Program : **B.Tech**

Subject Name: **Analysis and Design of Algorithm**

Subject Code: **IT-403**

Semester: **4th**

Unit-4 Backtracking

Introduction:

Backtracking (General method)

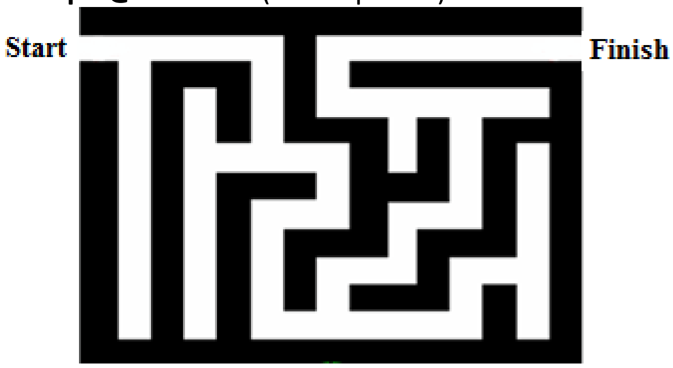
Many problems are difficult to solve algorithmically. Backtracking makes it possible to solve at least some large instances of difficult combinatorial problems.

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 (more than one choices) may be a solution to your problem.

Backtracking is a methodical (Logical) way of trying out various sequences of decisions, until you find one that "works"

Example@1 : Maze (a tour puzzle)



Given a maze, find a path from start to finish.

1. In maze, at each intersection, you have to decide between 3 or fewer choices:
   - ✓ Go straight
   - ✓ Go left
   - ✓ Go right
2. You don't have enough information to choose correctly
3. Each choice leads to another set of choices.
4. One or more sequences of choices may or may not lead to a solution.
5. Many types of maze problem can be solved with backtracking.

Example@ 2

Sorting the array of integers in a[1:n] is a problem whose solution is expressible by an n-tuple x_i is the index in 'a' of the i^{th} smallest element.

The criterion function 'P' is the inequality $a[x_i] \leq a[x_{i+1}]$ for $1 \leq i \leq n$

$S_{i=}$ is finite and includes the integers 1 through n.

$M_{i=}$ size of set S_i $m=m_1m_2m_3$---m_n n tuples that possible candidates for satisfying the function P. With brute force approach would be to form all these n-tuples, evaluate (judge) each one with P and save those which yield the optimum.

By using backtrack algorithm; yield the same answer with far fewer than 'm' trails.

Many of the problems we solve using backtracking requires that all the solutions satisfy a complex set of constraints.

For any problem these constraints can be divided into two categories:

- Explicit constraints.
- Implicit constraints.

Explicit constraints: Explicit constraints are rules that restrict each $\mathbf{x_i}$ to take on values only from a given set.

Example: $\mathbf{x_i \geq 0}$ or si={all non negative real numbers}

X_i=0 or 1 or S_i={0, 1}

$l_i \leq x_i \leq u_i$ or s_i={a: $l_i \leq a \leq u_i$ }

The explicit constraint depends 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:

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 $\mathbf{X_i}$ must relate to each other.

<u>Applications of Backtracking:</u>

- N Queens Problem
- Sum of subsets problem
- Graph coloring
- Hamiltonian cycles.

N-Queens Problem:

