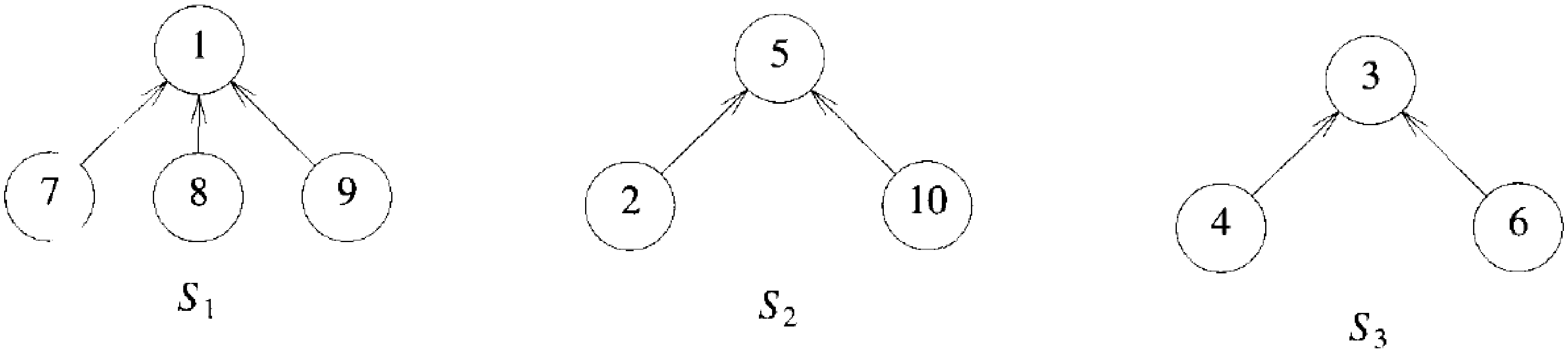
Disjoint Sets and Union and Find Operations

Disjoint Sets - Introduction

- A tree is used to represent each set and the root to name a set
- Each node points upwards to its parent
- Two sets S1 and S2 are said to be disjoint if $S1-S2 = \Phi$, i.e there is no common elements in both S1 and S2
- A Collection of disjoint sets is called a disjoint set forest
- An array can be used to store parent of each element

Disjoint Sets - Introduction

- Example: When n=10, the element can be portioned into three disjoint sets, S1 ={ 1,7,8,9 }, S2 = { 2,5,10 }and S3 = { 3,4,6}

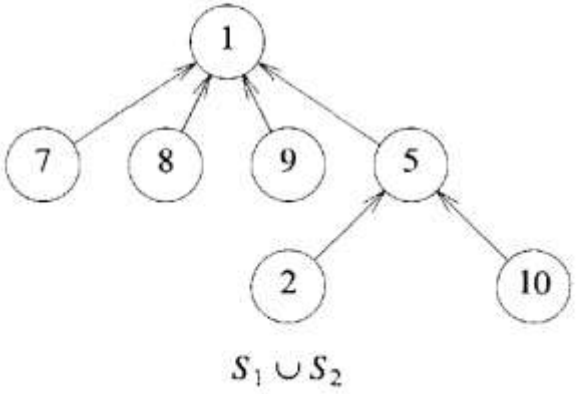


Disjoint Sets - Operations

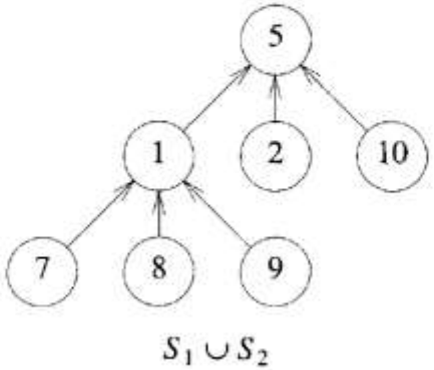
- Two important operations performed on disjoint sets
  - **Union:** If S_i and S_j are two disjoint sets, then their union SiU Sj= all elements x such that x is in S_i or S_j.Thus,,$S_i US_2$ = {1,7, 8,9,2,5,10}Since we have assumed that all sets are disjoint, we can assume that following the union of S_i and S_j, the sets S_i and S_j do not exist independently; that is, they are replacedby $S_i U\ S_j$ in the collection of sets.
  - **Find :** Given the element i, find the set containing i. Thus, 4 is in set S3,and 9 is in set S1.

