Backtracking

- **Backtracking** is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.
- The name backtrack was first coined by D.H.Lehmerin the 1950s.
- In many applications of the backtrack method, the desired solution is expressible as an n-tuple(x_i,...,x_n),where the x_i are chosen from some finite set Sj.
- Often the problem to be solved calls for finding one vector that maximizes(or minimizes or satisfies)a criterion function P(x_i,….x_n).
- 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.

Backtracking

- **Explicit constraints** are rules that restrict each x_i to take on values only from a given set.
- **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 explicit constraints depend on the instance I of the problem being solved.
- All tuples that satisfy the explicit constraints define a possible solution space for I.

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);
        }
    }
}
```

General iterative backtracking method

```
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.
     }
}
```

Applications

- N Queen Problem
- Sum of Subsets
- Graph Colouring
- Hamiltonian Cycle

N Queen Problem

- Given a chess board having N×N cells, we need to place N queens in such a way that no queen is attacked by any other queen.
- A queen can attack horizontally, vertically and diagonally.

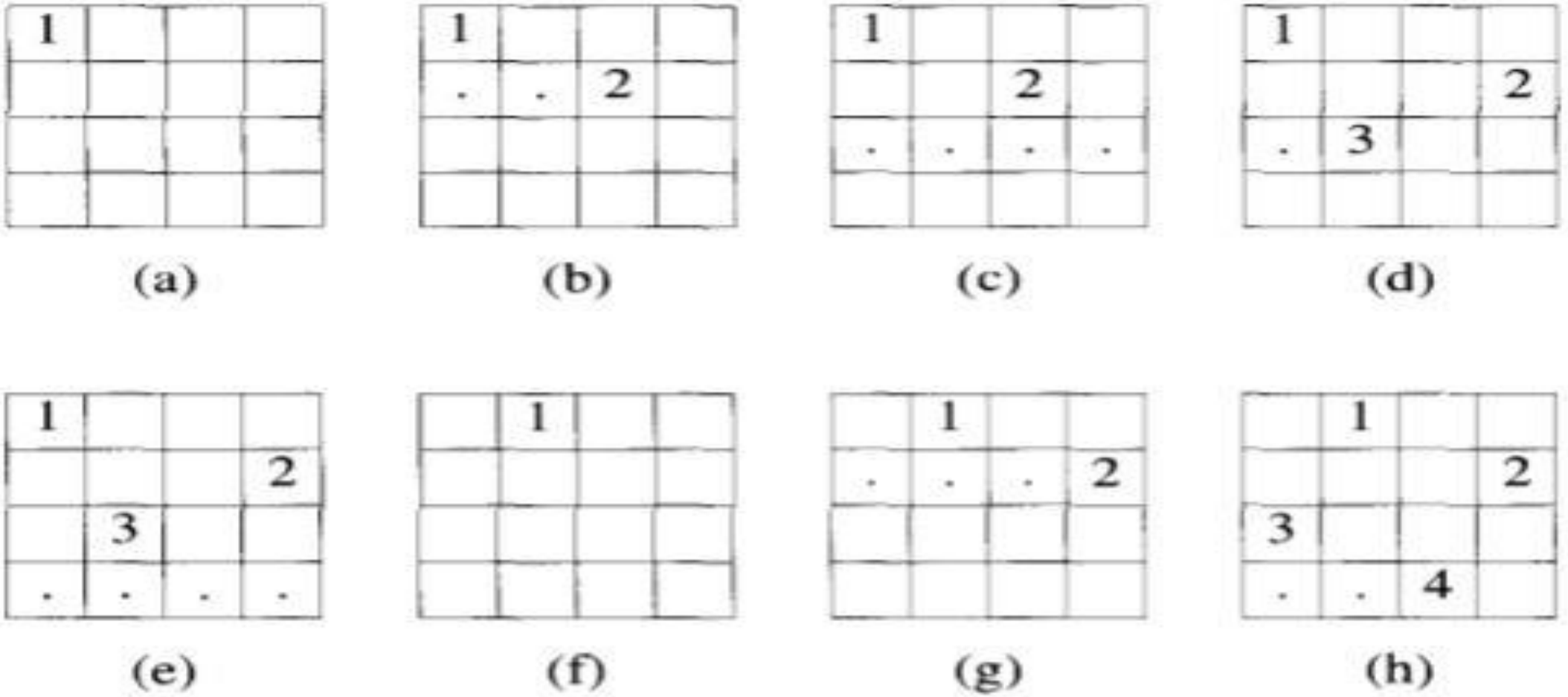
Approach

- So initially we are having N×N un attacked cells where we need to place N queens.
- Let's place the first queen at a cell (i,j), so now the number of un attacked cells is reduced, and number of queens to be placed is N−1.
- Place the next queen at some un attacked cell.
- This again reduces the number of un attacked cells and number of queens to be placed becomes N−2. Continue doing this, as long as following conditions hold.
  - The number of un attacked cells is not 0.
  - The number of queens to be placed is not 0.

Approach

- If the number of queens to be placed becomes 0, then it's over, we found a solution.
- But if the number of un attacked cells become 0, then we need to backtrack, i.e. remove the last placed queen from its current cell, and place it at some other cell.
- We do this recursively.

4-Queen Problem-Solution:



Algorithm

