

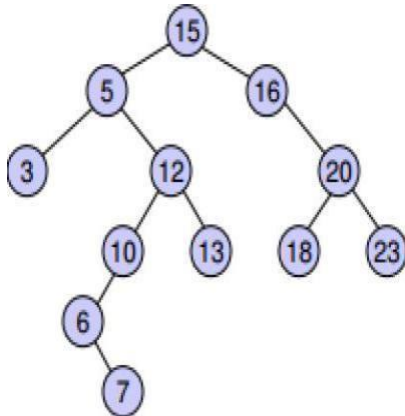
UNIT II

Disjoint Sets: Disjoint set operations, union and find algorithms

Backtracking: General method, applications, n-queen's problem, sum of subsets problem, graph coloring

PART - I – DISJOINT SETS

Efficient non recursive tree traversal algorithms



in-order: (left, root, right)
3,5,6,7,10,12,13
15, 16, 18, 20, 23

pre-order: (root, left, right)
15, 5, 3, 12, 10, 6, 7,
13, 16, 20, 18, 23

post-order: (left, right, root)
3, 7, 6, 10, 13, 12, 5,
18,23,20,16,15

Non recursive Inorder traversal algorithm

1. Start from the root. let's it is current.
2. If current is not NULL. push the node on to stack.
3. Move to left child of current and go to step 2.
4. If current is NULL, and stack is not empty, pop node from the stack.
5. Print the node value and change current to right child of current.
6. Go to step 2.

So we go on traversing all left node. as we visit the node. we will put that node into stack remember need to visit parent after the child and as We will encounter parent first when start from root. it's case for LIFO :) and hence the stack). Once we reach NULL node. we will take the node at the top of the stack. last node which we visited. Print it.

Check if there is right child to that node. If yes, move right child to stack and again start traversing left child node and put them on to stack. Once we have traversed all node. our stack will be empty.

Non recursive postorder traversal algorithm

Left node. right node and last parent node.

1.1 Create an empty stack

Do Following while root is not NULL

- a) Push root's right child and then root to stack.

b) Set root as root's left child.

Pop an item from stack and set it as root.

a) If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.

Ia) Else print root's data and set root as NULL.

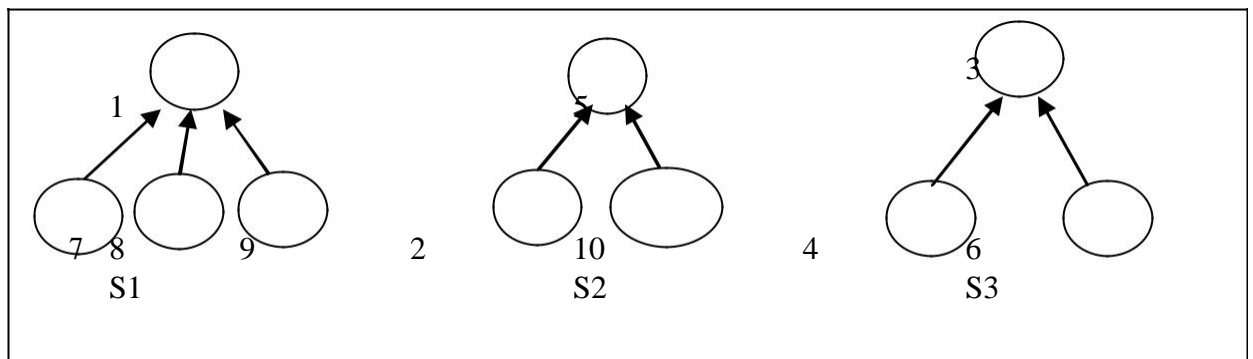
Repeat steps 2.1 and 2.2 while stack is not empty.

Disjoint Sets: If S_i and S_j , $i \neq j$ are two sets, then there is no element that is in both S_i and S_j .

