(A0510194) DESIGN AND ANALYSIS OF ALGORITHMS

UNIT V

UNIT-V:

Backtracking: General method, applications-n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.

Backtracking:

In the search for fundamental principles of algorithm design, backtracking represents one of the most general techniques. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation.

In order to apply the backtrack method, the desired solution must be expressible as an n-tuple (x_i, ... , x_n) where the x1 are chosen from some finite set S_1

The problem to be solved calls for finding one vector which maximizes (or minimizes or satisfies) a criterion function P(x_i, ... ,x_n).

Example:
Sorting the integers in A(l:n) is a problem whose solution is expressible by an n-tuple where x; is the index in A of the i^{th} smallest element.

"The criterion function P_i the inequality $A(x_i) \leq A(x_{i+1})$ for $1 \leq i < n$"

Suppose m_i is the size of set S_i. Then there are $m = m_1 m_2 \cdots m_n$ n-tuples which are possible candidates for satisfying the function P. The *brute force approach* would be to form all of these n-tuples and evaluate each one with P, saving those which yield the optimum. The backtrack algorithm has as its virtue the ability to yield the same answer with far fewer than m trials.

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:

1. explicit
2. implicit

Explicit constraints are rules that restrict each x_i to take on values only from a given set.
Eg:

1. $x_i > 0$ or Si = {all non-negative real numbers}
2. $x_i = 0$ or 1 or Si = {0,1}
3. $l_i < x_i < u_i$ or Si = {a: l_i < a < u_i }

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.

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.

- **Implicit constraint:** It is a rule in which how each element in a tuple is related.
- **Explicit constraint:** The rules that restrict each element to be chosen from the given set.

Let $(x_1, x_2, x_3, \ldots, x_i)$ be a path from the root to a node in a state space tree. Let $T(x_1, x_2, x_3, \ldots, x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, x_2, x_3, \ldots, x_{i+1})$ is also a path to a problem state.

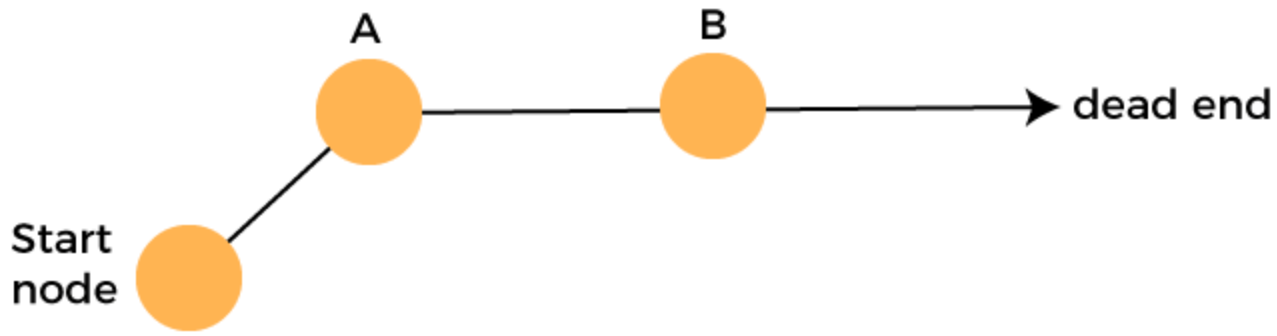
Let the bounding function B_{i+1} such that if $B_{i+1}(x_1, x_2, x_3, \ldots, x_{i+1})$ is false for a path$(x_1, x_2, x_3, \ldots, x_{i+1})$ from the root node to a problem state, then the path cannot be extended to reach an answer node.

Backtracking is one of the techniques that can be used to solve the problem. We can write the algorithm using this strategy. It uses the Brute force search to solve the problem, and the brute force search says that for the given problem, we try to make all the possible solutions and pick out the best solution from all the desired solutions. This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems. In contrast, backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.