```
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

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k – 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k – 1 do
8           if ((x[j] = i) // Two in the same column
9               or (Abs(x[j] – i) = Abs(j – k)))
10                  // or in the same diagonal
11              then return false;
12      return true;
13  }
```

SUM OF SUBSETS

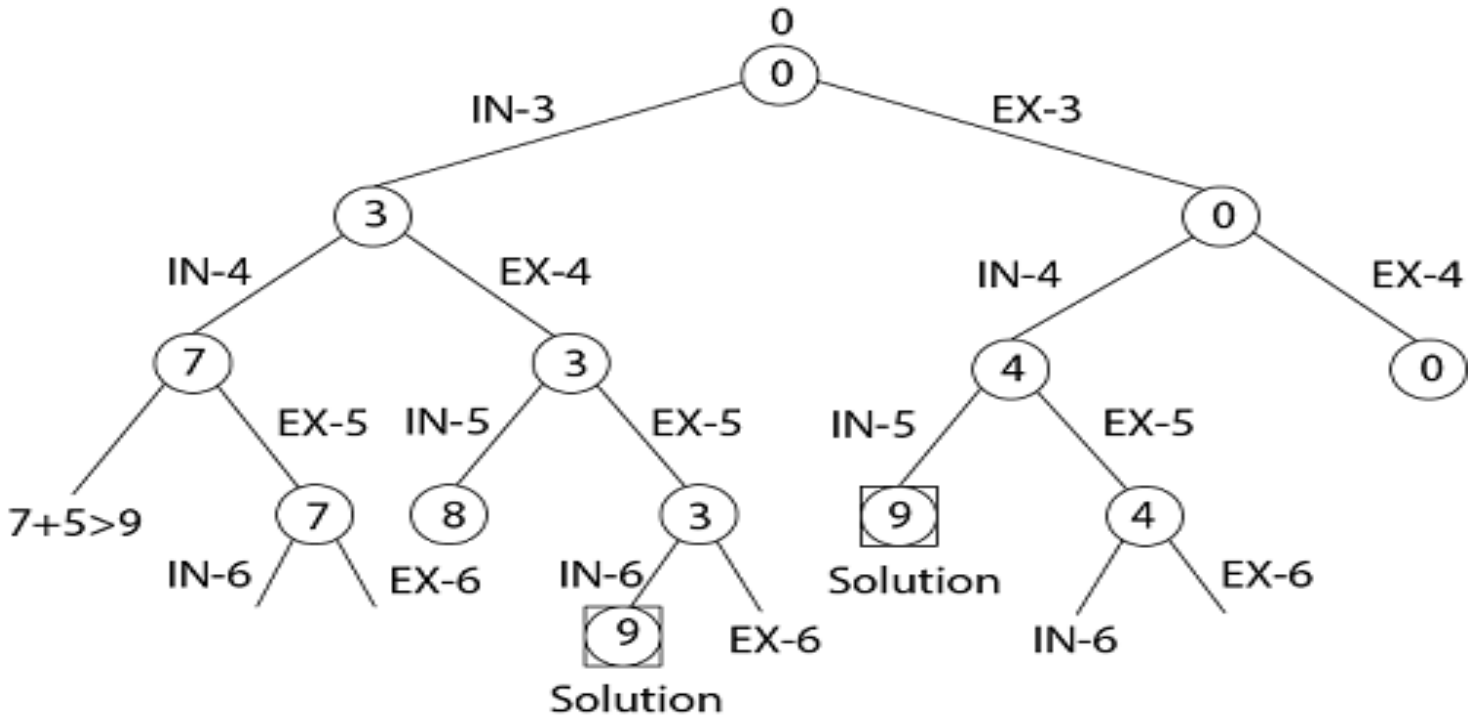
- Given a set of non-negative integers, and a value *sum*, determine if there is a subset of the given set with sum equal to given *sum*.
- Here backtracking approach is used for trying to select a valid subset when an item is not valid, we will backtrack to get the previous subset and add another element to get the solution.

Example

- Input: The Set: {10, 7, 5, 18, 12, 20, 15},The sum Value: 35
- Output:
- All possible subsets of the given set, where sum of each element for every subsets is same as the given sum value.
  - {10, 7, 18}
  - {10, 5, 20}
  - {5, 18, 12}
  - {20, 15}

Example and Portion of State Space Tree

- Given a set S = (3, 4, 5, 6) and X =9. Obtain the subset sum using Backtracking approach.
- Output:
  - {3,6}
  - {4,5}



Recursive backtracking Algorithm