For example: $n=10$ elements can be partitioned into three disjoint sets,

$S_1 = \{1, 7, 8, 9\}$
 $S_2 = \{2, 5, 10\}$
 $S_3 = \{3, 4, 6\}$

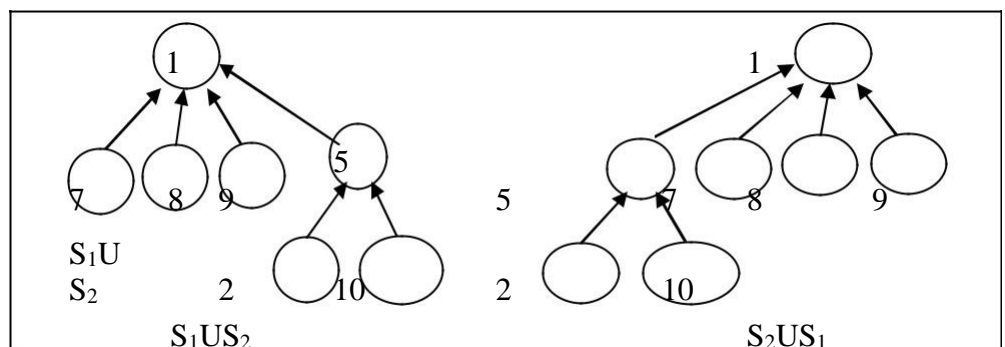
Tree representation of sets:



DISJOINT SET OPERATIONS:

- Disjoint set Union
- Find(i)

DISJOINT SET UNION: Means Combination of two disjoint sets elements. Form above example $S_1 \cup S_2 = \{1, 7, 8, 9, 5, 2, 10\}$
 For $S_1 \cup S_2$ tree representation, simply make one of the tree is a subtree of the other.



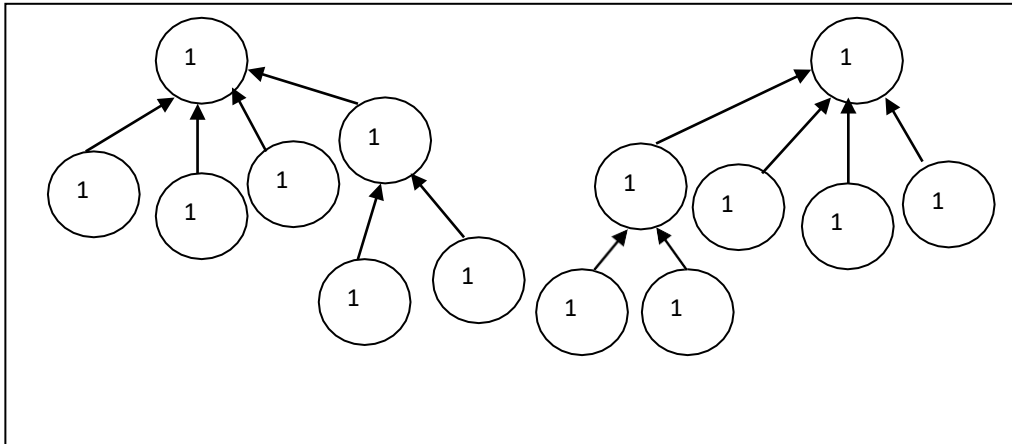
Find: Given element i , find the set containing i .

Form above example:

Find(4) \square S_3

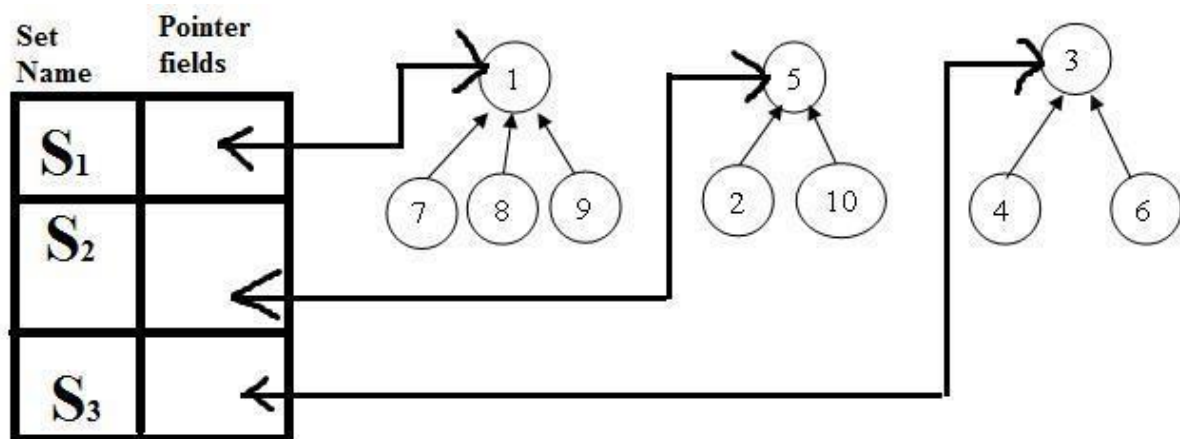
Find(1) \square S_1

Find(10) \square S_2



Data representation of sets:

Tress can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set.