Disjoint Sets – Operation Union

- IF S1 and S2 are two disjoint sets, their union S1 U S2 is a set of all elements x such that x is in either S1 or S2
- As the sets should be disjoint S1 U S2 replace S1 and S2 which no longer exist
- Union is achieved by simply making one of the trees as a subtree of other i.e to set parent field of one of the roots of the trees to other root



or



Possible representation of S1 U S2

Disjoint Sets – Representation

- An array can be used to store parent of each element
- The ith element of this array represents the tree node that contains the element i and it gives the parent of the element
- Root node has a parent -1
- An array P[1:n] can be taken for all n elements in the forest
- Element at the root node is taken as the name of the set

i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	5	-1	3	-1	3	1	1	1	5

Array representation of S_1, S_2, and S_3

Disjoint Sets – Union and Find Algorithms

- **Union(i, j):** We passing two trees with roots i and j. Adopting the convention that the first tree becomes subtree of the second the statement p[i] :=j;
- **Find(i) :** by following the indices, starting at i until we reach a node with parent value -1.For example, Find(6) start sat 6 and then moves to 6'sparent,3. Since p[3] is negative, we have reached the root

```
Algorithm SimpleUnion(i, j)
{
    p[i] := j;
}

Algorithm SimpleFind(i)
{
    while (p[i] ≥ 0) do i := p[i];
    return i;
}
```

Disjoint Sets – Union and Find Algorithms

- In a worst case scenario SimpleUnion() and SimpleFind() perform badly. Suppose we start with the single element sets {1}, {2}, {3} … {n}, then execute the following sequence of union and find operations

- Union(1,2), Union(2,3), Union(3,4). …., Union(n-1,n)
- Find(1), Find(2), Find(3), …. Find(n)

- The total time needed to process the n finds is $O(\sum_{i=1}^{n} i) = O(n^2)$.

Disjoint Sets – Weighted Union

- Simple Union leads to high time complexity in some cases
- Weighted union is a modified union algorithm with weighting rule
- Widely used to analyze the time complexity of an algorithm is average case
- Weighted union deals with making the smaller tree a subtree of the large
- If the no.of nodes in the tree with root i is less the no.of nodes in the tree with root j, then make j the parent of i, otherwise make i the parent of j
- Count of nodes can be placed as a negative number in the P[i] value of the root i.

Disjoint Sets – Weighted Union Algorithm

Algorithm WeightedUnion(i, j)
// Union sets with roots i and j, $i \neq j$, using the
// weighting rule. $p[i] = -count[i]$ and $p[j] = -count[j]$.
{
    $temp := p[i] + p[j]$;
    if $(p[i] > p[j])$ **then**
    { // i has fewer nodes.
        $p[i] := j$; $p[j] := temp$;
    }
    else
    { // j has fewer or equal nodes.
        $p[j] := i$; $p[i] := temp$;
    }
}

Disjoint Sets – CollapsingFind Algorithm

Collapsing Rule: If j is a node on the path from i to its root and p[i] ≠ root[i] ,then set p[j] to root[i].

```
Algorithm CollapsingFind(i)
// Find the root of the tree containing element i. Use the
// collapsing rule to collapse all nodes from i to the root.
{
    r := i;
    while (p[r] > 0) do r := p[r]; // Find the root.
    while (i ≠ r) do  // Collapse nodes from i to root r.
    {
        s := p[i]; p[i] := r; i := s;
    }
    return r;
}
```