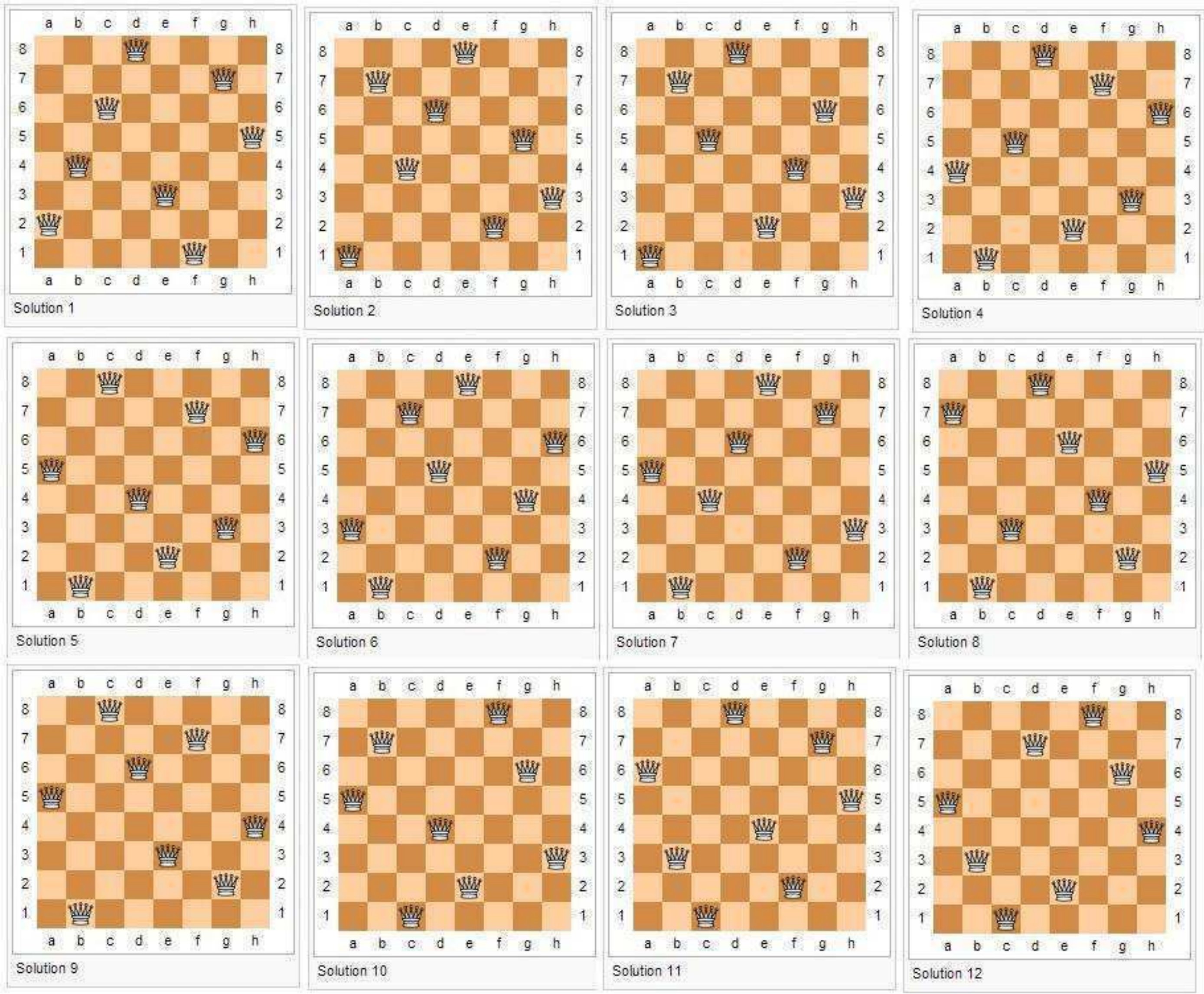
It is a classic combinatorial problem. The eight queen's puzzle is the problem of placing eight queens puzzle is the problem of placing eight queens on an 8×8 chessboard so that no two queens attack each other. That is so that no two of them are on the same row, column, or diagonal.
The 8-queens puzzle is an example of the more general n-queens problem of placing n queens on an n×n chessboard.

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

One solution to the 8-queens problem

Here queens can also be numbered 1 through 8
Each queen must be on a different row
Assume queen 'i' is to be placed on row 'i'