For presenting the union and find algorithms, we ignore the set names and identify sets just by the roots of the trees representing them.

For example: if we determine that element 'i' is in a tree with root 'j' has a pointer to entry 'k' in the set name table, then the set name is just **name[k]**

For unite (**adding or combine**) to a particular set we use FindPointer function. **Example:** If you wish to unite to S_i and S_j then we wish to unite the tree with roots FindPointer (S_i) and FindPointer (S_j)

FindPointer[□] is a function that takes a set name and determines the root of the tree that represents it.

For determining operations:

Find(i)[□] 1st determine the root of the tree and find its pointer to entry in setname table.

Union(i, j)[□] Means union of two trees whose roots are i and j.

If set contains numbers 1 through n, we represents tree node **P[1:n]**.

n[□]Maximum number of elements.

Each node represent in array

i	1	2	3	4	5	6	7	8	9	10
P	-1	5	-1	3	-1	3	1	1	1	5

Find(i) by following the indices, starting at i until we reach a node with parent value -1.
 Example: Find(6) start at 6 and then moves to 6's parent. Since P[3] is negative, we reached the root.

Algorithm for finding Union(i, j):	Algorithm for find(i)
Algorithm Simple union(i, j) { P[i]:=j; // Accomplishes the union }	Algorithm SimpleFind(i) { While(P[i]≥0) do i:=P[i]; return i; }

If n numbers of roots are there then the above algorithms are not useful for union and find.

For union of n trees □ Union(1,2), Union(2,3), Union(3,4),.....Union(n-1,n).

For Find i in n trees □ Find(1), Find(2),....Find(n).

Time taken for the union (simple union) is □ O(1) (constant).

For the n-1 unions □ O(n).

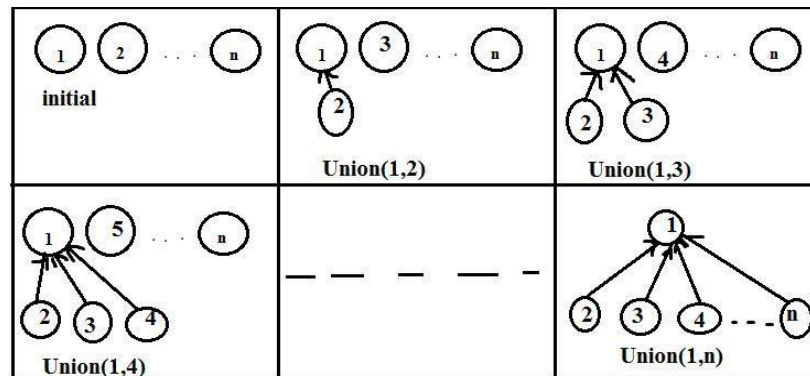
Time taken for the find for an element at level i of a tree → O(i).

is For n finds → O(n²).

To improve the performance of our union and find algorithms by avoiding the creation of degenerate trees. For this we use a weighting rule for union(i, j)

Weighting rule for Union(i, j):

If the number of nodes in the tree with root 'i' is less than the tree with root 'j', then make 'j' the parent of 'i'; otherwise make 'i' the parent of 'j'.



Tree obtained using the weighting rule

Algorithm for weightedUnion(i, j)

```

Algorithm WeightedUnion(i,j)
//Union sets with roots i and j, i≠j
// 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;
}
}

```

For implementing the weighting rule, we need to know how many nodes there are in every tree.

For this we maintain a count field in the root of every tree.

i root node

count[i] = number of nodes in the tree.

Time required for this above algorithm is $O(1)$ + time for remaining unchanged is determined by using **Lemma**.

Lemma:-Let T be a tree with **m** nodes created as a result of a sequence of unions each performed using WeightedUnion. The height of T is no greater than $\lceil \log_2 m \rceil + 1$.

Collapsing rule: If 'j' is a node on the path from 'i' to its root and $p[i] \neq \text{root}[i]$, then set $p[j]$ to $\text{root}[i]$.

Algorithm for Collapsing find.

