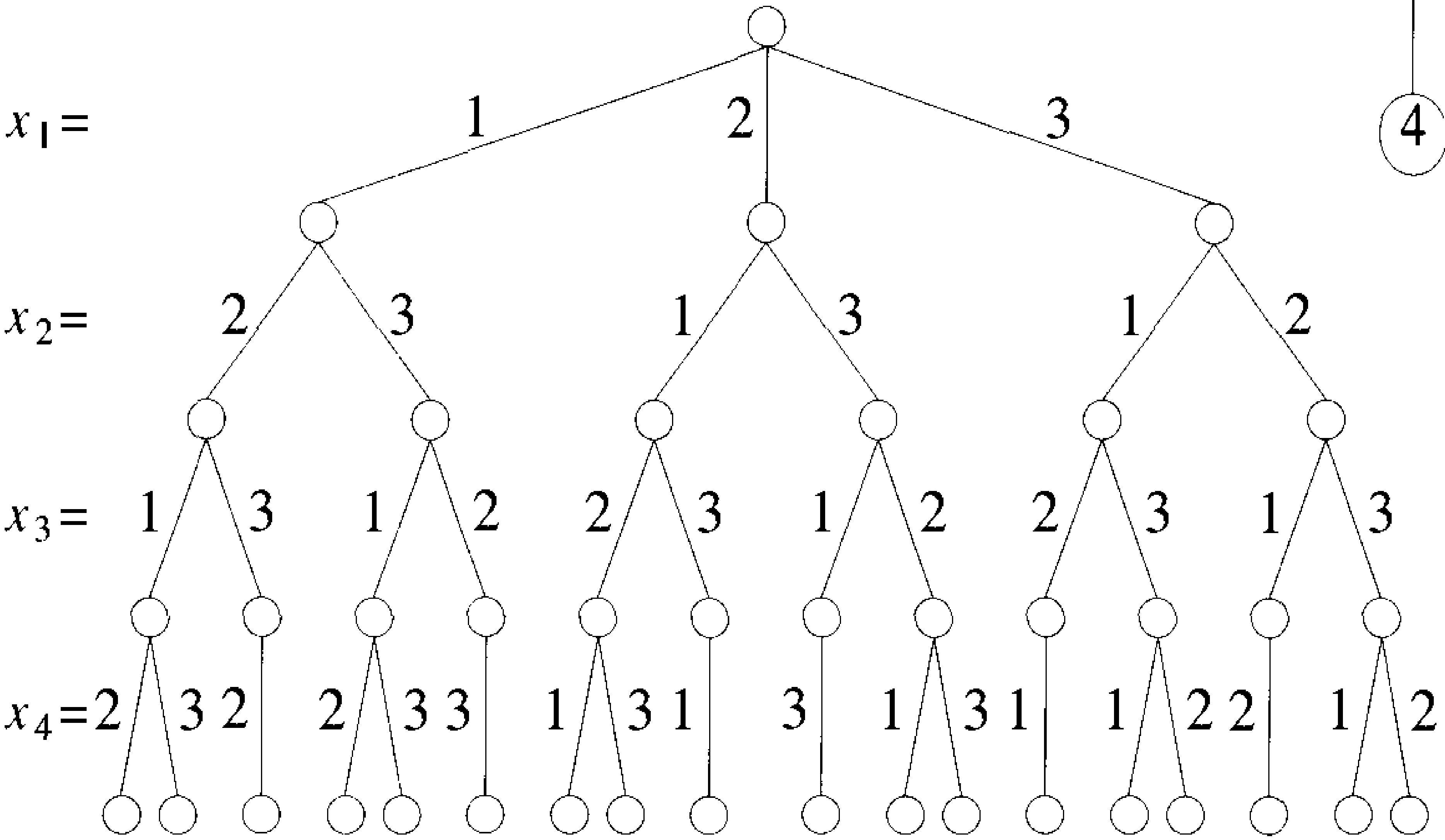
```
Algorithm NextValue(k)
// x[1], ..., x[k − 1] have been assigned integer values in
// the range [1, m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0, m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from 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.
}
```

