UNIT-VI–BACKTRACKING

Backtracking: General method, Applications-N-QUEEN Problem, Sum of Sub Sets problem, Graph Coloring, Hamiltonian Cycles, 0/1 Knapsack Problem.

-0-0-0-0-

Introduction

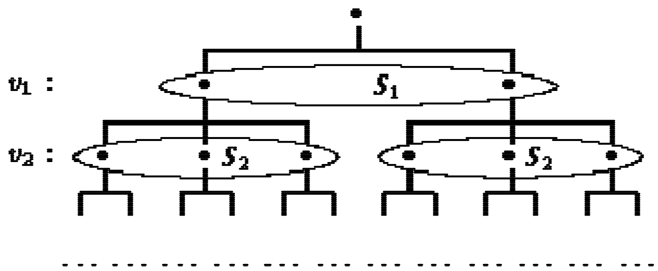
Backtracking is a refinement of the brute force approach, which systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (v_1, ..., v_m) of values and by traversing, in a depth first manner, the domains of the vectors until the solutions are found.

When invoked, the algorithm starts with an empty vector. At each stage it extends the partial vector with a new value. Upon reaching a partial vector (v_1, ..., v_i) which can't represent a partial solution, the algorithm backtracks by removing the trailing value from the vector, and then proceeds by trying to extend the vector with alternative values.

```
ALGORITHMtry(v1,...,vi)
{
   IF(v1,...,vi)is a solution THEN RETURN(v1,...,vi) FOR
   each v DO
      IF(v1,...,vi,v)is acceptable vector THEN
        sol = try(v1,...,vi,v)
        IF sol!=()THEN RETURN sol END
   END
   RETURN()
}
```

If S_i is the **domain** of v_i, then $S_1 \times ... \times S_m$ is the **solution space** of the problem. The **validity criteria** used in checking for acceptable vectors determines what portion of that space needs to be searched, and so it also determines the resources required by the algorithm.

The traversal of the solution space can be represented by a depth-first traversal of a tree. The tree itself is rarely entirely stored by the algorithm in discourse; instead just a path toward a root is stored, to enable the backtracking.



In case of greedy and dynamic programming techniques, we will use Brute force approach. It means, we will evaluate all possible solutions, among which, we select one solution as optimal solution. In backtracking technique, we will get same optimal solution with less number of steps. So we use backtracking technique. We can solve problems in an efficient way when compared to other methods like greedy method and dynamic programming. In this we will use bounding functions (criterion functions), implicit and explicit conditions. While explaining the general method of backtracking technique, there we will see implicit and explicit constraints. The major advantage of backtracking method is, if a partial solution $(x_1,x_2,x_3.....,x_i)$ can't lead to optimal solution then $(x_{i+1}...x_n)$ solution may be ignored entirely.

Explicit constraints: These are rules which restrict each xi to take on values only from a given set.

Example

1) Knapsack problem , the explicit constraints are,
   i) x_i=0 or 1
   ii) 0<=x_i<=1
2) 4-queens problem : in 4 queens problem, the 4 queens can be placed in 4x4 chess board in 4^4 ways.

Implicit constraints: These are rules which determine which of the tuples in the solution space satisfy criterion function.

Example: In 4 queens problem, the implicit constraints are no 2 queens can be on the same row, same column and same diagonal.

Let us see some terminology which is being used in this method.

1) **Criterion Function:** it is a function p(x1,x2,x3,…xn) which needs to be maximized or minimized for a given problem.

2) **Solution Space** : All tuples that satisfy the explicit constraints define a possible solution space for a particular instance 'i' of the problem.
For example consider the following tree. ABD, ABE, AC are the tuples in solution space.

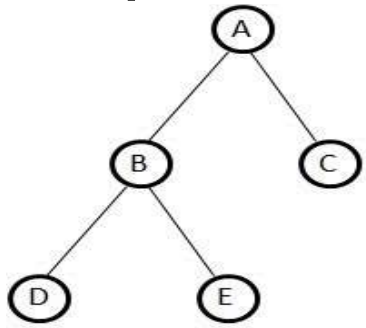


Fig) The organization of a solution space

3) **Problem state:** each node in the tree organization defines a problem state. So, A,B ,C are problem states.

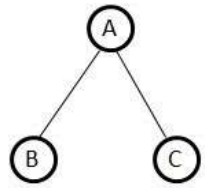


Fig)Tree(Problem State)

4) **Solution states:** These are those problem states S for which the path from the root to S define a tuple in the solution space.

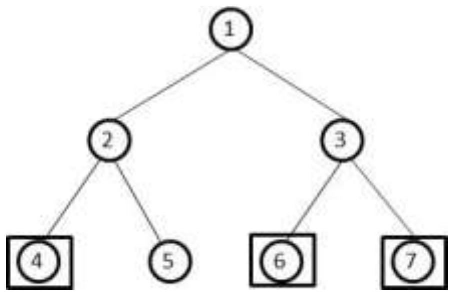


Fig)Tree(Solution state)

Here square nodes indicate solution. For the above solution space, there exists 3 solution states. These solution states represented in the form of tuples i.e. (1,2,4), (1,3,6) and (1,3,7) are the solution states.

5) **state space tree:** if we represent solution space in the form of a tree then the tree is referred as the state space tree.

For example given is the state space tree of 4-queen problem. Initially x1=1 or 2 or 3 or 4. It means we can place first queen in either of 1/2/3/4 column. If x1=1then x2 can be paced in either 2^{nd},3^{rd} , or 4^{th} column. If x2=2 then x3 can be placed either in 3^{rd} or 4^{th} column. If x3=3 then x4=4. So nodes 1-2-3-4-5 is one solution in solution space. It may or may not be feasible solution. Similarly we can observe the remaining solutions in the figure.