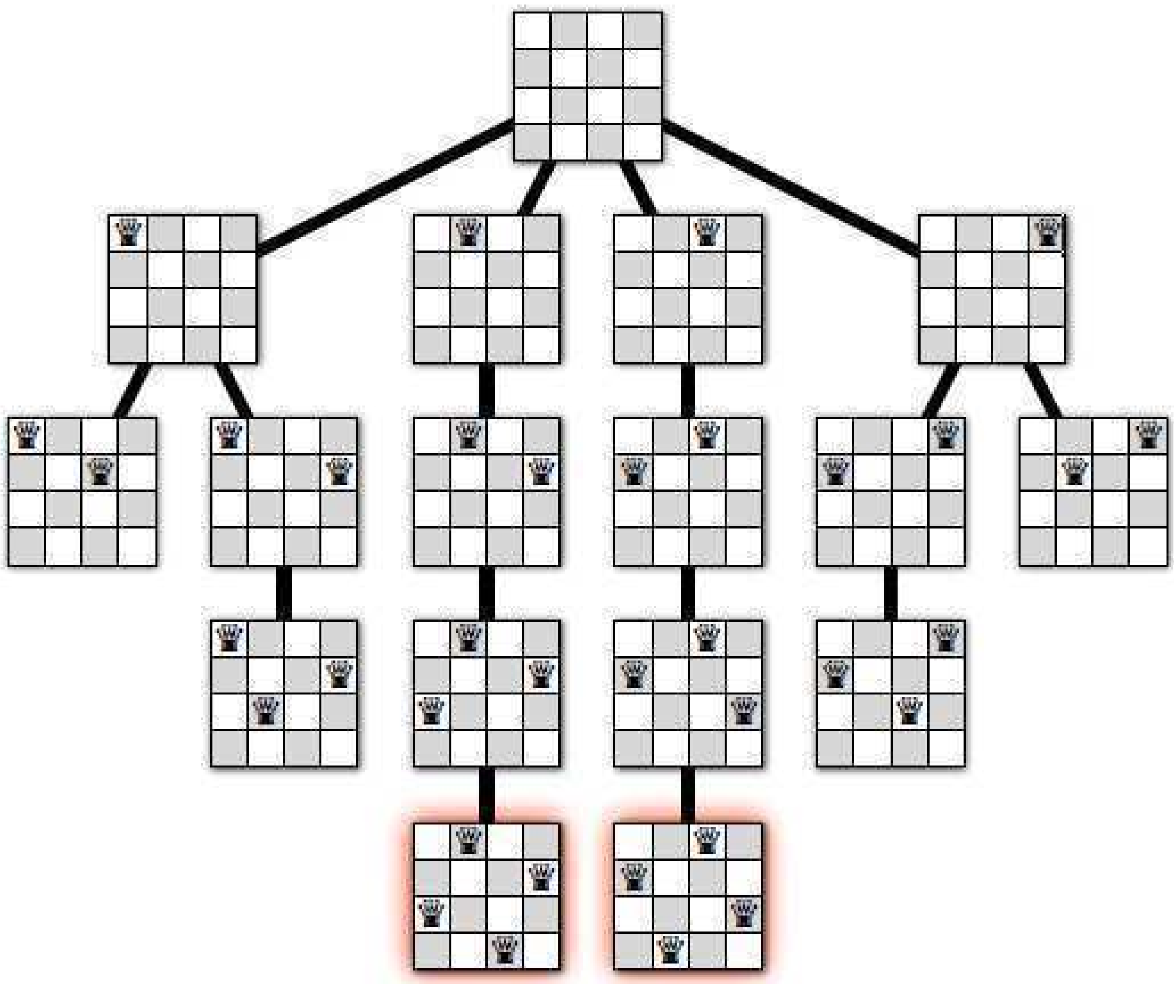
All solutions to the 8-queens problem can therefore be represented a s s-tuples(x_1, x_2, x_3—x_8)
xi= the column on which queen 'i' is placed
si={1, 2, 3, 4, 5, 6, 7, 8}, $1 \le i \le 8$
Therefore the solution space consists of 8^8 s-tuples.
The implicit constraints for this problem are that no two x_i's can be the same column and no two queens can be on the same diagonal. By these two constraints the size of solution pace reduces from 88 tuples to 8! Tuples.
Form example s_i(4,6,8,2,7,1,3,5)
In the same way for n-queens are to be placed on an n×n chessboard, the solution space consists of all n! Permutations of n-tuples (1,2,----n).



Some solution to the 8-Queens problem

Algorithm for new queen be placed	All solutions to the n·queens problem

Algorithm Place(k,i)
//Return true if a queen can be placed in kth row & ith column
//Other wise return false
{
for j:=1 to k-1 do
if(x[j]=i or Abs(x[j]-i)=Abs(j-k)))
then return false
return true
}
	Algorithm NQueens(k, n)
	// its prints all possible placements of n-queens on an n×n chessboard.
	{
	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 complete recursion tree for our algorithm for the 4 queens problem.

Sum of Subsets Problem: Given positive numbers w_i $1 \le i \le n$, & m, here sum of subsets problem is finding all subsets of w_i whose sums are m.

Definition: Given n distinct +ve numbers (usually called weights), desire (want) to find all combinations of these numbers whose sums are m. this is called sum of subsets problem.

To formulate this problem by using either fixed sized tuples or variable sized tuples. Backtracking solution uses the fixed size tuple strategy.

For example:

If n=4 (w_1, w_2, w_3, w_4)=(11,13,24,7) and m=31.

Then desired subsets are (11, 13, 7) & (24, 7).

The two solutions are described by the vectors (1, 2, 4) and (3, 4).

In general all solution are k-tuples (x_1, x_2, x_3---x_k) $1 \le k \le n$, different solutions may have different sized tuples.