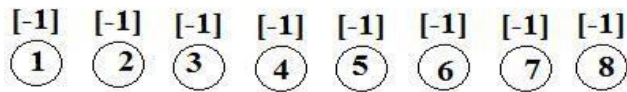
```

Algorithm CollapsingFind(i)
//Find the root of the tree containing element i.
//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;
}

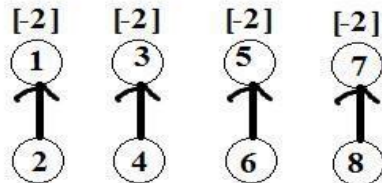
```

Collapsing find algorithm is used to perform find operation on the tree created by WeightedUnion.

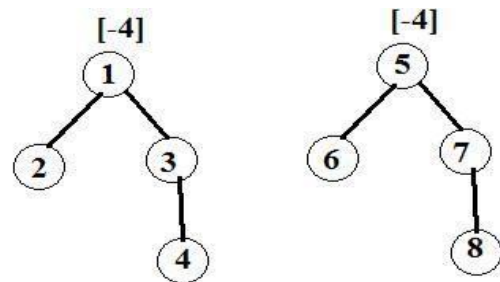
For example: Tree created by using WeightedUnion



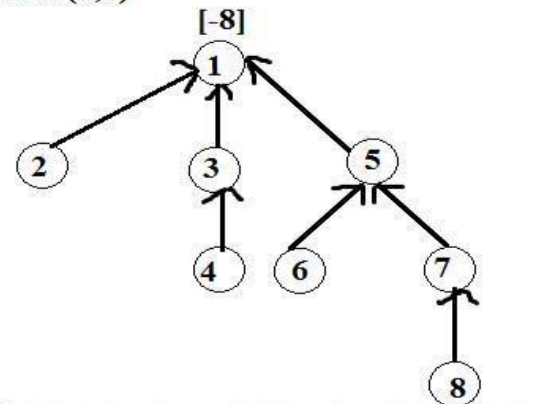
(a) initial height -1 tree



(b) Height -2 trees following Union(1,2),(3,4),(5,6),(7,8)



(c) Height -3 trees following Union (1,3) and (5,7)



(d) Height -4 tree Following Union(1,5)

Now process the following eight finds: Find(8), Find(8),...Find(8)
If SimpleFind is used, each Find(8) requires going up three parent link fields for a total of 24 moves to process all eight finds.

When CollapsingFind is used the first Find(8) requires going up three links and then resetting two links. Total 13 moves requires for process all eight finds.

PART – II - BACKTRACKING

GENERAL METHOD:

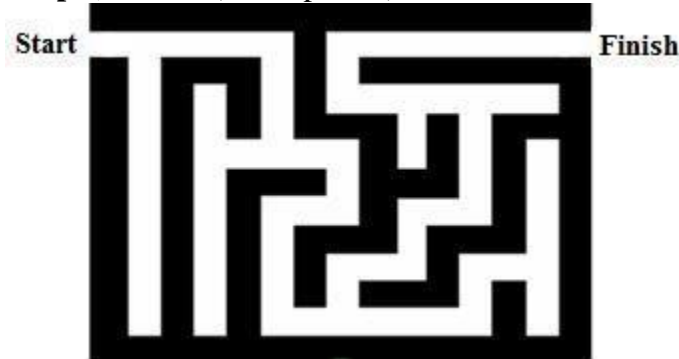
Many problems are difficult to solve algorithmically. Backtracking makes it possible to solve at least some large instances of difficult combinatorial problems.

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 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 .

$m=m_1m_2m_3\cdots m_n$ 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: $\mathbf{x}_i \geq \mathbf{0}$ or $S_i = \{\text{all non negative real numbers}\}$

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

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

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 X_i must relate to each other.

APPLICATIONS OF BACKTRACKING:

- N Queens Problem
- Sum of subsets problem
- Graph coloring

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 $n \times n$ 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'