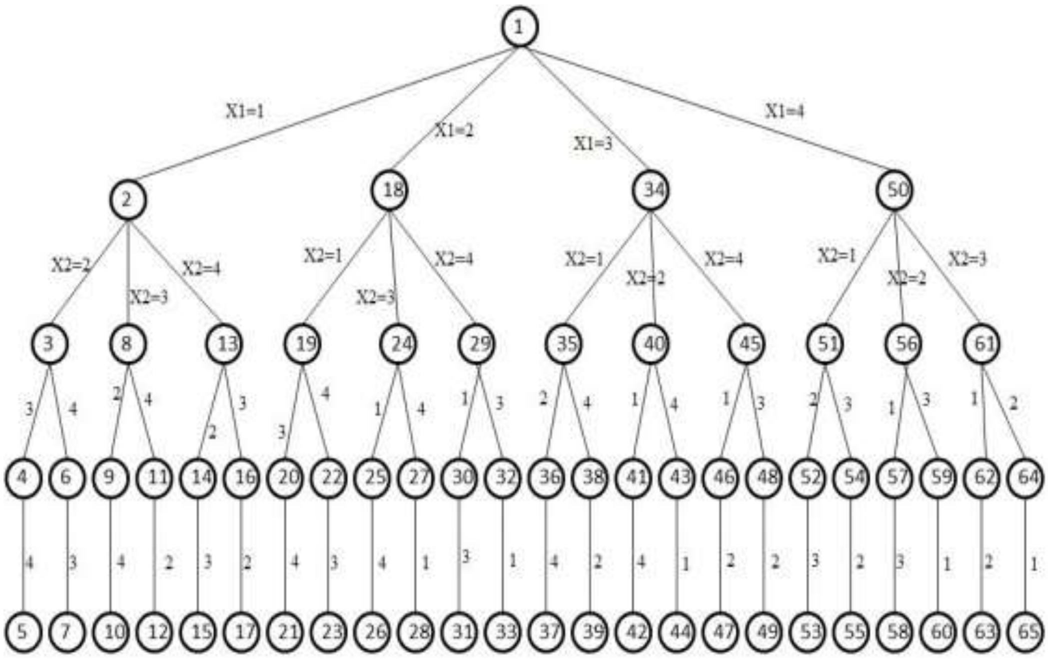


Fig)Tree organization of the 4 queen solution space

6)**Answer states** : These solution states s for which the path from the root to S defines a tuple which is a member of the set of solutions.(i.e. it satisfies the implicit constraints) of the problem. Here 3,4, are answer states. (1,3) and (1,2,4) are solution states.

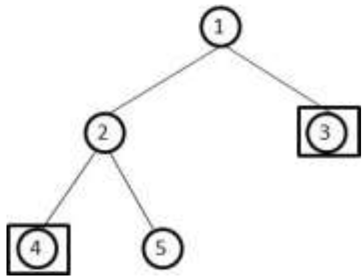


Fig)Tree(answer states)

7)**Live node**: A node which has been generated and all of whose children have not yet been generated is live node. In the fig (a)node A is called live node since the children of node A have not yet been generated.

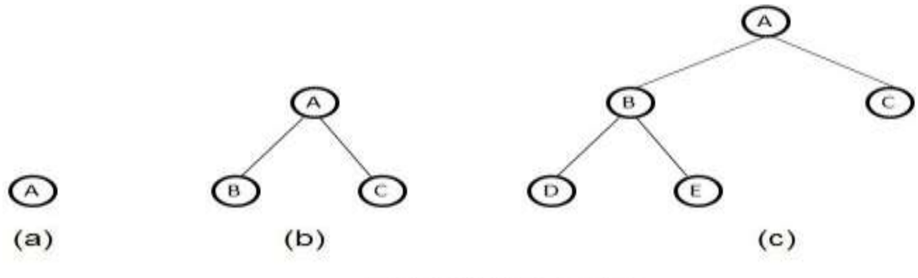


Figure) Live nodes

In fig (b) node A is not a live node but B,C are live nodes.
In fig(c)nodes A,B are not live and D, E Care live nodes.

8) **E-node :** The live node whose children are currently being generated is called E-node.(node being expanded).

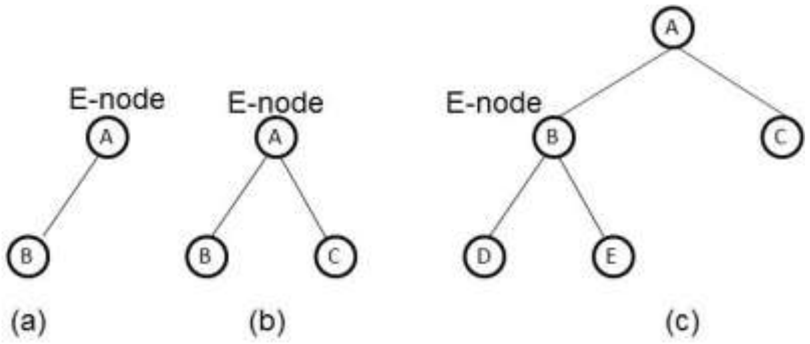


Figure) E-nodes

9) **Dead node:** it is a generated node that is either not to be expanded further or one for which all of its children have been generated.
Ex)In figure(a) nodes A,B,C are dead nodes since node A's children already generated and Nodes B,C are not expanded.

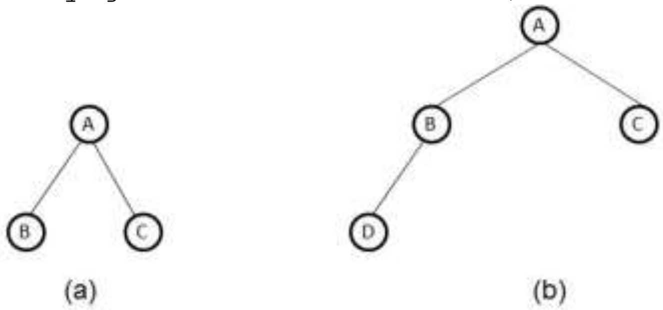


Figure) Dead nodes

In figure (b) assumed that node B can generate one more node so nodes A,D,C are dead nodes.

Applications:
1) n-Queens Problem(4-Queensand8-QueensProblem)

Consider an nxn chess board. Let there are n-Queens. These n Queens are to be placed on the nxn chess board so that no two queens are on the same column, same row or same diagonal.

n-queens Problem: The n-queens problem is a generalization of the8-queens problem. Now n-queens are to be placed on an nxn cross board so that no two attack; that is no two queens are on the same row, column, or diagonal. The solution space consists of all n! permutations of n-tuple (1,2,3,..n).

The following figure shows a possible tree organization for the case n = 4. A tree such as this is called a permutation tree. The edges are labeled by possible values of x_i.