- Explicit constraints requires $x_i \in \{j \,/\, j$ is an integer $1 \le j \le n\}$
- Implicit constraints requires:
  No two be the same & that the sum of the corresponding w_i's be m
  i.e., (1, 2, 4) & (1, 4, 2) represents the same. Another constraint is $x_i < x_{i+1}$ $1 \le i \le k$

$W_{i=}$ weight of item i

M= Capacity of bag (subset)

X_i = the element of the solution vector is either one or zero.
X_i value depending on whether the weight wi is included or not.
If X_i=1 then wi is chosen.
If X_i=0 then wi is not chosen.

$$\underbrace{\sum_{i=1}^{k} W(i)X(i)}_{\text{Total sum till now}} + \underbrace{\sum_{i=k+1}^{n} W(i)}_{\text{Still there}} \geq M$$

The above equation specify that x_1, x_2, x_3, --- x_k cannot lead to an answer node if this condition is not satisfied.

$$\sum_{i=1}^{k} W(i)X(i) + W(k+1) > M$$

The equation cannot lead to solution.

$$B_k(X(1), \ldots, X(k)) = true \text{ iff } \left(\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) \geq M \text{ and } \sum_{i=1}^{k} W(i)X(i) + W(k+1) \leq M \right)$$

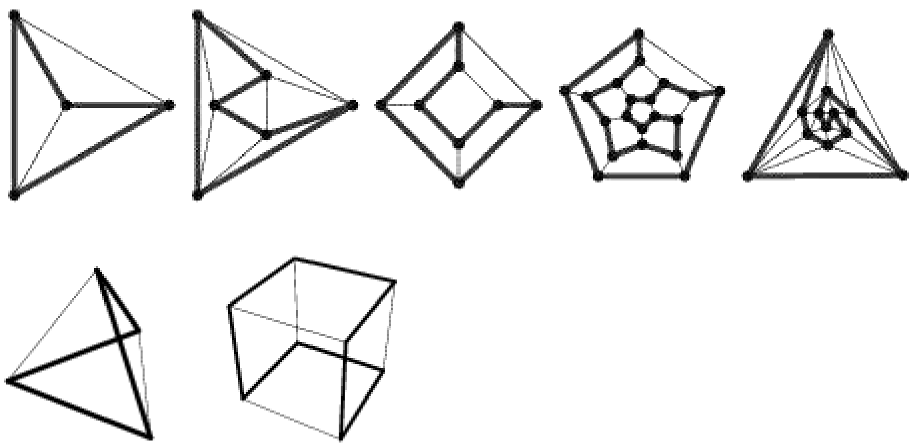
$$s = \sum_{j=1}^{k-1} W(j)X(j). \quad \text{and} \quad r = \sum_{j=k}^{n} W(j)$$

Recursive backtracking algorithm for sum of subsets problem

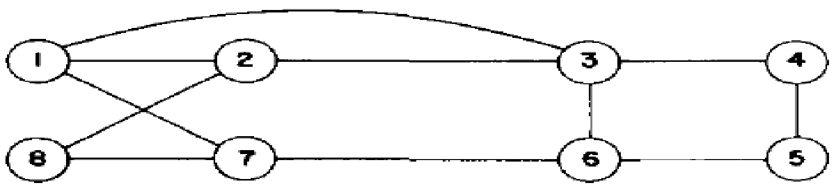
Algorithm SumOfSub(s, k, r)
{
// $s = \sum_{j=1}^{k-1} W(j)X(j).$ and $r = \sum_{j=k}^{n} W(j)$
X[k]=1
If(S+w[k]=m) then write(x[1:]); // subset found.
Else if (S+w[k] + w{k+1] ≤ M)
Then SumOfSub(S+w[k], k+1, r-w[k]);
if ((S+r - w{k] ≥ M) and (S+w[k+1] ≤M)) then
{
X[k]=0;
SumOfSub(S, k+1, r-w[k]);
}
}

Hamiltonian Cycle:

- **Def:** 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 & returns to its starting position.
- It is also called the Hamiltonian circuit.
- Hamiltonian circuit is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once.
- A graph possessing a Hamiltonian cycle is said to be Hamiltonian graph.
  Example:

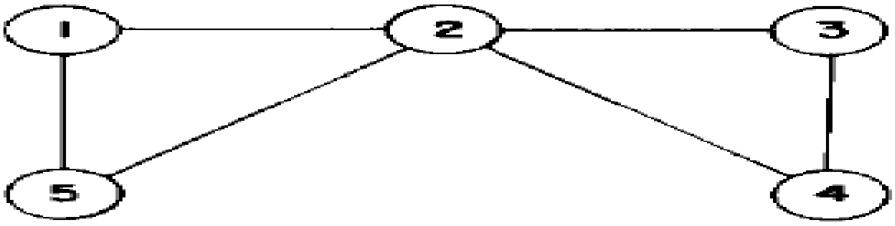


- In graph G, Hamiltonian cycle begins at some vertiex v1 ∈G and the vertices of G are visited in the order $v_1, v_2, \ldots v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \le i \le n$.



g1

The above graph contains Hamiltonian cycle: 1,2,8,7,6,5,4,3,1



The above graph contains no Hamiltonian cycles.