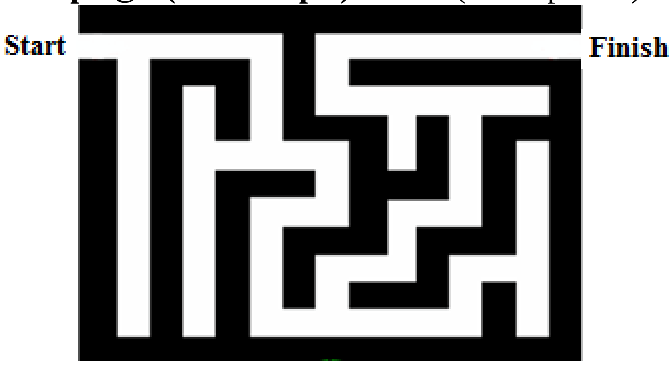
UNIT IV:
Backtracking: General method, applications-n-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.
Branch and Bound: General method, applications - Travelling sales person problem,0/1 knapsack problem- LC Branch and Bound solution, FIFO Branch and Bound solution.

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 (net example) : Maze (a tour puzzle)



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

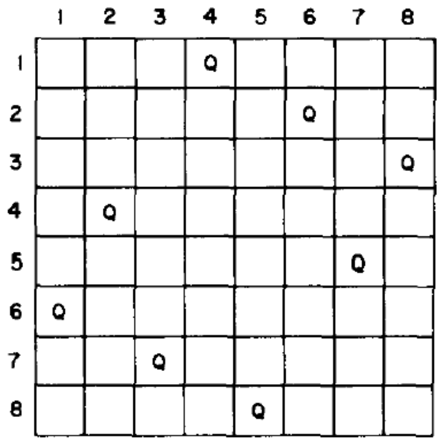
- ➢ In maze, at each intersection, you have to decide between 3 or fewer choices:
  - ✓ Go straight
  - ✓ Go left
  - ✓ Go right
- ➢ You don't have enough information to choose correctly
- ➢ Each choice leads to another set of choices.
- ➢ One or more sequences of choices may or may not lead to a solution.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

Example@ 2 (text book):
Sorting the array of integers in a[1:n] is a problem whose solution is expressible by an n-tuple
$x_i \rightarrow$ 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 $1 \leq i \leq n$
$S_i \rightarrow$ is finite and includes the integers 1 through n.
$m_i \rightarrow$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.