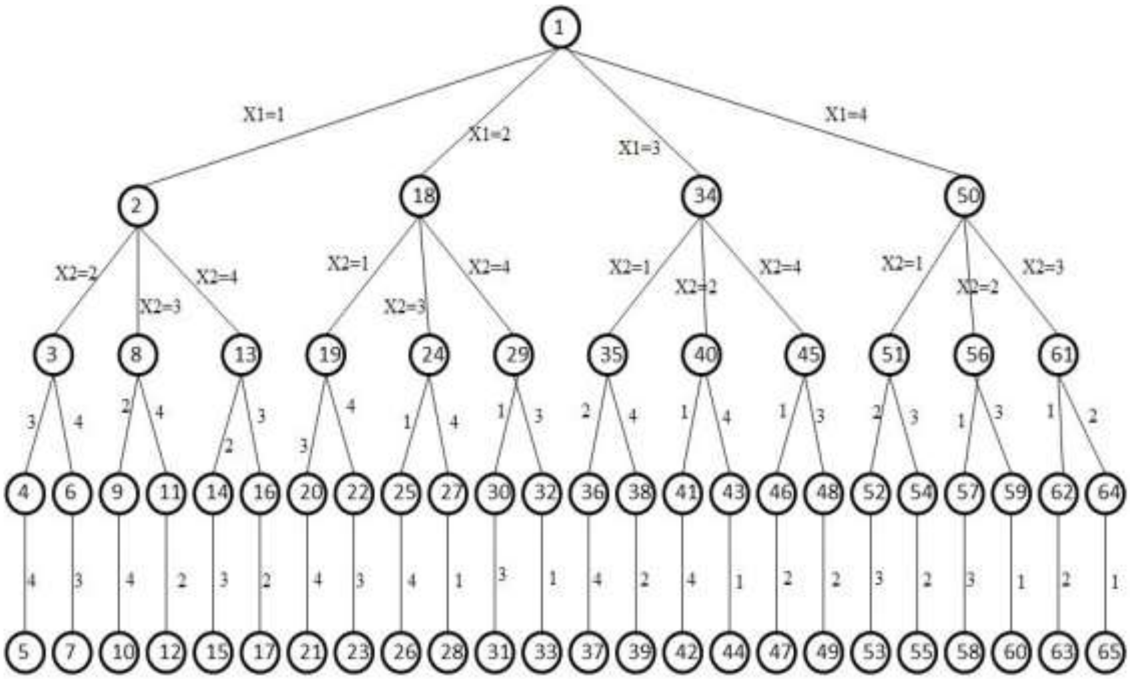


Figure)The organization of 4-queens solution space

Edges from level 1 to level 2 nodes specify the values for x_1.Thustheleftmostsub-treecontainsallsolutionswith x_1=1.

Edges from level i to level i+1 are labeled with the values of x_i. The solution space is defined by all paths from the root node to a leaf node. There are 4!=24 leaf nodes in the permutation tree.

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 upper left to lower right has the same row-col value.

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

Consider the queen at a[4,2]. The squares that are diagonal to this queen (running from upper left to lower right) are a[3,1],a[5,3], a[6,4], a[7,5],a[8,6]. All these squares have a (row – column) value of 2. Also every element on the same diagonal that goes from the upper right to the lower left has the same (row + column) value.

Suppose two queens are placed at positions (i,j) and (k,l)then by the above we can say they are on the same diagonal if

i-j=k-l which is primary diagonal or
i+j=k+l which is secondary diagonal

Equation for primary diagonal	Equation for Secondary diagonal
i-j=k-l	i+j=k+l
this can be written as follows	this can be written as follows
j-l=i-k	j-l=k-i

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

The algorithm place(k,i) returns a Boolean value that is true if kth queen can be placed in column 'i'. it tests both whether 'i' is distinct from all previous values x[i]..x[k-1] and whether there is no other queen on the same diagonal.

Its computing time is O(k-1).

The array x[1..n] is a global array. Let (x1,x2,x3,…xn) be the solution vector where xi is the column number on which the i^{th} queen is placed.(i may be row number).

Using the algorithm place() a queen is placed in k^{th} row , i^{th} column and return true otherwise false.

```
 Algorithm place(k,i)
 {
   forj:=1tok-1 do
     if((x[j]=i)or(abs(x[j]-i)=abs(j-k))then return
        false;

   returntrue;
 }
```

This algorithm is invoked by nqueens(1,n).

The algorithm for obtaining solution n-queens problem is given below.

```
Algorithmnqueens(k,n)
{
    fori:=1tondo
      {
         if(place(k,i)then
          {
             X[k]:=i;
             if (k=n) then
               write(x[i:n);
             else
               nqueens(k+1,n);
           }
      }
}
```

For an 8x8chess board there are ${}^{64}C_8$ possible ways to place 8 Queens using brute force approach. However by allowing only placements of queens on distinct rows and columns, we require the examination of at most 8! Tuples.

Fora4x4chessboardthereare${}^{16}C_4$possiblewaystoplace 4 Queens using brute force approach. However by allowing only placements of queens on distinct rows and columns, we require the examination of at most 4! Tuples.

Place first queen in the first row
In the first column.
As it is the first queen It is
Not under attack.
X[1]=1(column value is assigned)

	1	2	3	4
1	1			
2				
3				
4				

X[1]=1

To place second queen in second row
Start with first column.
It is under attack
Second column also Under attack
Third column not under attack by other queens.
Soweplacequeenin3rdcolumn. X[2]=3

	1	2	3	4
1	1			
2	-	-	2	
3				
4				

X[2]=3

To place third queen in third row
First col under attack
Second column under attack
Third column under attack
Fourth column under attack
Not possible to place queen in third row
Because placement of previous queens is
Not correct. So **backtrack** to previous row
And move the queen to another possible place
And continue.

	1	2	3	4
1				
2	1			
3			2	
4	-	-	-	-

Go to second row
Move the queen to another col.
Another possibility is column 4.
Move to col4.
Now X[2]=4

	1	2	3	4
1	1			
2	-	-	-	2
3				
4				

X[2]=4

Go to third row to place 3rdqueen
First col under attack
Second column not under attack by other queens
Soplacethequeenin2ndcol.
X[3]=2

	1	2	3	4
1	1			
2				2
3	-	3		
4				

X[3]=2

Now to place 4th queen in 4th row
First col under attack by other queen
Second col under attack by other queen
Third col under attack by other queen
Fourth col under attack by other queen
Notpossibletoplacethequeenin4throw
As there is a problem in the placement of
Previous queens
Backtrack to previous placements
Go to 3rdrow and try to move the queen to another place.
The other places are under attack go to 2nd row
Already we checked all possibilities in 2nd row
we backtrack to first row.

	1	2	3	4
1	1			
2				2
3		3		
4	-	-	-	-

First queen is moved to 2nd column
X[1]=2

	1	2	3	4
1		1		
2				
3				
4				

X[1]=2

Second queen in second row
First col under attack
Second column under attack
Third col under attack
4th col not under attack
Soplacequeenin4thcol
X[2]=4

	1	2	3	4
1		1		
2	-	-	-	2
3				
4				

X[2]=4

To place third queen in third row
First col not under attack
So place the queen in first col
X[3]=1