- There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.
- By using backtracking method, it can be possible
  - Backtracking algorithm, that finds all the Hamiltonian cycles in a graph.
  - The graph may be directed or undirected. Only distinct cycles are output.
  - From graph g1 backtracking solution vector= {1, 2, 8, 7, 6, 5, 4, 3, 1}
  - The backtracking solution vector (x_1, x_2, --- x_n)
    x_i i^{th} visited vertex of proposed cycle.
  - By using backtracking we need to determine how to compute the set of possible vertices for x_k if x_1,x_2,x_3---x_{k-1} have already been chosen.
    If k=1 then x1 can be any of the n-vertices.

By using "NextValue" algorithm the recursive backtracking scheme to find all Hamiltoman cycles.

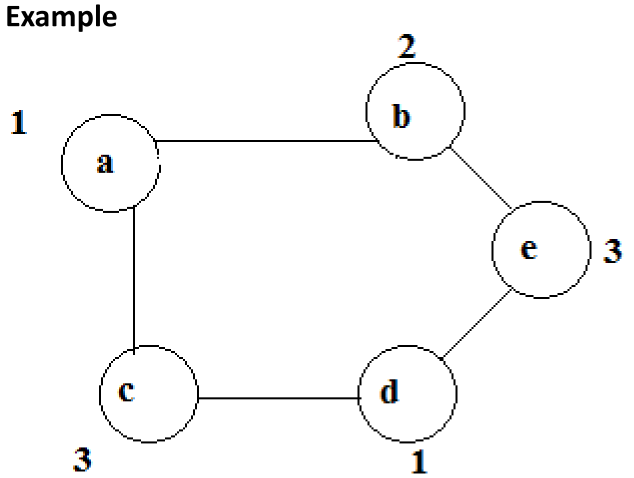
This algorithm is started by 1st initializing the adjacency matrix G[1:n, 1:n] then setting x[2:n] to zero & x[1] to 1, and then executing Hamiltonian (2)

Generating Next Vertex	**Finding all Hamiltonian Cycles**

Algorithm NextValue(k)
{
// x[1: k-1] is path of k-1 distinct vertices.
// if x[k]=0, then no vertex has yet been assigned to x[k]
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
{
For j:=1 to k-1 do if(x[j]=x[k]) then break;
//Check for distinctness
If(j=k) then //if true , then vertex is distinct
If((k<n) or (k=n) and G[x[n], x[1]]≠0))
Then return ;
}
}
Until (false);
}
	Algorithm Hamiltonian(k)
	{
	Repeat{
	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)
	}

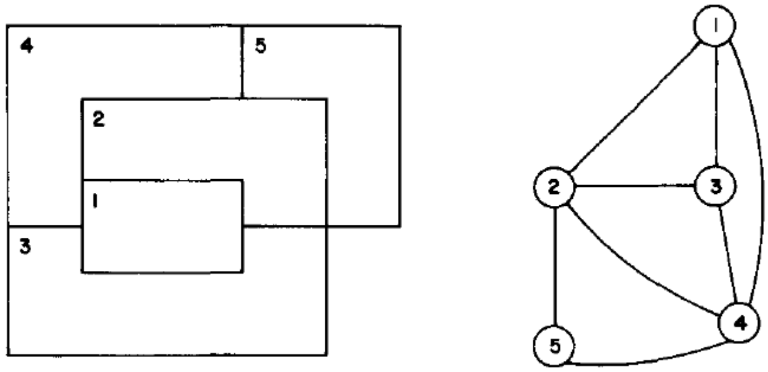
Graph Coloring:

Let G be a undirected graph and 'm' be a given +ve integer. The graph coloring problem is assigning colors to the vertices of an undirected graph with the restriction that no two adjacent vertices are assigned the same color yet only 'm' colors are used.
The optimization version calls for coloring a graph using the minimum number of coloring.
The decision version, known as K-coloring asks whether a graph is colourable using at most k-colors.
Note that, 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 as "**Chromatic number**" of the graph.
Example



- Above graph can be colored with 3 colors 1, 2, & 3.
- The color of each node is indicated next to it.
- 3-colors are needed to color this graph and hence this graph' Chromatic Number is 3.
- A graph is said to be planar iff it can be drawn in a plane (flat) in such a way that no two edges cross each other.
- **M-Colorability decision problem** is the 4-color problem for planar graphs.
- Given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only 4-colors are needed?
- To solve this problem, graphs are very useful, because 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.