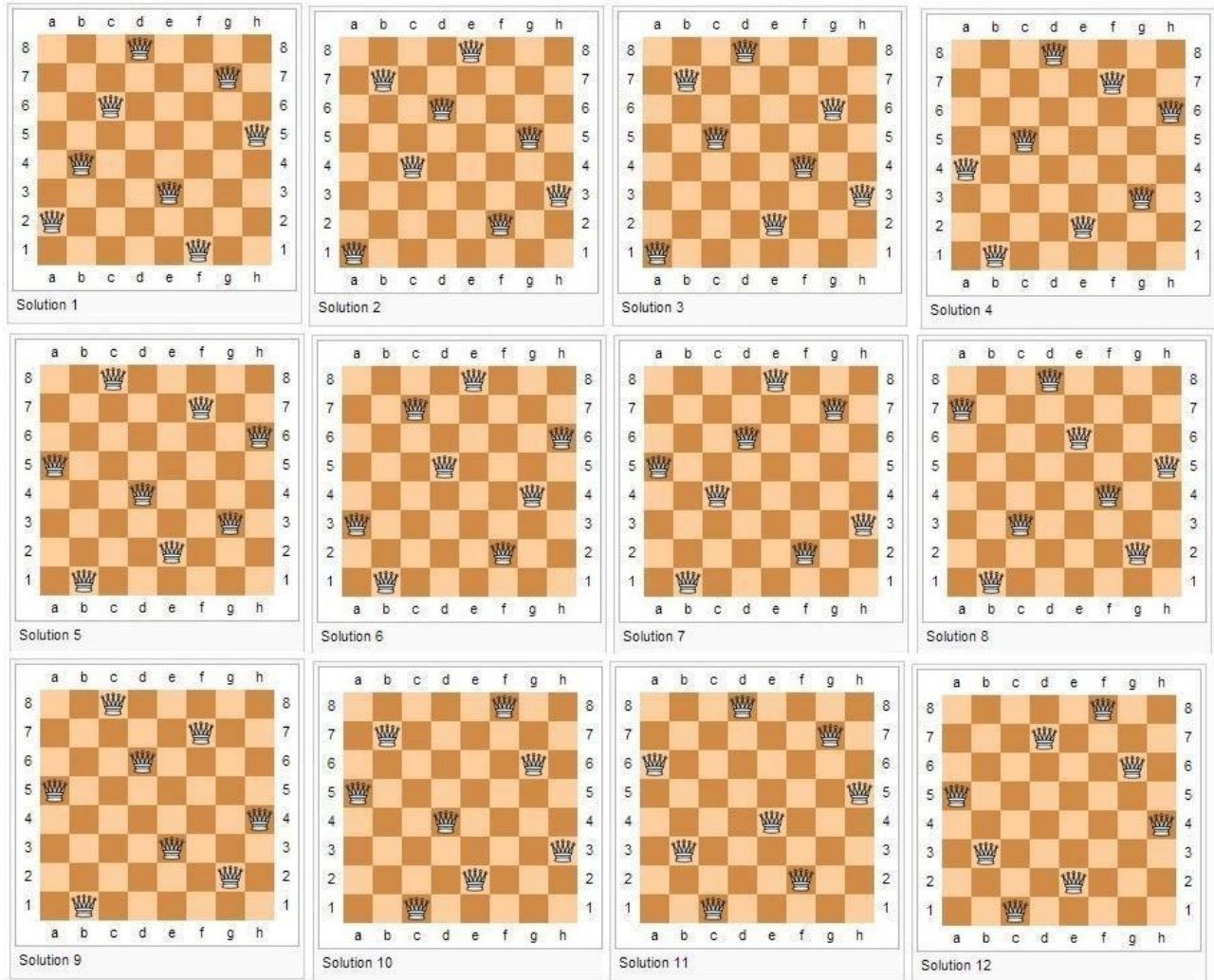
All solutions to the 8-queens problem can therefore be represented as s-tuples $(x_1, x_2, x_3, \dots, x_8)$ where x_i is the column on which queen 'i' is placed
 $s_i \in \{1, 2, 3, 4, 5, 6, 7, 8\}, 1 \leq i \leq 8$

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.