	1	2	3	4
1		1		
2				2
3	3			
4				

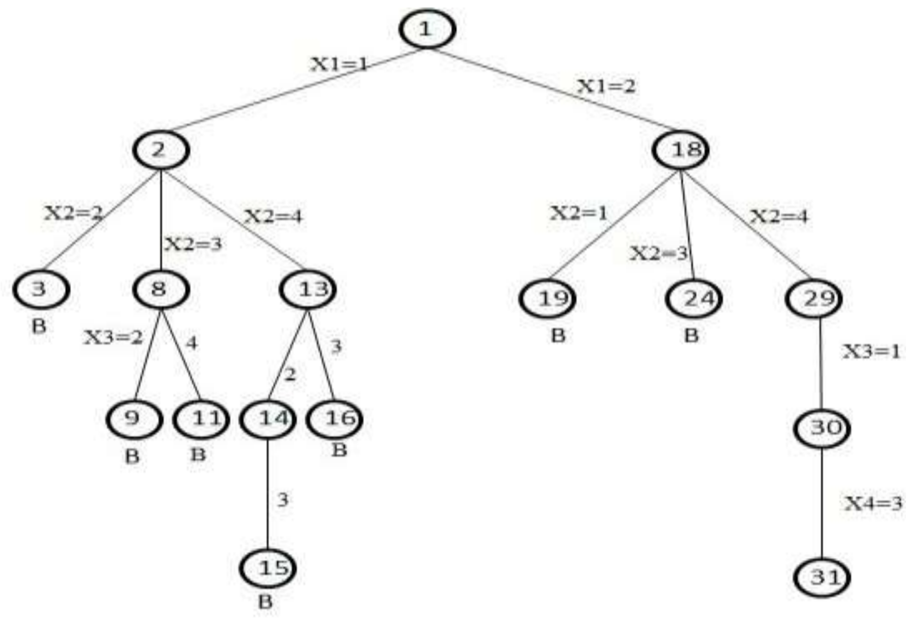
X[3]=1

To place 4th queenin4throw
first col under attack
second col under attack
third col not under attack
So place the 4thqueen in 3rdcol
x[4]=3

	1	2	3	4
1		1		
2				2
3	3			
4	-	-	4	

X[4]=3

All four queens are placed in the
4x4 chess board with out attacking each other.
In the same way it is possible to place all 8queens in
an 8x8 chess board without attacking each other.



Portion of the tree that is generated during backtracking

Figure shows the part of the solution space tree that is generated. The tree generated as per the above processing. Nodes are numbered in the order in which they are generated. A node that gets killed as a result of backtracking has a B under it.

Tracingofthealgorithmtoplace4queensona4x4cross board such that no two queens attack each other.

```
nqueens(k,n)     place(k,i)

nqueens(1,4)     place(1,1)returns Truesox[1]=1

nqueens(2,4)     place(2,1) returns False
                 place(2,2) returns False
                 place(2,3)returnsTruesoX[2]=3

nqueens(3,4)place(3,1)returnsFalse
                 place(3,1)returnsFalse
                 place(3,1)returnsFalse
                 place(3,1)returnsFalse
                                      Backtracking

nqueens(2,4)place(2,4) returns True so X[2]=4

nqueens(3,4)place(3,1)returnsFalse
                 place(3,2)returnsTruesoX[3]=2
nqueens(4,4)place(4,1) returns False
                 place(4,2)returnsFalse
                 place(4,3)returnsFalse
                 place(4,4)returnsFalse
                                   Backtracking
nqueens(1,4)place(1,2)returnsTruesox[1]=2

nqueens(2,4)     place(2,1)returnsFalse
                 place(2,2) returns False
                 place(2,3) returns False
                 place(2,4)returnsTruesoX[2]=4

nqueens(3,4)     place(3,1)returnsTruesoX[3]=1

nqueens(4,4)     place(4,1) returns False
                 place(2,2)returnsFalse
                 place(2,3)returnsTruesoX[4]=3
```

The solution vector for a 4x4 cross board to place 4 non attacking queens is

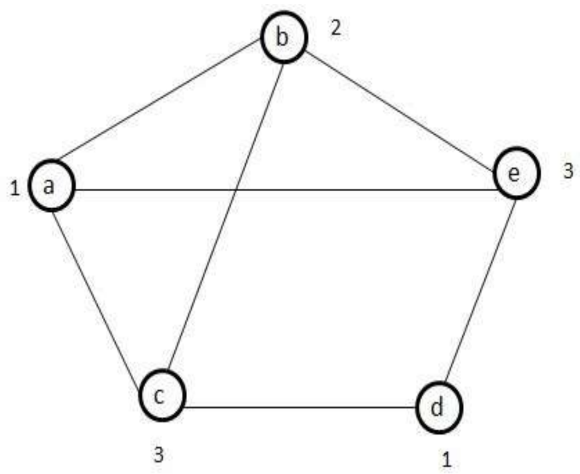
x[1]=2
x[2]=4
x[3]=1
x[4]=3

2) GRAPHCOLORING

Let G be a graph and m be a given positive integer. We want to discover whether the node 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 termed the m-colorability decision problem. 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. The integer is referred to as the chromatic number of the graph.

For example the following 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 thee colors are needed to color this graph and hence this graph’s chromatic number is 3.



An example graph and its coloring

State space tree for coloring a graph containing 3 nodes using 3 colors

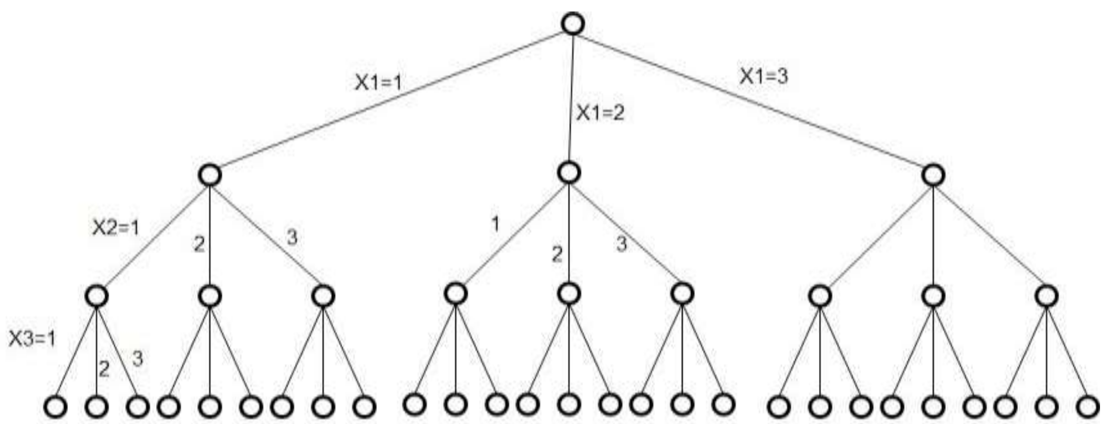


Fig)State space tree for m-Coloring when n=3 and m=3

The algorithm mcoloring 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.

```
Algorithmmcoloring(k)
{
  repeat
   {
      nextvalue(k);
      if(x[k]=0)thenreturn; if
      (k=n) then
         write(x[1:n]);
      else
         mcoloring(k+1);
   }until(false);
}
```

No of vertices=n
No of colors= m
Solution vector=X[1],X[2],X[3]....... X[n]
The values of solution vector may be longs to{0,1,2,3..m}

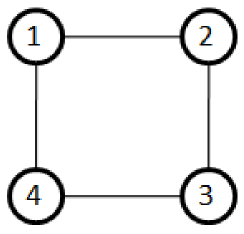
The following Algorithm is used to generate next color.