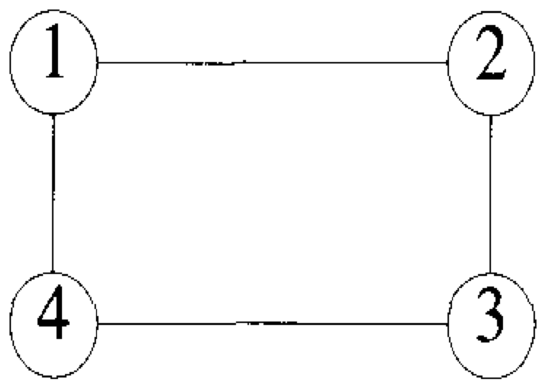
Graph Coloring



State space tree for mcoloring when n=3, m=3

Graph Coloring



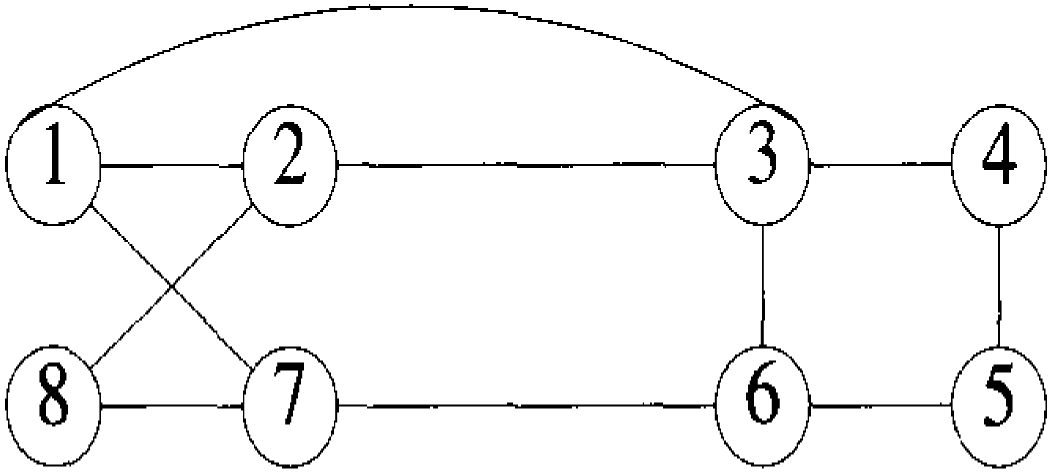


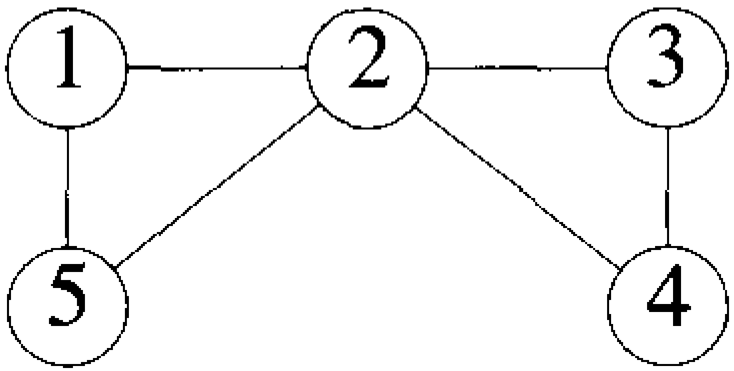
A 4-node graph and all possible 3-coloring

Hamiltonian Cycles

- 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, for all n and the vi are distinct except for v_1 and v_{n+1}, which are equal

Hamiltonian Cycles





Any??

Hamiltonian Cycles

```
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);
}
```

Hamiltonian Cycles

```
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);
}
```

Thanks for your Attention

Q&A