```
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

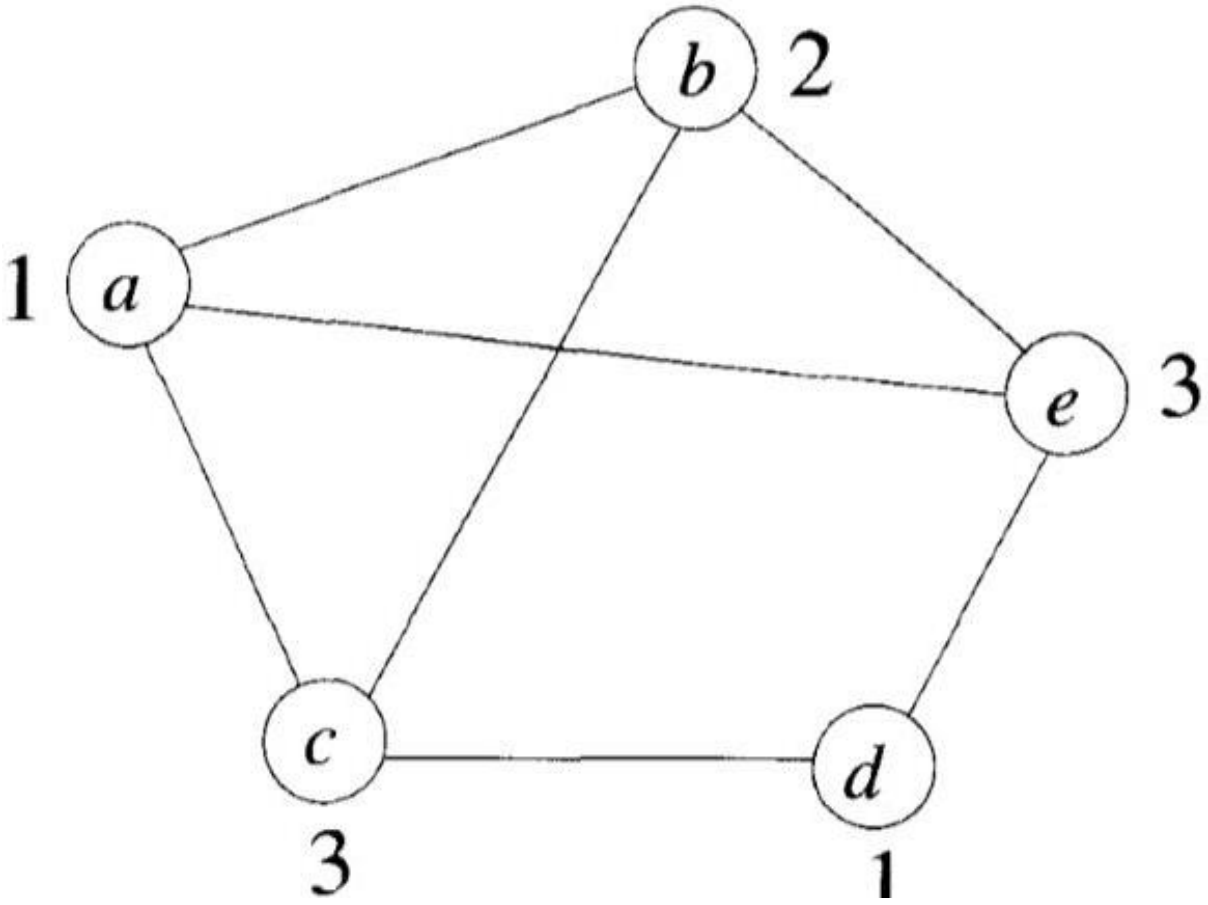
- Let G be a graph and m be a given positive integer.
- We want to discover whether the nodes of G can be coloured in such a way that no two adjacent nodes have the same colour yet only m colours are used.
- This is termed the m-colorability decision problem.
- The **chromatic number** χ(G) of a **graph** G is the minimal **number** of colors for which such an assignment is possible.