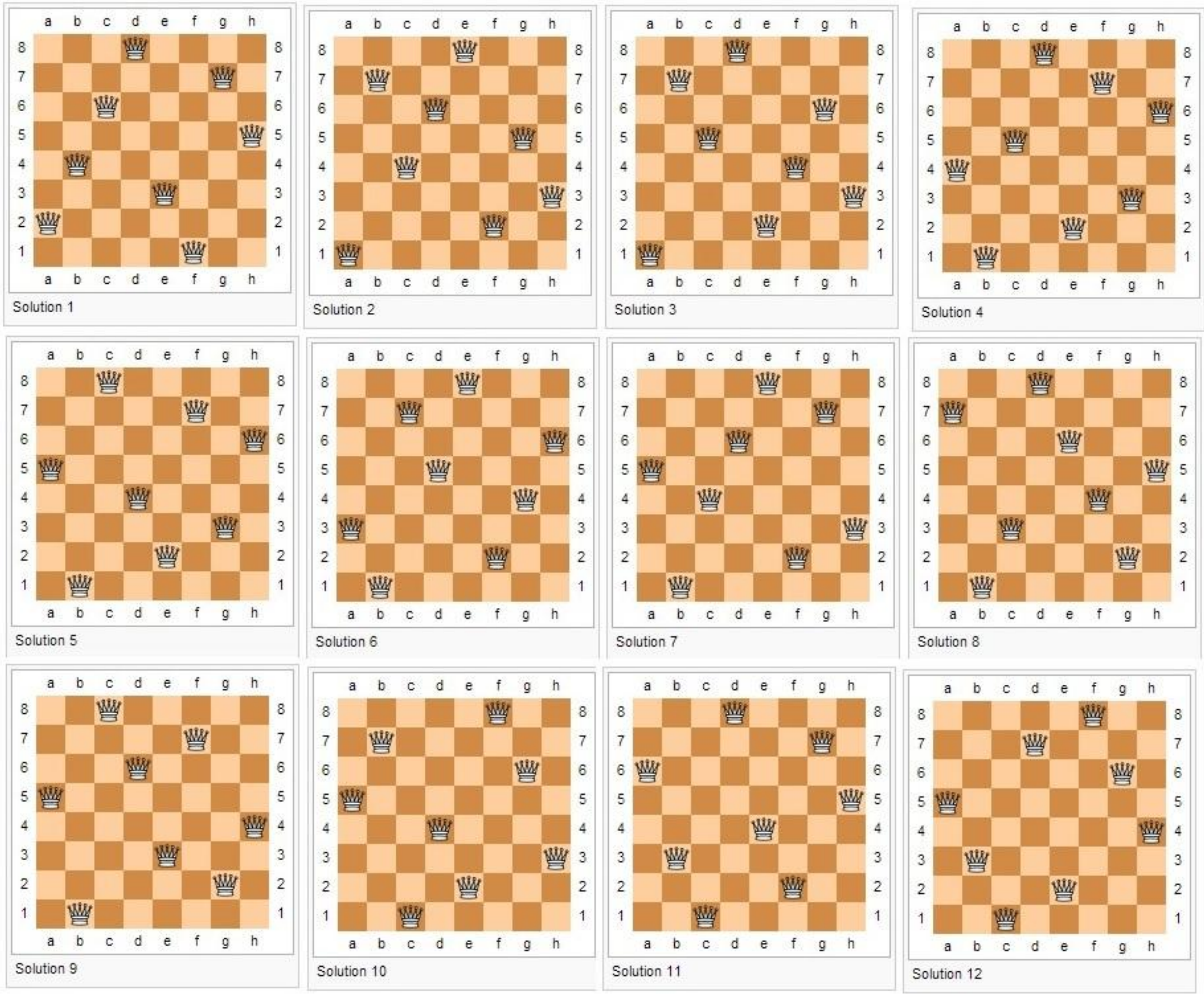
N-Queens Problem:
It is a classic combinatorial problem. The eight queen's puzzle is the problem of placing eight queens puzzle is the problem of placing eight queens on an 8×8 chessboard so that no two queens attack each other. That is so that no two of them are on the same row, column, or diagonal.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



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}, $1 \leq i \leq 8$
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)