Assume that 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, the x[k]=0.

```
Algorithm nextvalue(k)
{
  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;
                      //g[k,j]an edge and
                      //vertices k and j have same color
      }
      if(j=n+1)thenreturn;
    }until(false);
 }
```



Graph

Adjacency Matrix G

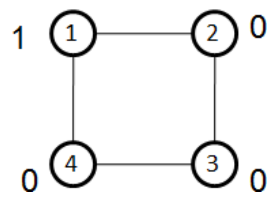
	1	2	3	4
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	1	0	1	0

```
 Assume that n=4 and m=3
 X[1]=0,x[2]=0,x[3]=0,x[4]=0
 If we call the algorithm coloring(k)

mcoloring(1)    i.e.k=1
nextvalue(1)
k=1
x[1]=(x[1]+1)mod(m+1)
x[1]=0+1mod4 x[1]=1

             G[k,j]!=0andx[k]=x[j]
j=1G[1,1]false       and        true=false
j=2G[1,2]trueandfalse=false
j=3G[1,3]falseandfalse=false
j=4G[1,4]trueandfalse= false

x[1]=1,x[2]=0,x[3]=0,x[4]=0
```



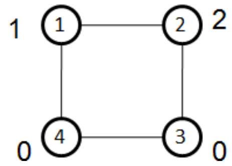

```
mcoloring(2)    i.e.k=2
nextvalue(2)
k=2
x[2]=(x[2]+1)mod(m+1)
x[2]=0+1mod4 x[2]=1
             G[k,j]!=0andx[k]=x[j]
j=1 G[2,1]     True    and     True=True         break
             G[2,1] is an edge and
             adjacent vertices have same color

x[2]=(x[2]+1)mod(m+1)
x[2]=(1+1)mod4=2mod4
x[2]=2

             G[k,j]!=0andx[k]=x[j]
j=1G[2,1]TrueandFalse=False
j=2G[2,2]FalseandTrue=False
j=3G[2,3]TrueandFalse=False
j=4G[2,4]FalseandFalse= False

x[1]=1,x[2]=2,x[3]=0,x[4]=0
```

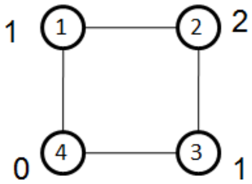
assume that the number mentioned outside the node belongs to color



mcoloring(3)   i.e.k=3
nextvalue(3)
k=3
x[3]=(x[3]+1) mod(m+1)
x[3]=0+1mod   4
x[3]=1

```
             G[k,j]!=0and  x[k]=x[j]
j=1G[3,1]      False    and    True=     False
j=2G[3,2]      True     and     False=   False
j=3G[3,3]      False    and      True=   False
j=4G[3,4]      True     and     False=   False
```

x[1]=1,x[2]=2,x[3]=1,x[4]=0.



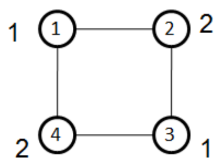
mcoloring(4)   i.e.k=4
nextvalue(4)
k=4
x[4]=(x[4]+1)mod(m+1)
x[4]=0+1mod4 x[4]=1

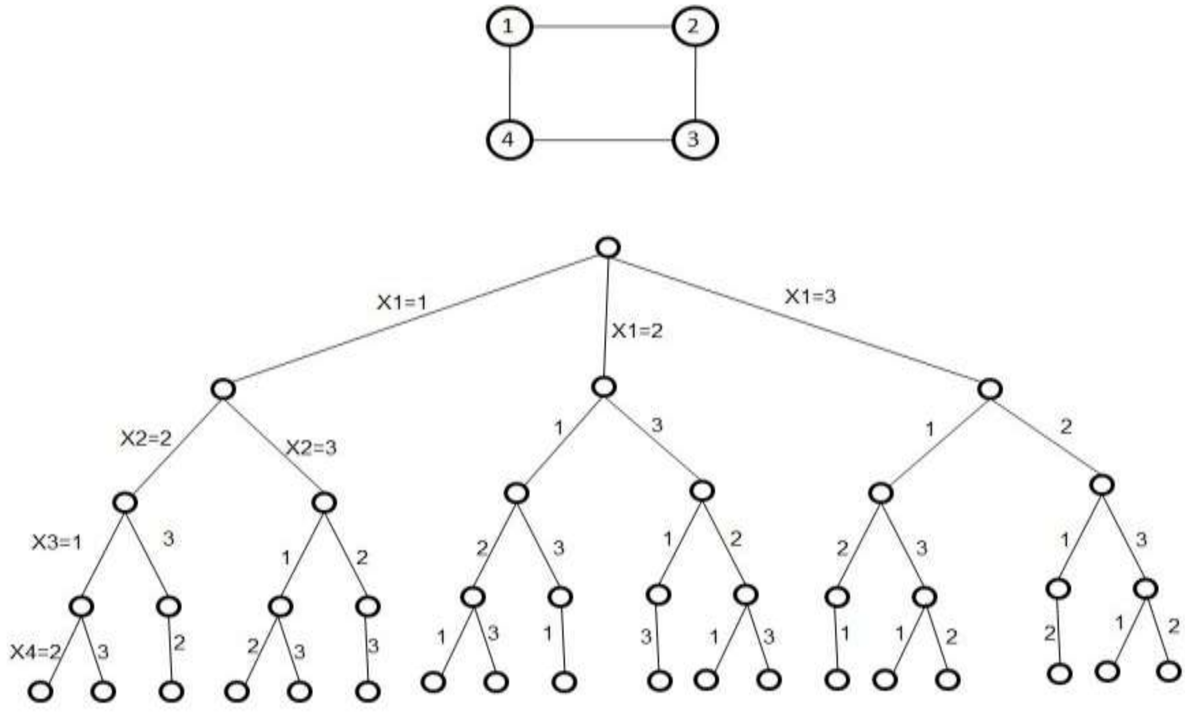
```
             G[k,j]!=0andx[k]=x[j]
j=1G[4,1]      true     and    true=True        so break
                      adjacent vertices have same color
```

x[4]=(x[4]+1) mod(m+1)
x[4]=1+1mod   4
x[4]=2

```
             G[k,j]!=0 and x[k]=x[j]
j=1G[4,1]      True     and    False   =  False
j=2G[4,2]      False    and    True    =  False
j=3G[4,3]      True     and    False   =  False
j=4G[4,4]      False    and    True    =  False
```