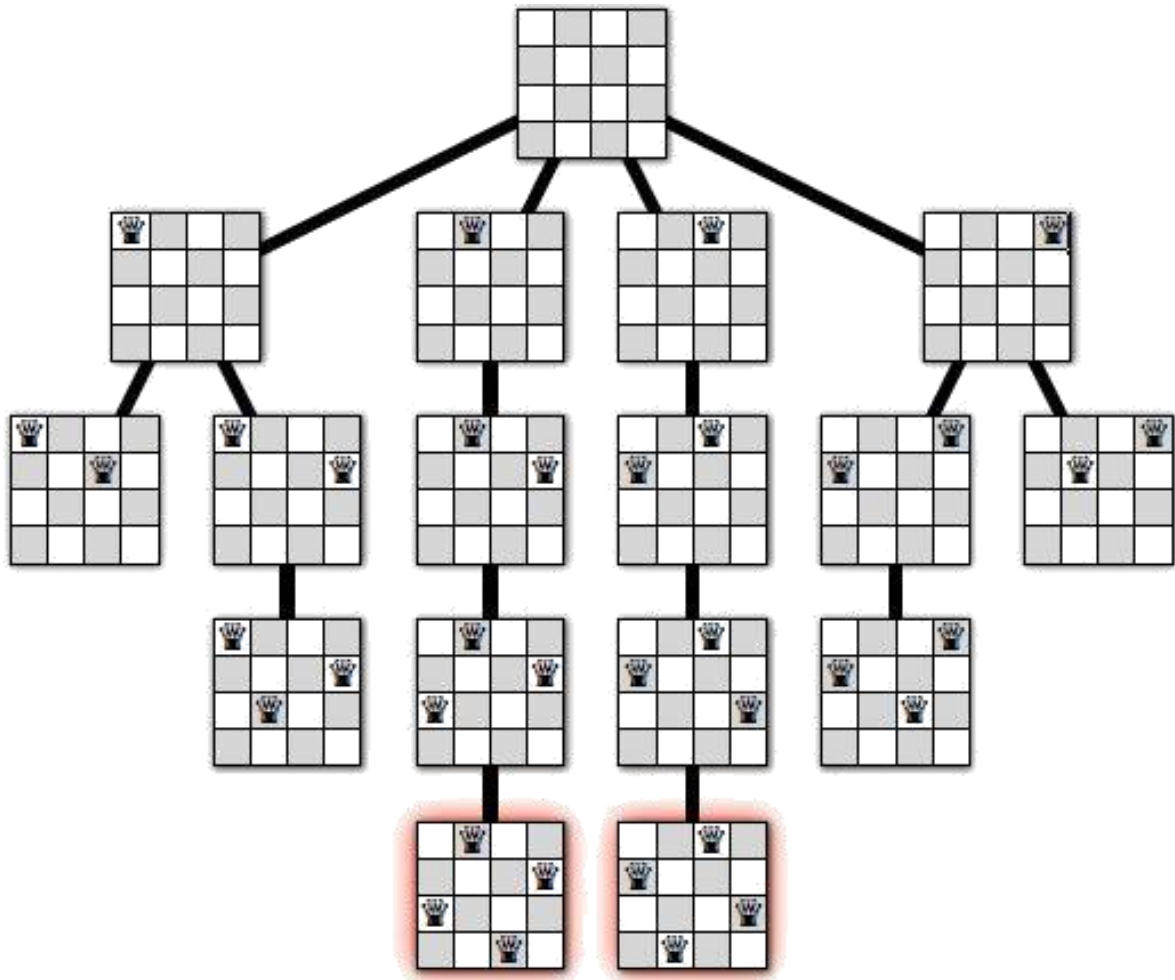
By these two constraints the size of solution space reduces from 8^8 tuples to $8!$ Tuples. For example $s_i(4, 6, 8, 2, 7, 1, 3, 5)$

In the same way for n-queens are to be placed on an $n \times n$ 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
<p>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 }</p>	<p>Algorithm NQueens(k, n) // its prints all possible placements of n-queens on an $n \times n$ 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); } }}}</p>



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 $(x_1, x_2, x_3 \dots x_k)$ $1 \leq k \leq n$, different solutions may have different sized tuples.

- Explicit constraints requires $x_i \in \{j / j \text{ is an integer } 1 \leq j \leq n\}$
- 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 □ 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 w_i is included or not.

If $X_i=1$ then w_i is chosen.

If $X_i=0$ then w_i 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 specifies that $x_1, x_2, x_3, \dots, x_k$ 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.

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

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

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 $(S+w[k] + w[k+1] \leq M)$

Then SumOfSub($S+w[k], k+1, r-w[k]$);

if $((S+r - w[k] \geq M) \text{ and } (S+w[k+1] \leq M))$ then

{

$X[k]=0$;

SumOfSub($S, k+1, r-w[k]$);

}

}

GRAPH COLORING:

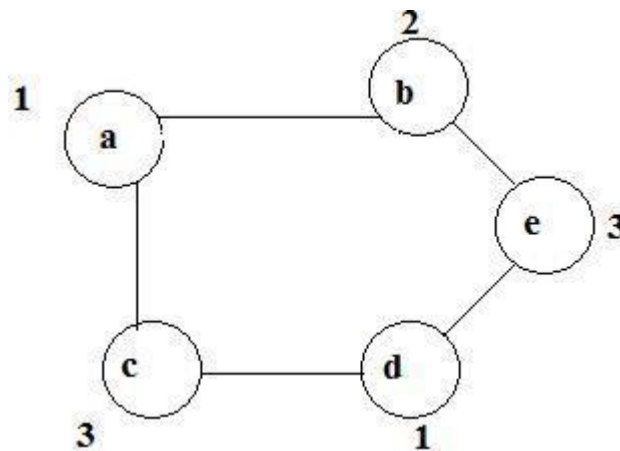
Let G be an undirected graph and ' m ' be a given +ve integer. The graph coloring problem is assigning colors to the vertices of an undirected graph with the restriction that no two adjacent vertices are assigned the same color yet only ' m ' colors are used.