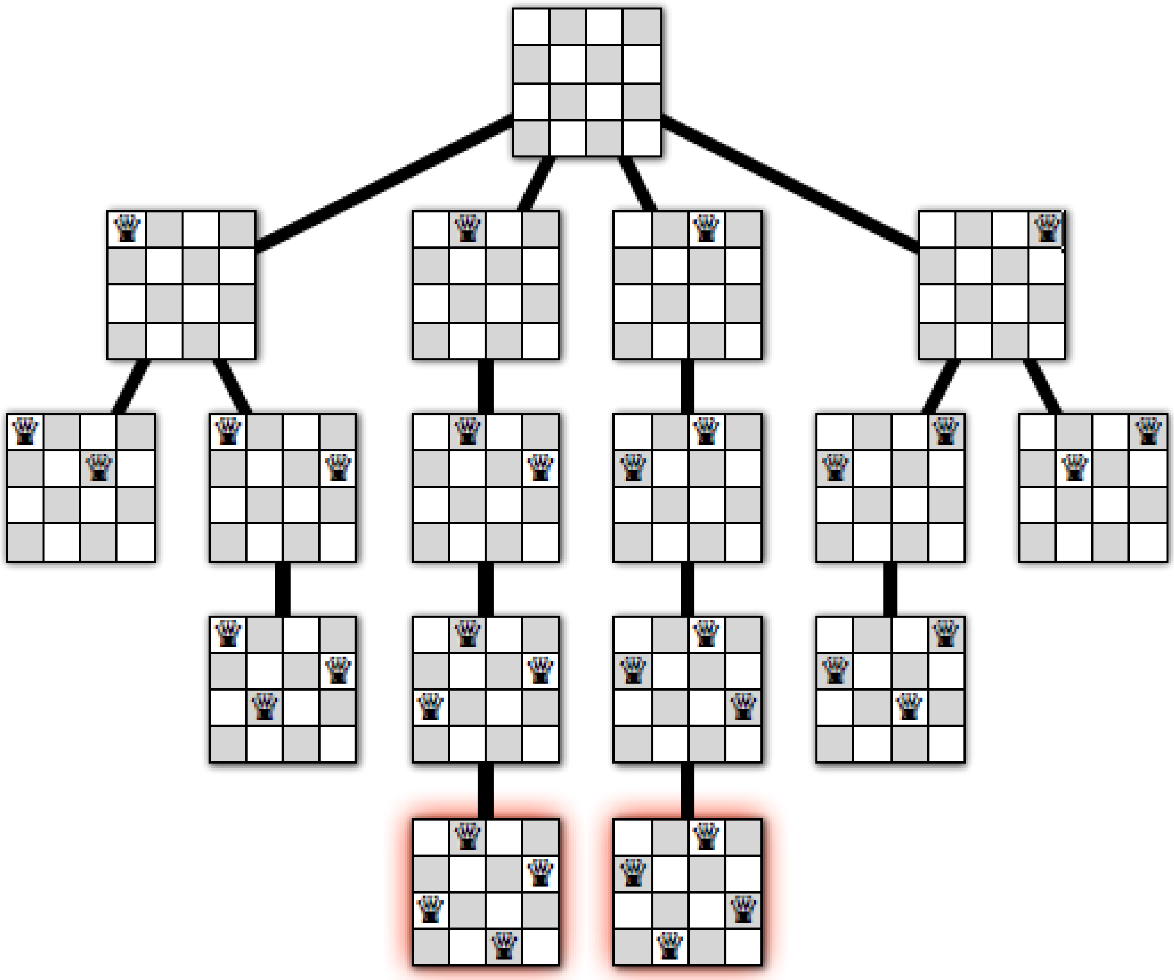
In the same way for n-queens are to be placed on an n×n chessboard, the solution space consists of all n! Permutations of n-tuples (1,2,----n).



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 $1 \leq i \leq n$, & 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) $1 \leq k \leq n$, different solutions may have different sized tuples.

- Explicit constraints requires $x_i \in \{j \mid j \text{ is an integer } 1 \leq j \leq n\}$
- 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 $x_i < x_{i+1}$ $1 \leq i \leq k$

$W_i \rightarrow$ 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.

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

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.

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

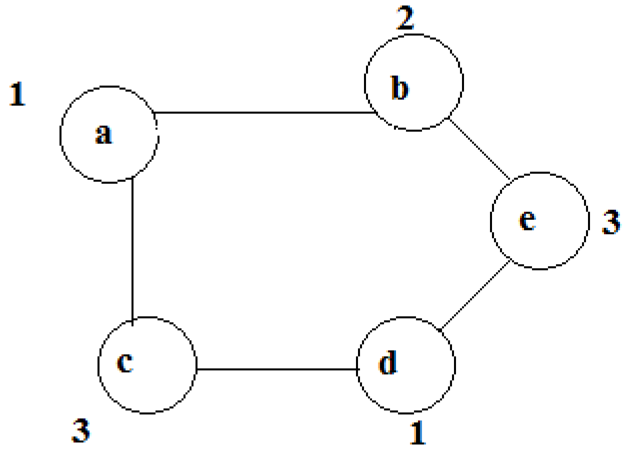
$$s = \sum_{j=1}^{k-1} W(j)X(j). \quad \text{and} \quad r = \sum_{j=k}^{n} W(j)$$

Recursive backtracking algorithm for sum of subsets problem

Algorithm SumOfSub(s, k, r)
{
$$// s = \sum_{j=1}^{k-1} W(j)X(j). \quad \text{and} \quad r = \sum_{j=k}^{n} W(j)$$
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]);
}
}

Graph Coloring:

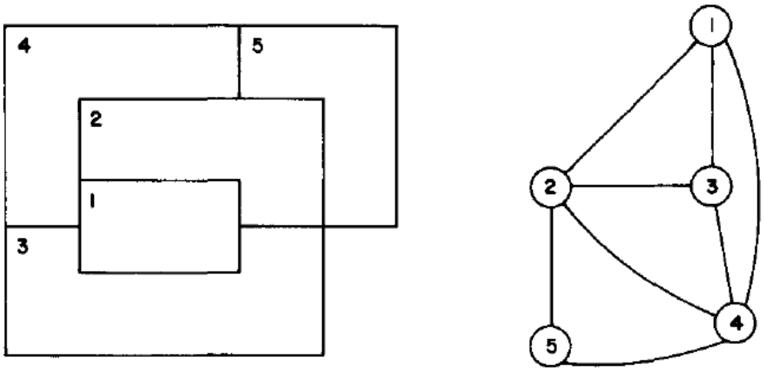
Let G be a undirected graph and 'm' be a given +ve integer. The graph coloring problem is assigning colors to the vertices of an undirected graph with the restriction that no two adjacent vertices are assigned the same color yet only 'm' colors are used.
The optimization version calls for coloring a 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:**



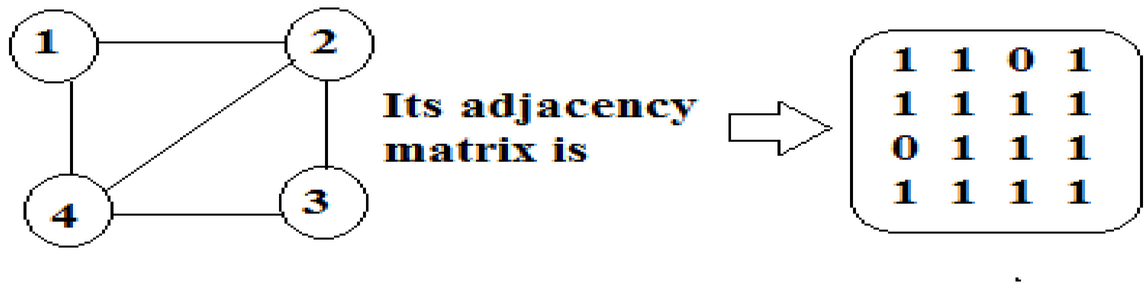
A map and its planar graph representation

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.

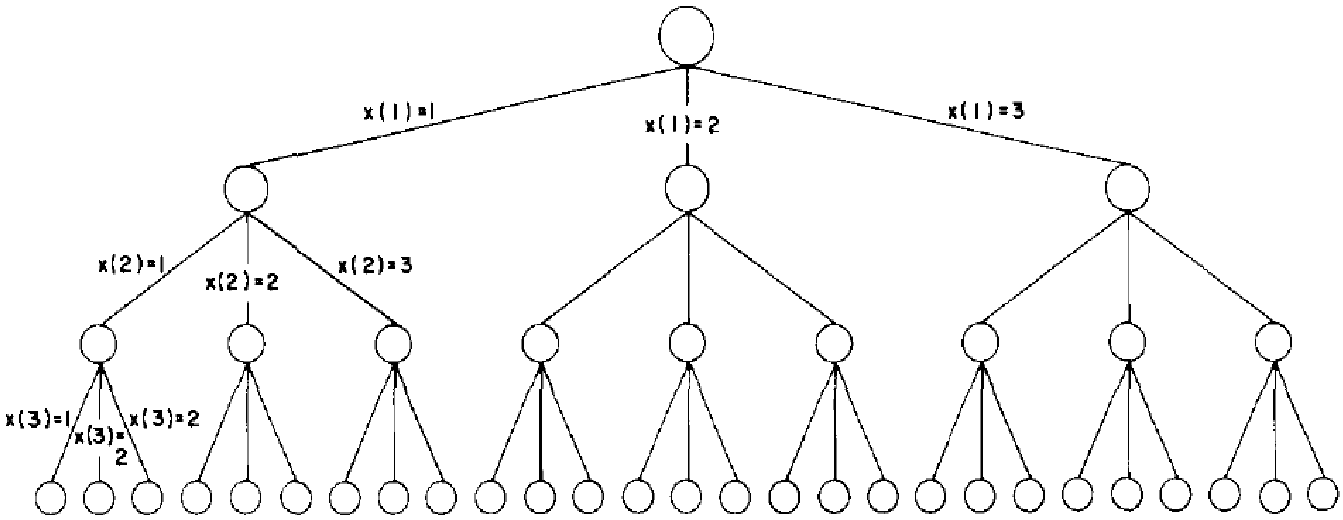
Suppose we represent a graph by its adjacency matrix G[1:n, 1:n]