    - **Example:**



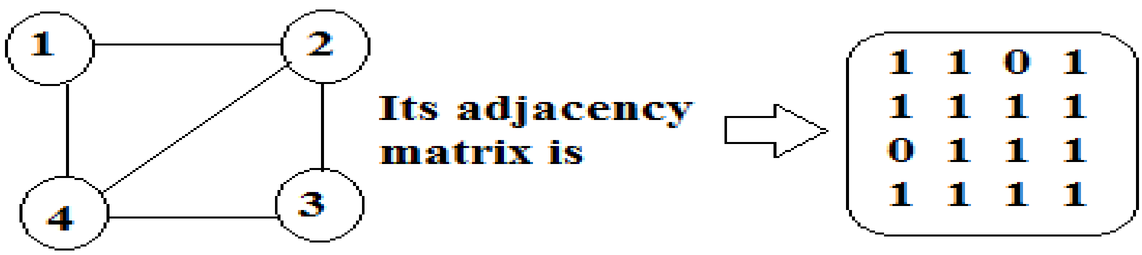
A map and its planar graph representation

The above map requires 4 colors.

- Many years, it was known that 5-colors were required to color this map.
- After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They show that 4-colors are sufficient.

Suppose we represent a graph by its adjacency matrix G[1:n, 1:n]

Ex:

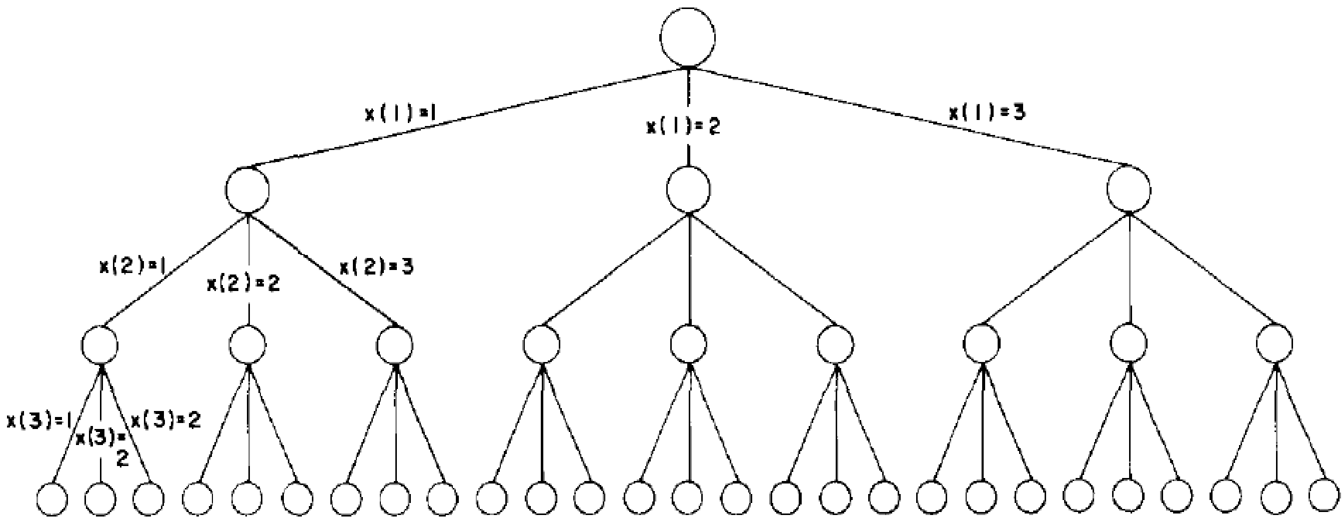


Here G[i, j]=1 if (i, j) is an edge of G, and G[i, j]=0 otherwise.

Colors are represented by the integers 1, 2,---m and the solutions are given by the n-tuple (x1, x2,---xn)

xi= Color of node i.

State Space Tree for n=3 nodes, m=3 colors



State space tree for MCOLORING when $n = 3$ and $m = 3$

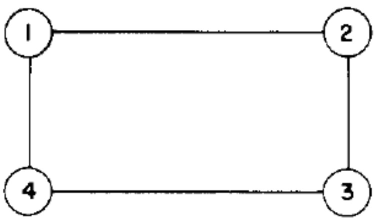
1st node coloured in 3-ways

2^{nd} node coloured in 3-ways
3^{rd} node coloured in 3-ways
So we can colour in the graph in 27 possibilities of colouring.