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.

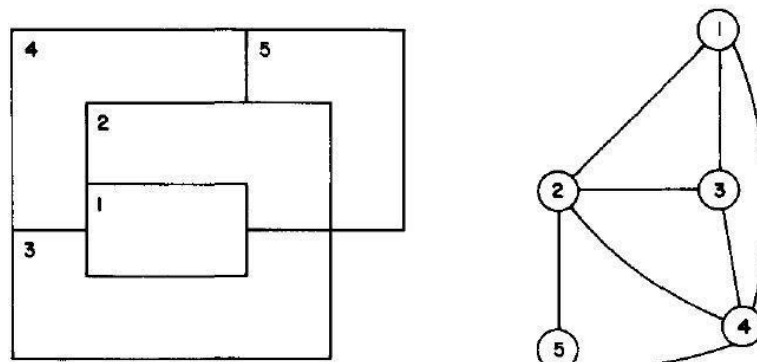
The m -colorability optimization problem asks for the smallest integer ' m ' for which the graph G can be colored. This integer is referred as "**Chromatic number**" of the graph.

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.

o Example:



o **A map and its planar graph representation**

The

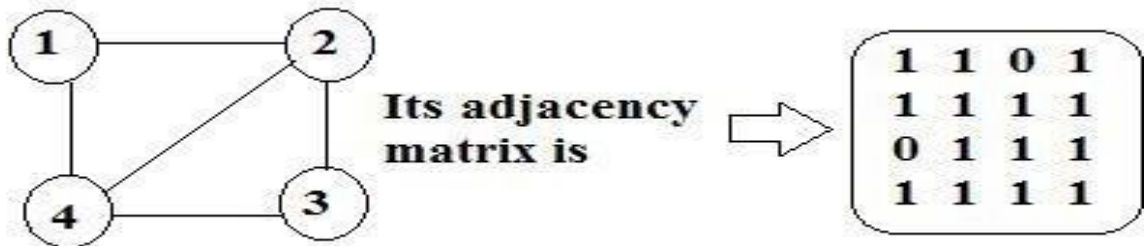
above map requires 4 colors.

- 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.

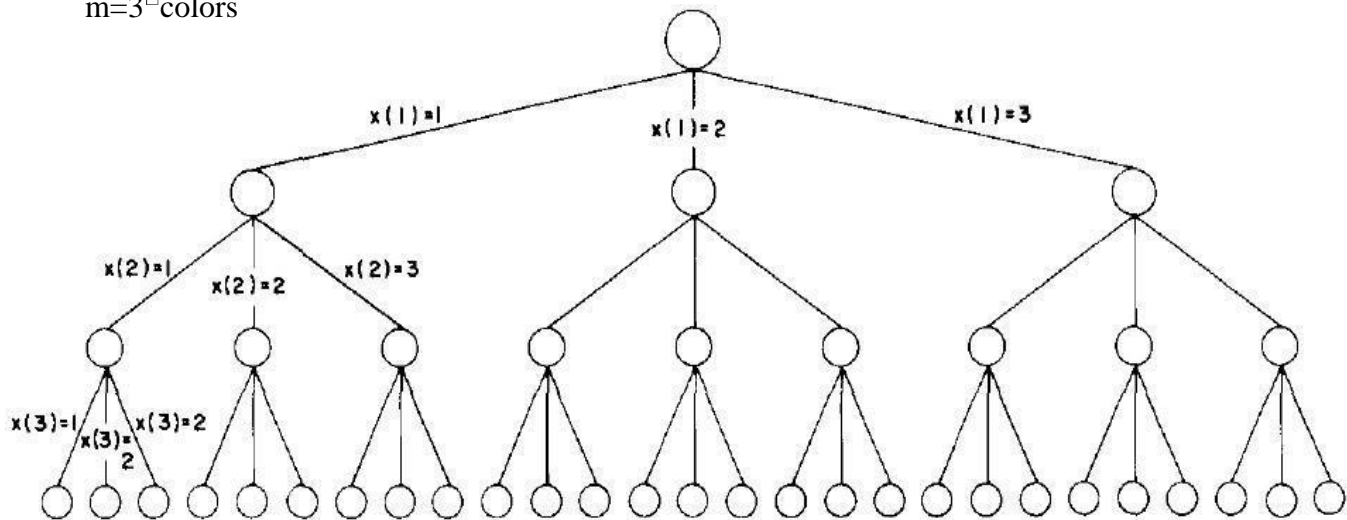
Colors are represented by the integers 1, 2, ..., m and the solutions are given by the n -tuple (x_1, x_2, \dots, x_n)