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



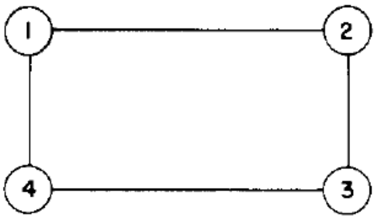
State space tree for MCOLORING when $n = 3$ and $m = 3$

1st node coloured in 3-ways
2nd node coloured in 3-ways
3rd 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

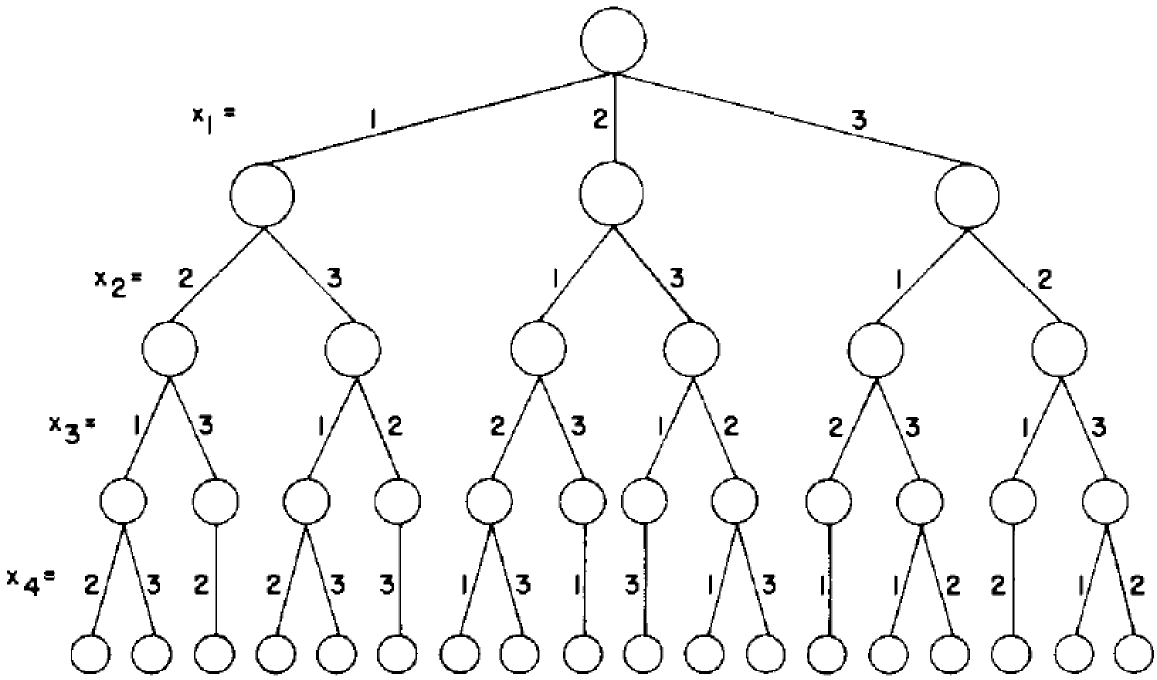
Algorithm mColoring(k){
// g(1:n, 1:n)→ boolean adjacency matrix.
// k→index (node) of the next vertex to color.
repeat{
nextvalue(k); // assign to x[k] a legal color.
if(x[k]=0) then return; // no new color possible
if(k=n) then write(x[1: n];
else mcoloring(k+1);
}
until(false)
}
	Algorithm NextValue(k){
	//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