Note: that if **d** is the degree of the given graph, then it can be colored with **d+1** colors.

GRAPH COLORING

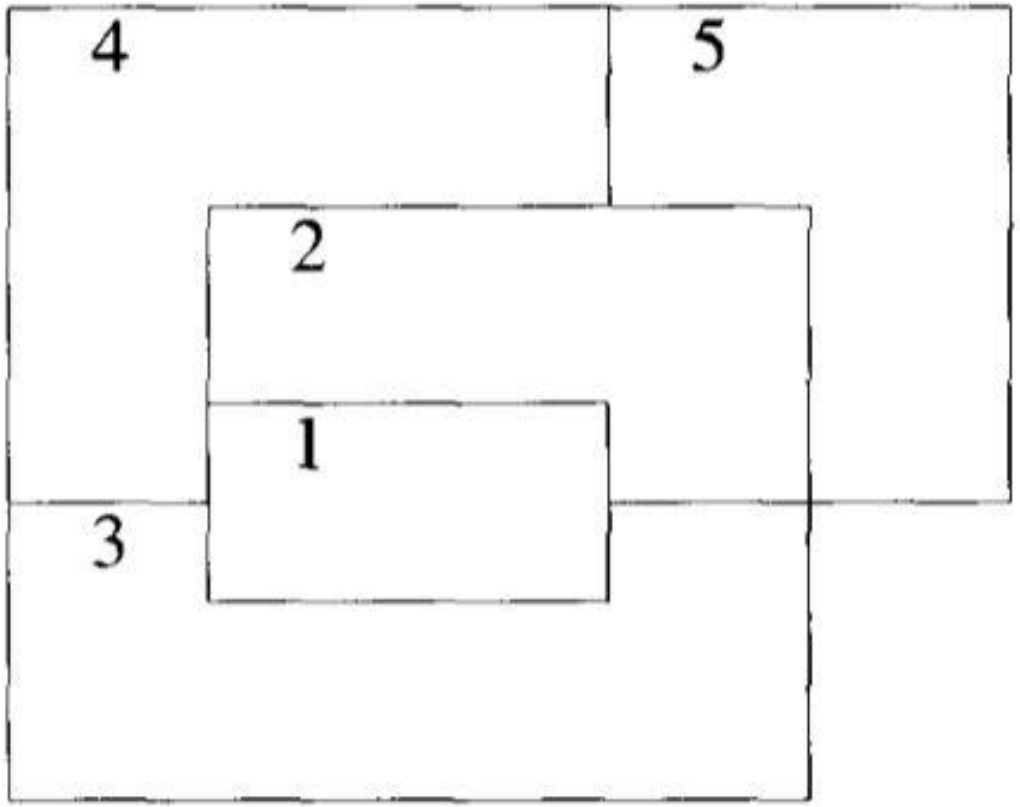
- A graph is said to be planar iff it can be drawn in a plane in such a way that no two edges cross each other.
- A famous special case of the m-colorability decision problem is the 4-color problem for planar graphs.