Finding all m-coloring of a graph	**Getting next color**

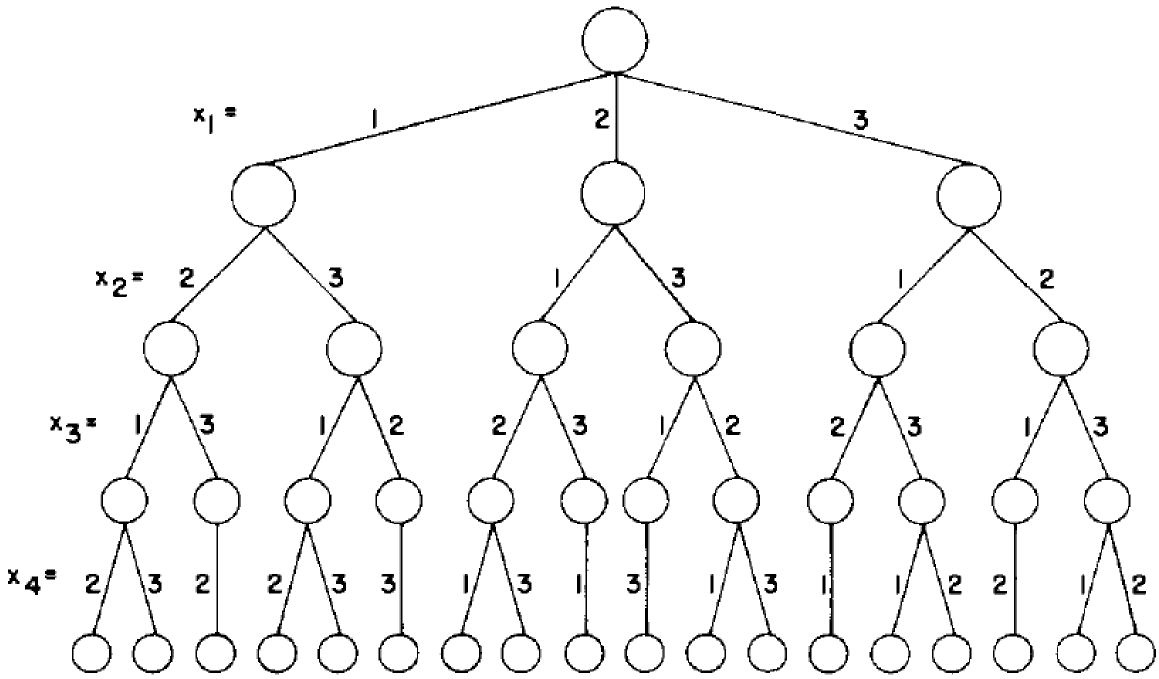
Algorithm mColoring(k){
// g(1:n, 1:n) boolean adjacency matrix.
// kindex (node) of the next vertex to color.
repeat{
nextvalue(k); // assign to x[k] a legal color.
if(x[k]=0) then return; // no new color possible
if(k=n) then write(x[1: n];
else mcoloring(k+1);
}
until(false)
}
	Algorithm NextValue(k){
	//x[1],x[2],---x[k-1] have been assigned integer values in the range [1, m]
	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
	{
	if ((g[k,j]≠0) and (x[k]=x[j]))
	then break;
	}
	if(j=n+1) then return; //new color found
	} until(false)
	}

Previous paper example:



Adjacency matrix is

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$



A 4 node graph and all possible 3 colorings

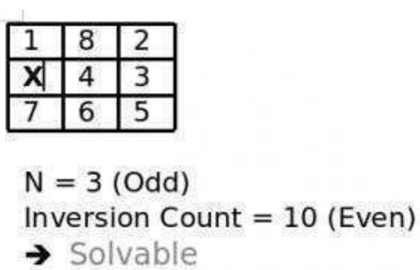
15 puzzle Problem:

The "15 puzzle" is a sliding square puzzle commonly (but incorrectly) attributed to Sam Loyd
The 15 puzzle consists of 15 squares numbered from 1 to 15 that are placed in a 4×4 box leaving one position out of the 16 empty. The goal is to reposition the squares from a given arbitrary starting arrangement by sliding them one at a time into the final configuration .
In general, for a given grid of width N, we can find out check if a N*N – 1 puzzle is solvable or not by following below simple rules :

- If N is odd, then puzzle instance is solvable if number of inversions is even in the input state.
- If N is even, puzzle instance is solvable if
  - the blank is on an even row counting from the bottom (second-last, fourth-last, etc.) and number of inversions is odd.
  - the blank is on an odd row counting from the bottom (last, third-last, fifth-last, etc.) and number of inversions is even.
- For all other cases, the puzzle instance is not solvable.

What is an inversion here?

If we assume the tiles written out in a single row (1D Array) instead of being spread in N-rows (2D Array), a pair of tiles (a, b) form an inversion if a appears before b but a > b.