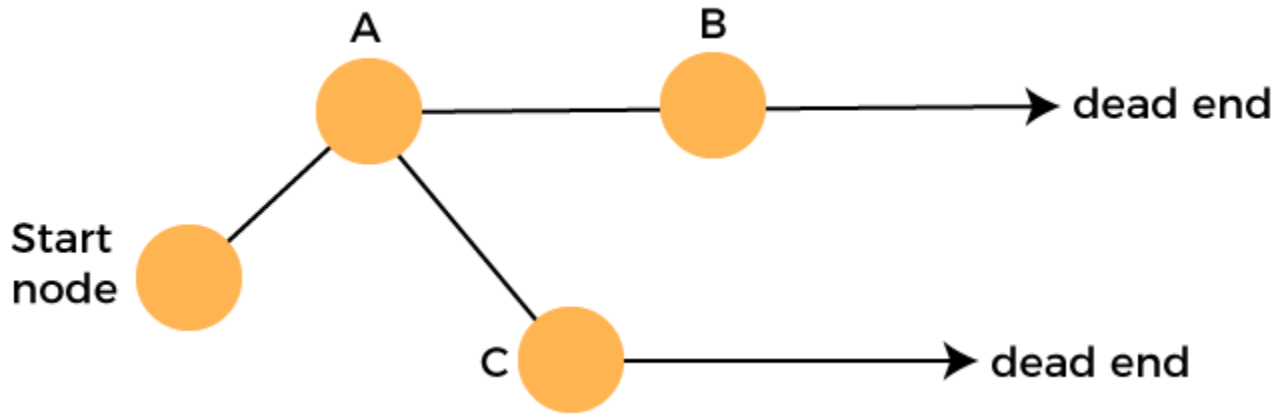
Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

Backtracking is a systematic method of trying out various sequences of decisions until you find out that works. Let's understand through an example.

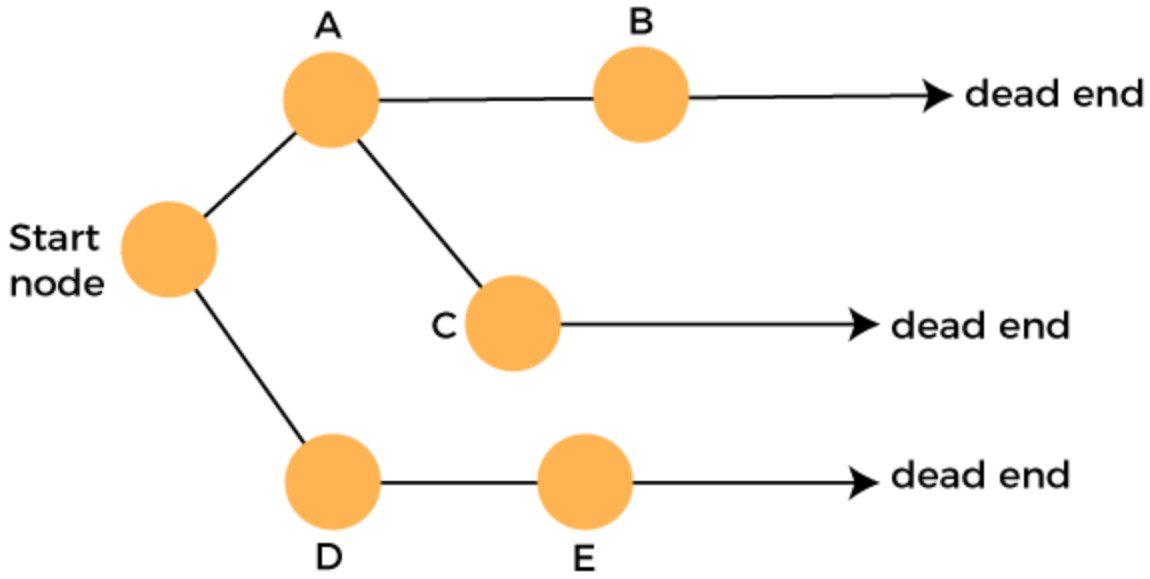
We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.