x[1]=1,x[2]=2,x[3]=1,x[4]=2

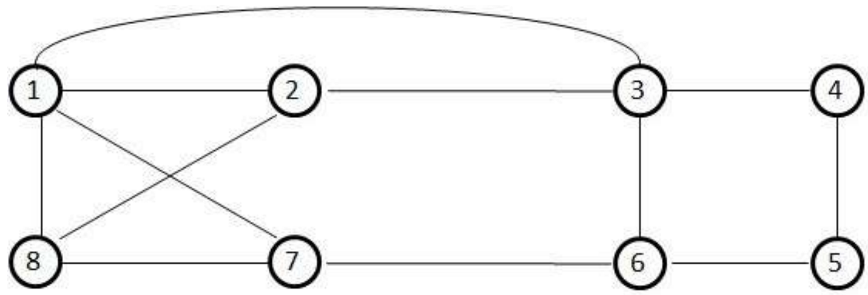




A 4-node graph and all possible 3-colorings

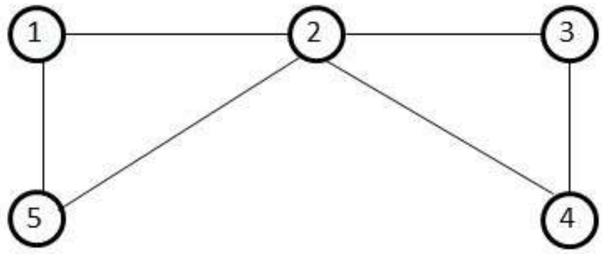
HAMILTONIANCYCLES

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. In other words if a Hamiltonian cycle begin sat Some vertex v1∈G and the vertices of Garevisitedin the order v_1, v_2,…v_{n+1} then the edges (v_i,v_{i+1}) are in E, 1<=i<=n, and the v_i are distinct except for v_1 and v_{n+1}, which are equal.



Graph containing a Hamiltonian Cycle

The above graph contains the Hamiltonian cycles
1,2,3,4,5,6,7,8,1
1,3,4,5,6,7,8,2,1
1,2,8,7,6,5,4,3,1



Graph Does not containing a Hamiltonian Cycle

The graph contains no Hamiltonian cycle.

To check whether there is a Hamiltonian cycle or not we may use backtracking method. The graph may be directed or undirected. Only distinct cycles are output.

The backtracking solution vector (X_1,X_2,X_3,…X_n) is defined so that xi represents the i^{th} visited vertex of the proposed cycle.

Now all we need to do is determine how to compute the set of possible vertices for x_k if x_1,..x_{k-1} have already been chosen. If k=1 then x1 can be any of the n vertices.

The algorithm nextvalue(k) which determines a possible next vertex for the proposed cycle.

Using nextvalue we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix G[1:n,1:n], then setting x[2:n] to 0 and x[1] to 1 and then executing Hamiltonian(2).

```
//x[1:k-1]is a path of k-1 distinct vertices
//if x[k]=0 then no vertex has 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].Otherwise x[k]=0.
//ifk=n then in addition x[k]is connected to x[1].

Algorithmnextvalue(k)
{
  Repeat
  {
    X[k]:=(x[k]+1)mod(n+1);
    if (x[k]=0) then
          return;
    if(G[x[k-1],x[k]]!=0)then
      {
        Forj:=1tok-1 do
            if(x[j]=x[k])then break;
        if(j=k)then
          if((k<n)or((k=n)andG[x[n],x[1]]!=0))then return;
      }
   }until(false);
}
```

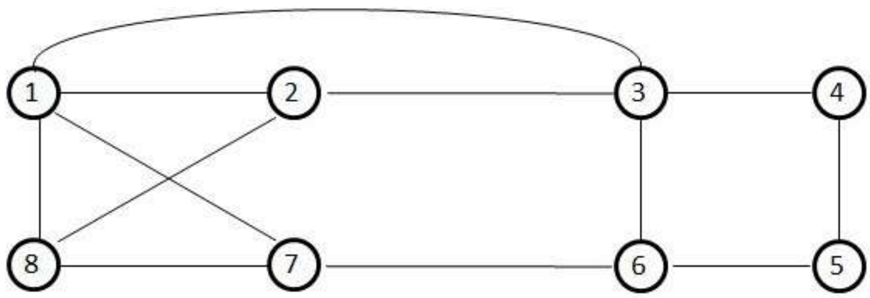
Algorithm to generate nextvertex.

The algorithm Hamiltonian()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.

```
AlgorithmHamiltonian(k)
{
   Repeat
   {
     nextvalue(k);
     if(x[k]=0)thenreturn;
     if(k=n)thenwrite(x[1:n]); else
       Hamiltonian(k+1);
   }until(false);
}
```

Algorithm to find all Hamiltonian cycles.

Example)



Graph containing a Hamiltonian Cycle

No of verticesn=8

Adjacency matrix G

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	1	1
2	1	0	1	0	0	0	0	1
3	1	1	0	1	0	1	0	0
4	0	0	1	0	1	0	0	0
5	0	0	0	1	0	1	0	0
6	0	0	1	0	1	0	1	0
7	1	0	0	0	0	1	0	1
8	1	1	0	0	0	0	1	0

Solution vertex

X[1]	1
X[2]	0
X[3]	0
X[4]	0
X[5]	0
X[6]	0
X[7]	0
X[8]	0

Algorithm starts with vertex 1 as initial vertex.
Solution vertex must contain a series of vertices in the cycle.
X[1:n]i.e.x[1:8]
X[1]=1
andx[2:8]=0
we will add one by one vertices to the solution vector.

Hamiltonian(2) k=2
Nextvalue(2) k=2
X[2]=(x[2]+1)mod(8+1)=(0+1)mod9= 1
Isthereanedgebetweenkandk-1 G[x[k-1],x[k]]!=0
G[1,1] no edge False
X[2]=(x[2]+1)mod(8+1)
=(1+1)mod9
=2

If(G[x[k-1],x[k]]!=0)
G[1,2]edge True

Solution vector

X[1]	1
X[2]	2
X[3]	0
X[4]	0
X[5]	0
X[6]	0
X[7]	0
X[8]	0

Are there duplicate vertices in the path
J=1is x[j]=x[k]
isx[1]=x[2]no false

k<n returns till we need to add vertices

```
Hamiltonian(3)     k=3
Nextvalue(3)       k=3
X[3]=(x[3]+1)mod(8+1)=(0+1)mod9=1 X[3]=1
Isthereanedgebetweenkandk-1

 If(G[x[k-1],x[k]]!=0)
       G[2,1]edge True

Are there duplicate vertices in the path
   J=1is x[j]=x[k]
        isx[1]=x[3]
              1=1   True   break

X[3]=(x[3]+1)mod(8+1)
X[3]=(1+1)mod9
    =2

If(G[x[k-1],x[k]]!=0)
      G[2,2]edge False

X[3]=(x[3]+1)mod(8+1)
X[3]=(2+1)mod9
    =3

 If(G[x[k-1],x[k]]!=0)
       G[2,3]edge True

J=1   isx[j]=x[k]
      Is1=2nofalse
J=2   is2=3no false

      As k<n returns till we need to add vertices


Hamiltonian(4)k=4
Nextvalue(4)    k=4

X[4]=(x[4]+1)mod(8+1)
    =(0+1)mod9=1 X[4]=1
Isthereanedgebetweenkandk-1

 If(G[x[k-1],x[k]]!=0)
       G[3,1]edge True

Are there duplicate vertices in the path
   J=1is x[j]=x[k]
        Is x[1]=x[4]
              1=1   True   break
```