Example:

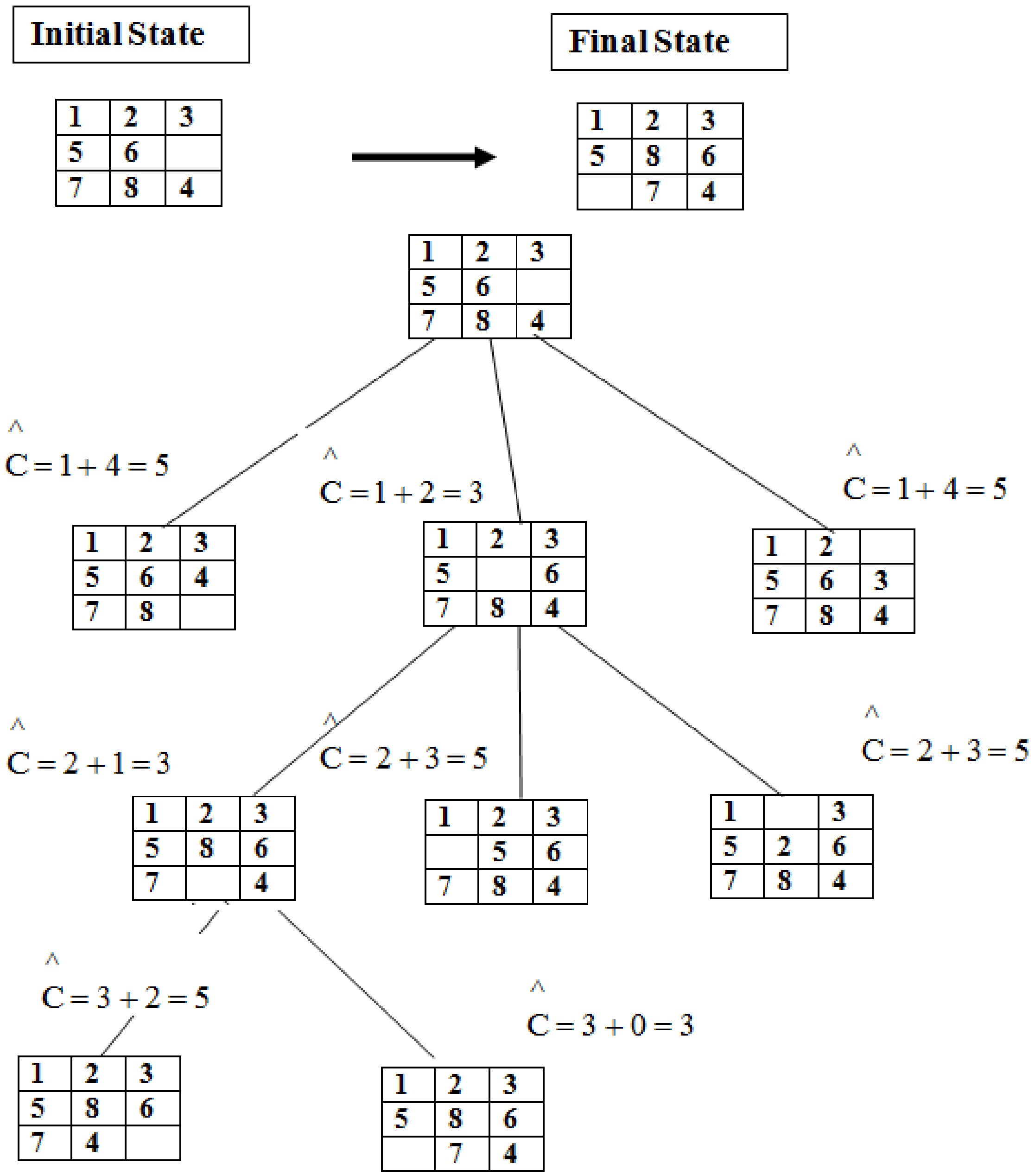
8-puzzle

Cost function: $\hat{C} = g(x) + h(x)$

where h(x) = the number of misplaced tiles

and g(x) = the number of moves so far

Assumption: move one tile in any direction cost 1.



LEAST COST SEARCH

A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(\cdot)$
∗ Such a search strategy is called an LC -search (Least Cost search)
∗ Both BFS and DFS are special cases of LC -search
∗ In BFS , we use $\hat{g}(x) \equiv 0$ and $f(h(x))$ as the level of node x.LC search generates nodes by level
∗ In DFS , we use $f(h(x)) \equiv 0$ and $\hat{g}(x) \geq \hat{g}(y)$ whenever y is a child of x
– An LC -search coupled with bounding functions is called an LC branch-and-bound search

Cost function:

$\hat{c}(x) = f(h(x)) + \hat{g}(x)$

where h(x) is the cost of reaching x from root
$\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer from node x

- If x is an answer node, c(x) is the cost of reaching x from the root of state space tree
- If x is not an answer node, $c(x) = \infty$, provided the subtree x contains no answer node
- If subtree x contains an answer node, c(x) is the cost of a minimum cost answer node in subtree x