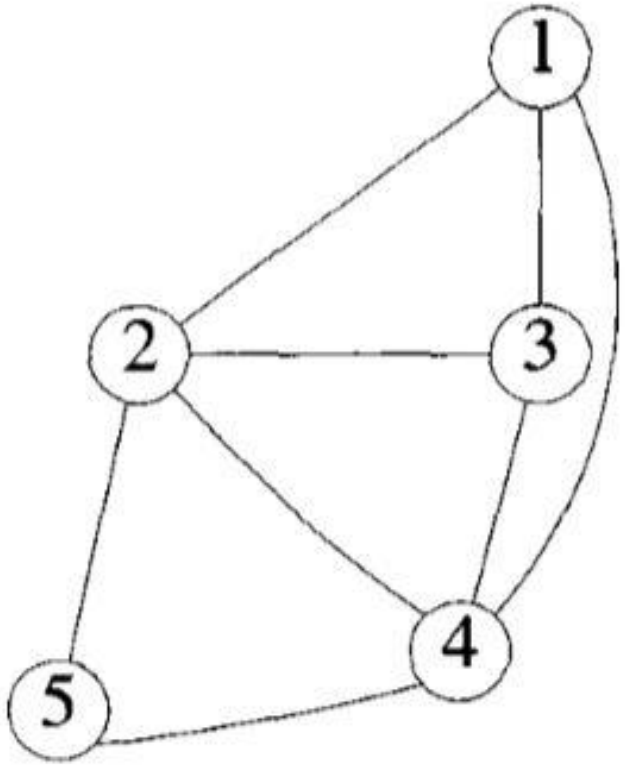
Example



- The given graph 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.