x_i Color of node i .

State Space Tree for

$n=3$ nodes

$m=3$ colors



State space tree for M Coloring when $n = 3$ and $m = 3$

1st node coloured in 3-ways

2nd node coloured in 3-ways

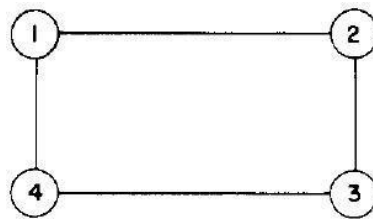
3rd node coloured in 3-ways

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. }	Algorithm NextValue(k){ // $x[1], x[2], \dots, x[k-1]$ have been assigned integer values in the range $[1, m]$ repeat { $x[k] = (x[k] + 1) \bmod (m + 1)$; //next highest color if($x[k] = 0$) then return; // all colors have }

<pre> if(x[k]=0) then return; // no new color possible if(k=n) then write(x[1: n]; else mcoloring(k+1); } until(false) } </pre>	<pre> 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) } </pre>
---	---

Previous paper example:

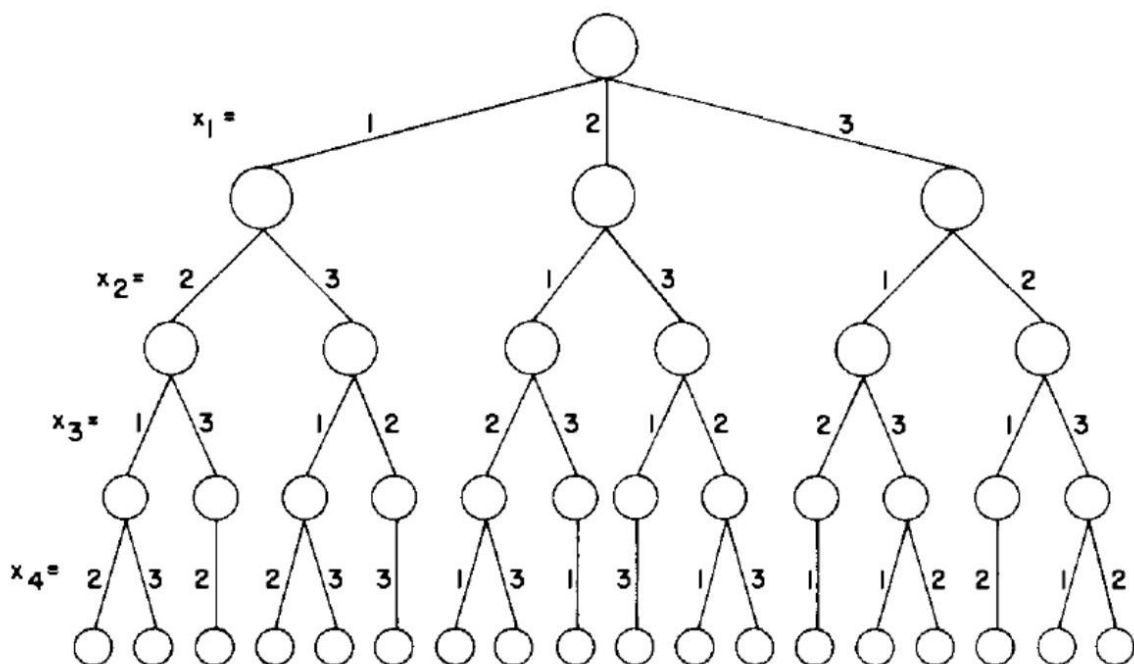


Adjacency matrix is

$k \sim 0$

~ ~

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



A 4 node graph and all possible 3 colorings