Solution vector

X[1]	1
X[2]	2
X[3]	3
X[4]	0
X[5]	0
X[6]	0
X[7]	0
X[8]	0

```
X[4]=(x[4]+1)mod(8+1)
X[4]=2

 If(G[x[k-1],x[k]]!=0)
       G[3,2]edge True

 Are there duplicate vertices in the path
   J=1is x[j]=x[k]
        isx[1]=x[4]
              1=2   False

    J=2isx[j]=x[k]
         isx[2]=x[4]
              2=2   True Break



X[4]=(x[4]+1)mod(8+1)
X[4]=3


If(G[x[k-1],x[k]]!=0)
      G[3,3]edge False

X[4]=(x[4]+1)mod(8+1)
X[4]=4

 If(G[x[k-1],x[k]]!=0)
       G[3,4]edge True


J=1is x[j]=x[k]
      Is1=4nofalse
J=2is2=4nofalse
J=3is3=4nofalse
      As k<n returns till we need to add vertices
```

Solution vector

X[1]	1
X[2]	2
X[3]	3
X[4]	4
X[5]	0
X[6]	0
X[7]	0
X[8]	0

```
Hamiltonian(5)k=5
Nextvalue(5)    k=5

X[5]=(x[5]+1)mod(8+1)
    =(0+1)mod9=1 X[5]=1
Isthereanedgebetweenkandk-1

If(G[x[k-1],x[k]]!=0)
      G[4,1]no edge False

X[5]=(x[5]+1)mod(8+1)
    =(1+1)mod9=2

If(G[x[k-1],x[k]]!=0)
      G[4,2]no edge False

X[5]=(x[5]+1) mod (8+1)
```

```
    = (3+1)mod 9 = 3
If(G[x[k-1],x[k]]!=0)
      G[4,3]edge True

Are there duplicate vertices in the path
   J=1is x[j]=x[k]
        isx[1]=x[5]
              1=3   False
J=2is x[j]=x[k]
        isx[2]=x[5]
              2=3   False
J=3is x[j]=x[k]
        isx[3]=x[5]
              3=3   True duplicate found break

X[5]=(x[5]+1)mod(8+1)
X[5]=5

 If(G[x[k-1],x[k]]!=0)
       G[4,5]edge True

 Are there duplicate vertices in the path
   J=1is x[j]=x[k]
        isx[1]=x[5]
              1=5   False

    J=2isx[j]=x[k]
         isx[2]=x[5]
              2=5   False

   J=3isx[j]=x[k]
        isx[3]=x[5]
              3=5   False

    J=4isx[j]=x[k]
         isx[4]=x[5]
              4=5   False

        As k<n returns till we need to add vertices
```

Solution vector

X[1]	1
X[2]	2
X[3]	3
X[4]	4
X[5]	5
X[6]	0
X[7]	0
X[8]	0

Hamiltonian(6)k=6
Nextvalue(6)   k=6

The solution vector for Hamiltonian cycles
1,2,3,4,5,6,7,8,1
1,8,2,3,4,5,6,7,1
1,3,4,5,6,7,8,2,1

SUMOFSUBSETS

Suppose we are given n distinct positive numbers(usually called weights) and we desire to find all combinations of these numbers whose sum are m.

This is called the sumofsubsets problem.

Ex1) given positive numbers W_i, 1<=i<=n, and m, this problem calls for finding all subsets of w_i whose sums are m. For example,if n=4,$(w_1,w_2,w_3,w_4)=(7,11,13,24)$and m=31, then the desired subsets are (7,11,13) and (7,24).

Rather than representing the solution vector by w_i which sum to m, we could represent the solution vector by giving the indices of these w_i.

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

In general all solution subset is represented by n-tuple $(X_1,X_2,X_3,\ldots X_n)$ such that Xi∈{0,1},1<=i<=n. The X_i is 0 if w_i is not chosen and xi=1if$w_i$ischosen.Thesolutions to the above instances are (1,1,1,0) and (1,0,0,1). This formulation expresses all solutions using a fixed sized tuple.

The sum of sub set is based on fixed size tuple. Let us draw a tree structure for fixed tuple size formulation.

All paths from root to a leaf node define a solution space. The left sub tree of the root defines all subsets containing W_1 and the right sub tree defines all subsets not containing W_1 and so on.

Step1) Start with an empty set

Step2) Add next element in the list to the subset

Step3) If the subset is having sum=m then stop with that sub set as solution.

Step 4) If the sub set is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.

Step5)if the subset is feasible then repeat step-2

Step 6)if we have visited all elements without finding a suitable subset and if no backtracking is possible, then stop with no solution.

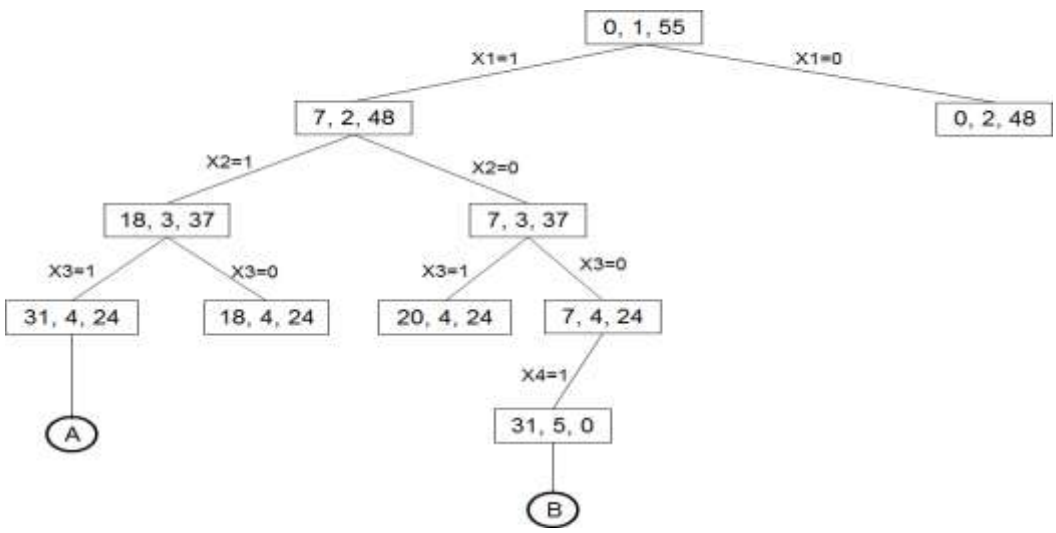
s-sum of all selected elements
k-denotes the index of chose n element
r - Initially sum of all elements. After selection of some element from the set subtract the chosen value from r each time.
W(1:n)-represents set containing n elements.
X[i]-solution vector1<=i<=k

```
Algorithm sumofsubsets(s,k,r)
{
  X[k]:=1;
  if(s+w[k]=m)thenwrite(x[1:k]);//subset found else
    if (s+w[k]+w[k+1]<=m) then
       sumofsubsets(s+w[k],k+1,r-w[k]);

   //generate right child and evaluate Bk.

   if((s+r-w[k]>=m)and(s+w[k+1]<=m))then
    {
      X[k]:=0;
      sumofsubsets(s,k+1,r-w[k]);
    }
}
```