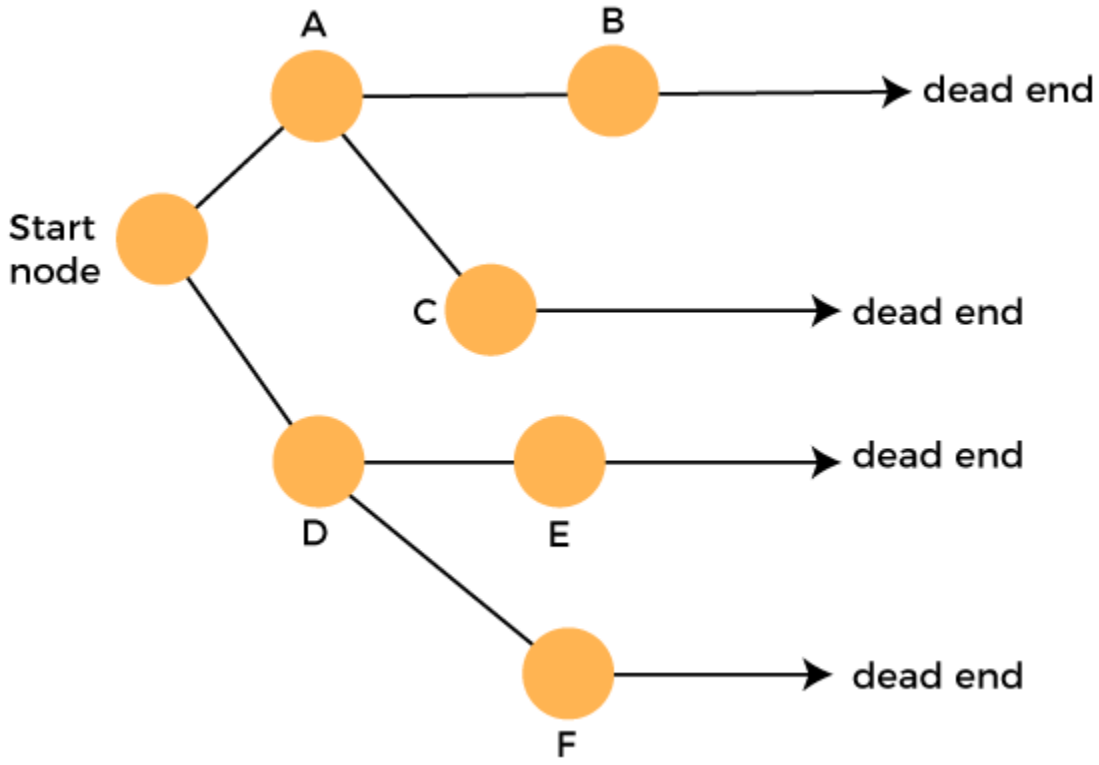
Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead-end, so again backtrack from node C to node A. We move from node A to the starting node.



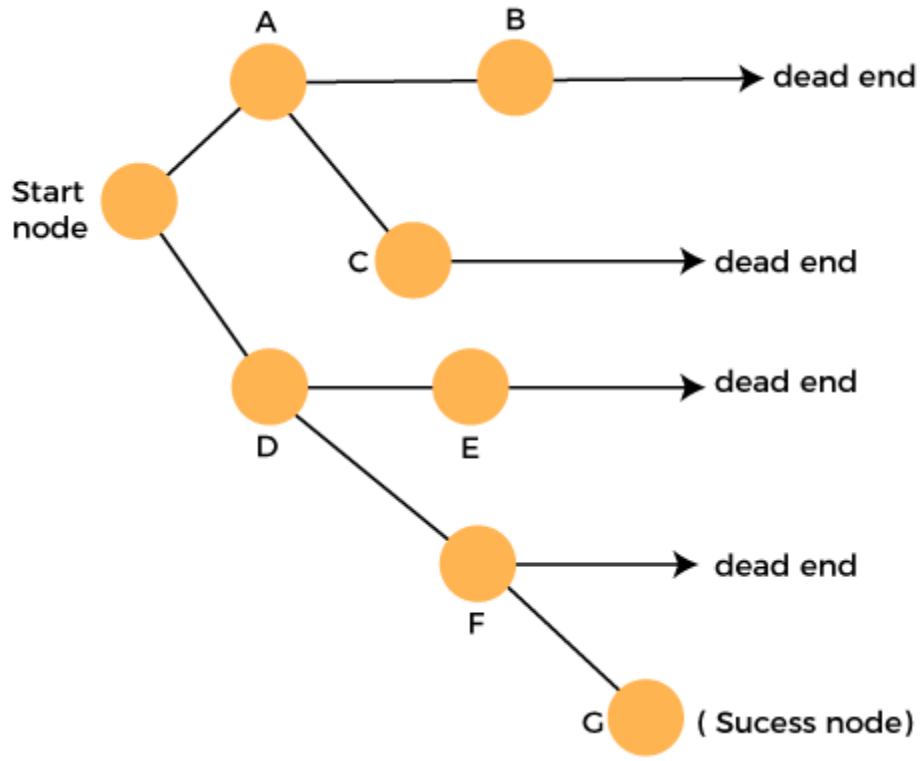
Now we will check any other path exists from the starting node. So, we move from start node to the node D. Since it is not a feasible solution so we move from node D to node E. The node E is also not a feasible solution. It is a dead end so we backtrack from node E to node D.