Example: A map and its planar graph representation





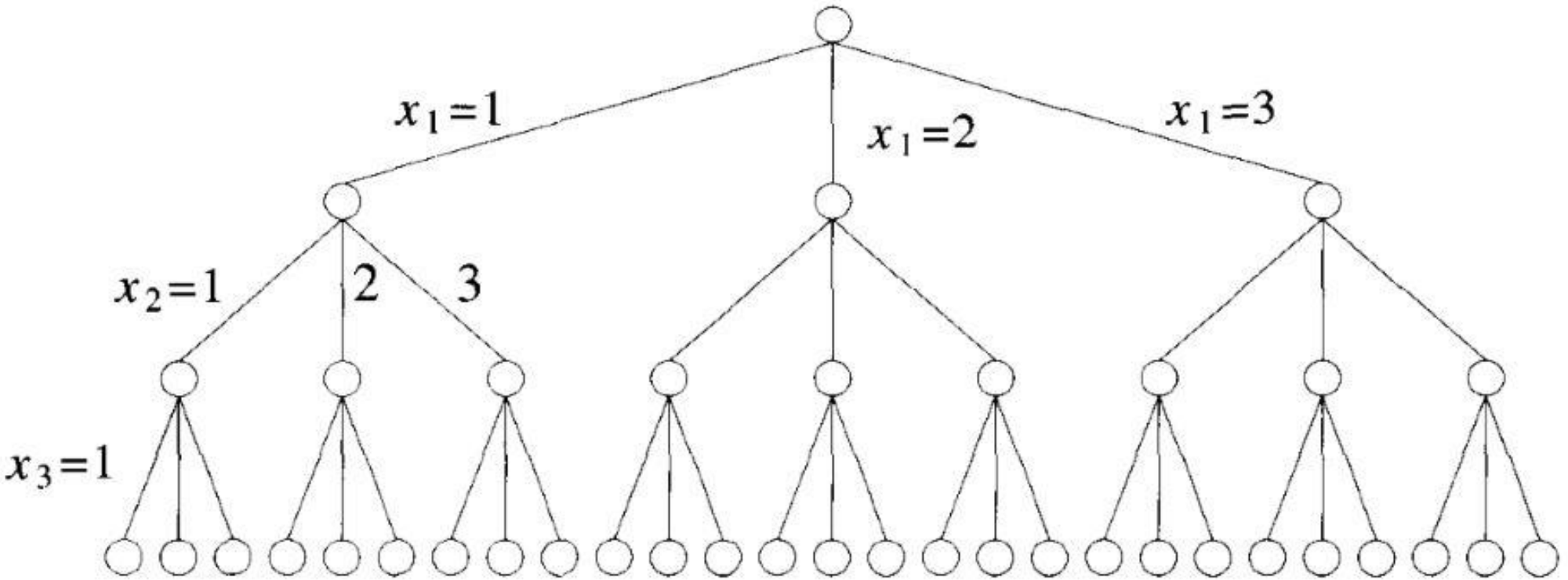
mColoring Algorithm

```
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);
}
```