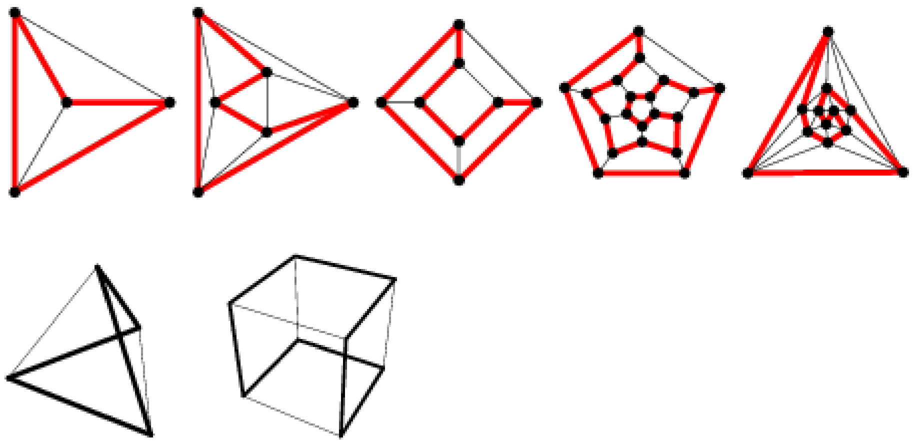
$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$



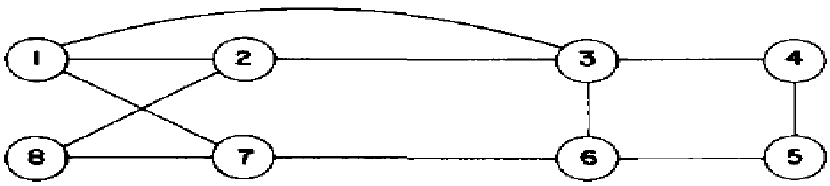
A 4 node graph and all possible 3 colorings

Hamiltonian Cycles:

- **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:

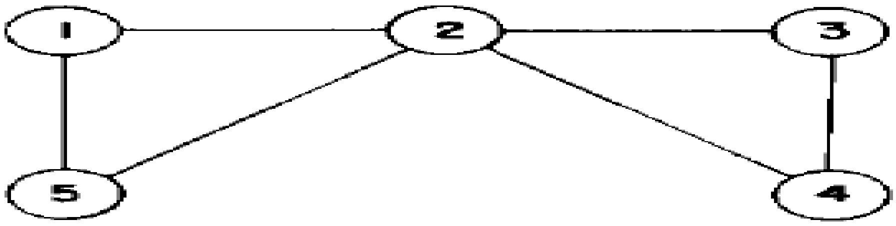


- In graph G, Hamiltonian cycle begins at some vertiex v1 $\in$ G and the vertices of G are visited in the order $v_1, v_2, --- v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \le i \le n$.



→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)
	}

Branch & Bound

Branch & Bound (B & B) is general algorithm (or Systematic method) for finding optimal solution of various optimization problems, especially in discrete and combinatorial optimization.

- The B&B strategy is very similar to backtracking in that a state space tree is used to solve a problem.
- The differences are that the B&B method
- ✓ Does not limit us to any particular way of traversing the tree.
- ✓ It is used only for optimization problem
- ✓ It is applicable to a wide variety of discrete combinatorial problem.
- B&B is rather general optimization technique that applies where the greedy method & dynamic programming fail.
- It is much slower, indeed (truly), it often (rapidly) leads to exponential time complexities in the worst case.
- The term B&B refers to all state space search methods in which all children of the "E-node" are generated before any other "live node" can become the "E-node"
- ✓ **Live node→** is a node that has been generated but whose children have not yet been generated.
- ✓ **E-node→**is a live node whose children are currently being explored.