Suppose another path exists from node D to node F. So, we move from node D to node F. Since it is not a feasible solution and it's a dead-end, we check for another path from node F.



Suppose there is another path exists from the node F to node G so move from node F to node G. The node G is a success node.



Recursive Backtracking algorithm:

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

Iterative Backtracking algorithm:

```
Algorithm IBacktrack(n)
// This schema describes the backtracking process.
// All solutions are generated in x[1 : n] and printed
// as soon as they are determined.
{
    k := 1;
    while (k ≠ 0) do
    {
        if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
            x[k − 1]) and B_k(x[1], ..., x[k]) is true) then
        {
                if (x[1], ..., x[k] is a path to an answer node)
                    then write (x[1 : k]);
                k := k + 1; // Consider the next set.
        }
        else k := k − 1; // Backtrack to the previous set.
    }
}
```

8-Queen's Problem:

- The eight queens problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal)

- We use a backtracking algorithm to find a solution to the 8 Queen problem, which consists of placing 8 queens on a chessboard in such a way that no two queens threaten each other.

- The algorithm starts by placing a queen on the first column, then it proceeds to the next column and places a queen in the first safe row of that column.

- If the algorithm reaches the 8th column and all queens are placed in a safe position, it prints the board and returns true.

- If the algorithm is unable to place a queen in a safe position in a certain column, it backtracks to the previous column and tries a different row.

N-Queen's problem:

Consider an n X n chess board and try to find all the ways to place n non-attacking queens.

Let $(x_1, x_2, x_3, \ldots, x_n)$ represent a solution in which x_i is the column of the i^{th} row where the i^{th} queen is placed. The x_i's will all be distinct since no two queens can be placed in the same column.

If we imagine the chess board squares being numbered as the indices of the two-dimensional array a[1:n, 1:n], then we observe that every element on the same diagonal that runs from the upper left to the lower right has the same **row - column** value. Also, every element on the same diagonal that goes from the upper right to lower left has the same **row + column** value.

Suppose two queens are placed at positions (i,j) and (k,l). Then they are on the same diagonal only if

$$i - j = k - l \quad \text{or} \quad i + j = k + l$$

$$\Rightarrow \quad j - l = i - k \quad \text{or} \quad \Rightarrow \quad j - l = k - i$$

Therefore, two queens lie on the same diagonal if and only if $|j - l| = |i - k|$.

```
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;
}
```

```
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);
        }
    }
}
```

Sum of Subsets Problem:

Suppose we are given n distinct positive numbers and we desire to find all combinations of these numbers whose sums are m. This is called the sum of subsets problem.

Let n = 6, m = 30, and w[1 : 6] = {5,10,12,13,15,18}.
The solutions are:

1. (1, 1, 0, 0, 1, 0)
2. (1, 0, 1, 1, 0, 0)
3. (0, 0, 1, 0, 0, 1)

A simple choice for the bounding functions is B({x_1,x_2,…,x_k) = true iff

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

Clearly x_i,…,x_k cannot lead to an answer node if this condition is not satisfied. The bounding functions can be strengthened if we assume the w_i's are initially in non-decreasing order. In this case x_i,…,x_k cannot lead to an answer node if

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} > m$$

Therefore, the bounding functions we use are

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

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

Algorithm SumOfSub uses two variables s and r to store the values of $\sum_{i=1}^{k} w_i x_i$ and $\sum_{i=k+1}^{n} w_i$ respectively.

$$\sum_{i=1}^{..} w_i \geq m$$

The algorithm assumes $w_1 \leq m$ and

The initial call is SumOfSub (0, 1, $\sum_{i=1}^{n} w_i$)

```
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]);
    }
}
```

Graph Coloring problem:

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 called as m-colorability decision problem.

If d is the degree of the given graph, then it can be colored with d + 1 colors.

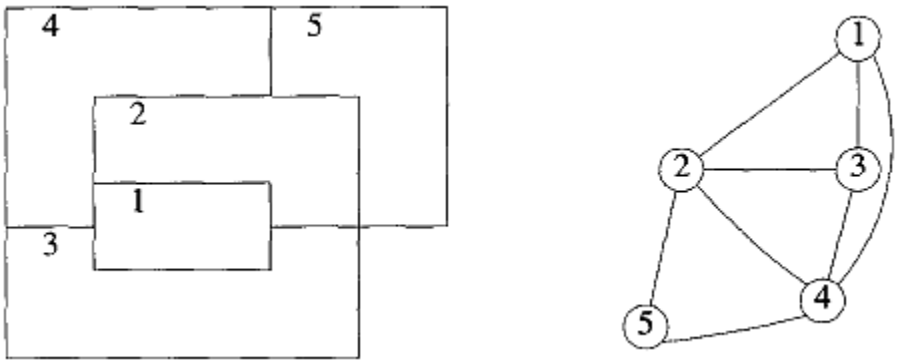
The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the chromatic number of the graph.