NextValue Algorithm

```
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



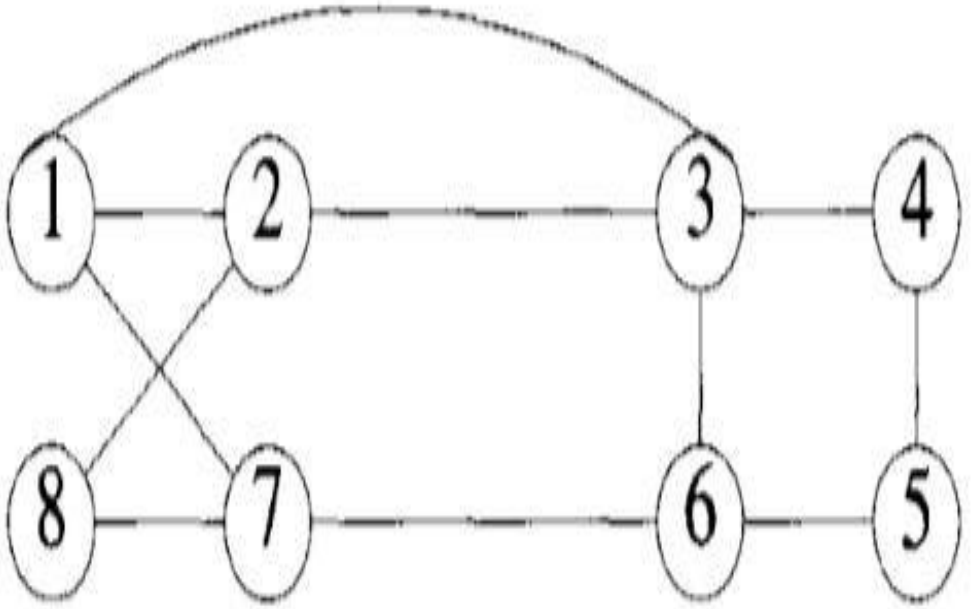
HAMILTONIAN CYCLES

- A **Hamiltonian cycle** is a closed loop on a **graph** where every node (vertex) is visited exactly once.

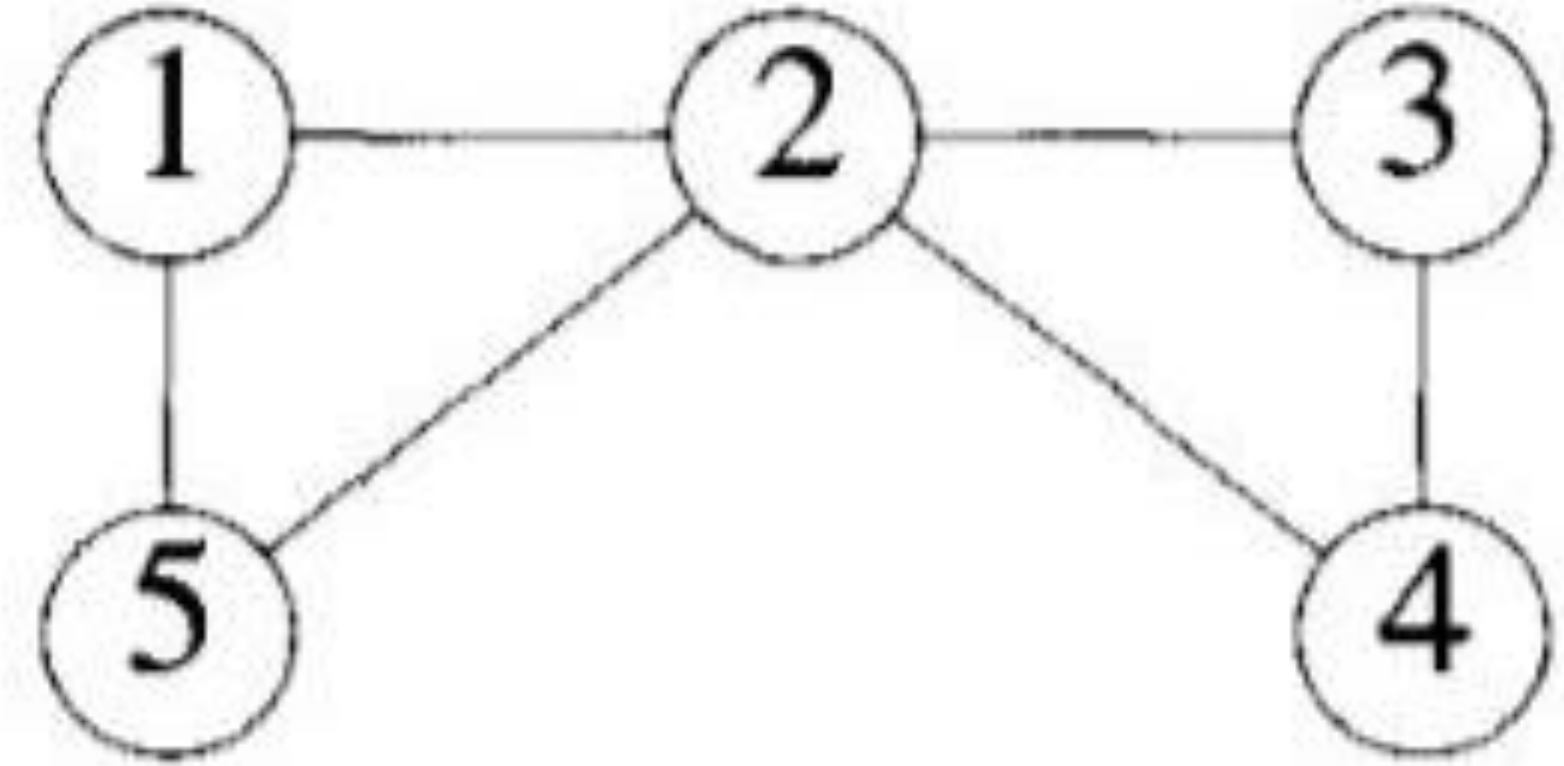
OR

- 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.

Example graph contains Hamiltonian Cycle



Example graph contains No Hamiltonian Cycle



Hamiltonian Algorithm

```
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);
}
```

NextValue Algorithm

```
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);
}
```