A graph is said to be planar if and only if 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. The problem is: 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.

Since a map can easily be transformed into a graph. Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge.

A map and its planar graph representation are shown below:



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).

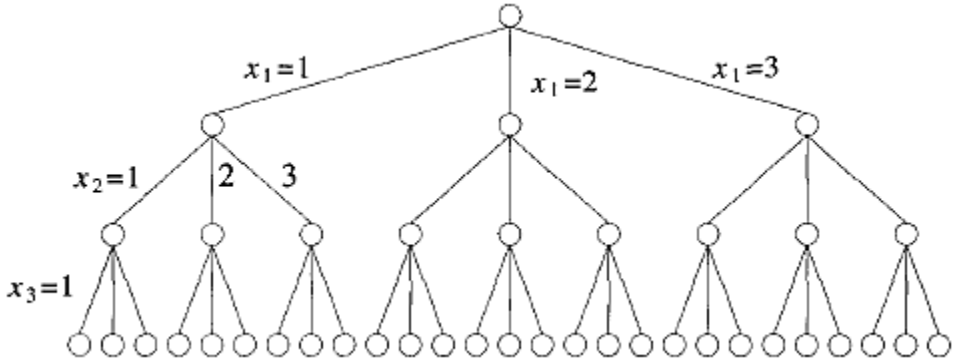
Finding all m-colorings of a graph:

```
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);
}
```

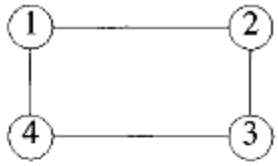
Generating a next color:

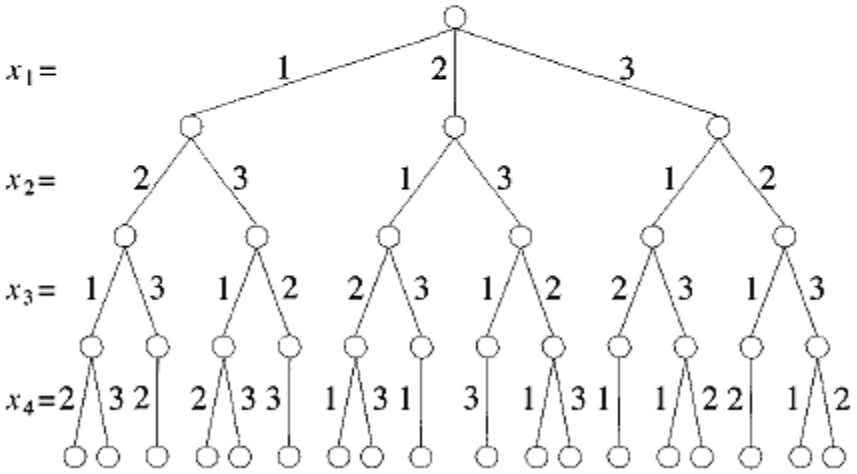
```
Algorithm NextValue(k)
// x[1], ..., x[k − 1] have been assigned integer values in
// the range [1, m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0, m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from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

State space tree for mColoring when n =3 and m = 3.



A 4-node graph and all possible 3-colorings

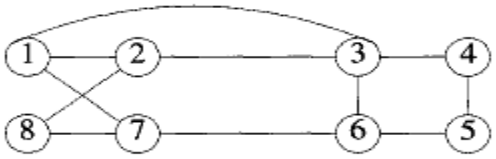




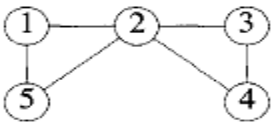
Hamiltonian Cycles:

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

G1:



G2:



Graph G1 contains a Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1.
Graph G2 does not contain any Hamiltonian cycle.

The backtracking solution vector (x_1, x_2... x_n) is defined so that x_i represents the i^{th} visited vertex of the proposed cycle.

Generating a next vertex:

```
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);
}
```

Finding all 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);
}
```

This algorithm is started by first initializing the adjacency matrix G[l:n, 1:n], then setting x[2:n] to zero and x[l]to 1, and then executing Hamiltonian (2).