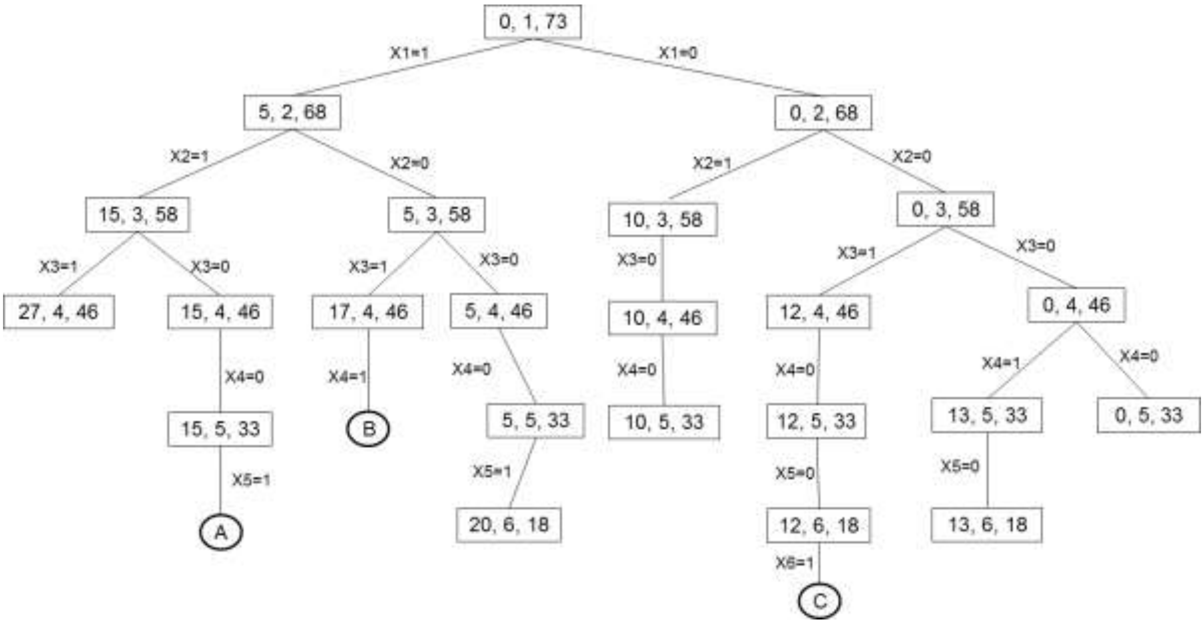
Ex)n=4,(w_1,w_2,w_3,w_4)=(7,11,13,24)and m=31
Solution Vector=(x[1],x[2],x[3],x[4])



Portion of state space Tree

Solution A = {1,1,1,0}
Solution B={1,0,0,1}

Ex2)n=6,m=30andw[1:6]={5,10,12,13,15,18}.
Portion of the state space tree generated by sum of subsets



State space tree with solution

The rectangular nodes list the values of s,k and r. Circular nodes represent points at which subsets with sums m are printed out.

Solution A = (1,1,0,0,1)
Solution B = (1,0,1,1)
Solution C=(0,0,1,0,0,1)

Note that the tree contains only 23 rectangular nodes. The full space tree for n=6 contains 2^6-1=63 nodes from which calls could be made.

Knapsack Problem using Backtracking

Knapsack Problem using Backtracking can be solved as follow:

- The knapsack problem is useful in solving resource allocation problems. Let X = <x1, x2, x3, ... , xn> be the set of n items,W = <w1, w2, w3, . . . , wn> and V = <v1, v2, v3, . . . , vn> be the set of weight and value associated with each item in X, respectively. Let M be the total capacity of the knapsack, i.e. knapsack cannot hold items having a collective weight greater than M. Select items from X such that it maximizes the profit and the collective weight of selected items does not exceed the knapsack capacity. The knapsack problem has two variants. 0/1 knapsack does not allow breaking up the item, whereas a fractional knapsack does. 0/1 knapsack is also known as a binary knapsack.
- Given *n* positive weights w_i, n positive profits p_i, and a positive number M which is the knapsack capacity, the 0/1 knapsack problem calls for choosing a subset of the weights such that

$$\Sigma_{i=1 \text{ to } k} w_i x_i \leq M \text{ and}$$
$$\Sigma_{i=1 \text{ to } k} p_i x_i \text{ is maximized}$$

Bounding function is needed to help kill some live nodes without actually expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than the value of the best solution determined so far then that live node may be killed.

Here we use the fixed tuple size formulation. If at node Z the values of x_i, $1 \leq i \leq k$ have already been determined, then an upper bound for Z can be obtained by relaxing the requirement $x_i = 0$ or 1 to $0 \leq x_i \leq 1$ for $k+1 \leq i \leq n$ and use the greedy method to solve the relaxed problem.

Procedure Bound(p,w,k,M) determines an upper bound on the best solution obtainable by expanding any node Z at level k+1 of the state space tree.

The object weights and profits are W(i) and P(i).
$p = \Sigma_{i=1 \text{ to } k} P(i)X(i)$ and it is assumed that $P(i)/W(i) \geq P(i+1)/W(i+1)$, $1 \leq i \leq n$

```
procedure BOUND(p,w,k,M)
// p: the current profit total
// w: the current weight total
// k : the index of the last removed item
// M : the knapsack size
// the return result is a new profit

global  n , P(1:n) , W(1:n)
integer k, i l real b,c,p,w, M
      b := p ;  c := w
      for  i := k+1 to n do
           c := c + W(i)
           if c < M then b := b + P(j)
                    else  return (b + (1 - (c - M)/W(i))*P(i))
           endif
      repeat
      return (b)
end BOUND
```

Remark :

It follows that the bound for a feasible left child (x(k) = 1) of a node Z is the same as that for Z. Hence , the bounding function need not be used whenever the backtracking algorithm makes a move to the left child of the node. Since the backtracking algorithm will attempt make a left child move whenever given a choice between a left and right child, the bounding function need be used only after a series of successful left child moves ,(i,e, moves to feasible left child).

```
procedure Knapsack(M,n,W,P, fw,fp,X)

// M : the size of the knapsack

// n : the number of the weights and profits

// W(1:n) : the weights

// P(1:n) : the corresponding profits ;  P(i)/W(i) ≥  P(i+1)/W(i+1),

// fw : the final weight of the knapsack

// fp : the final maximum profit

// X(1:n), either zero or one ; X(k) = 0  if W(k) is not in the knapsack else X(k)
= 1

1.  integer n,k, Y(1:n), i , X(1:n) ; real M, W(1:n), P(1:n), fw, fp, cw, cp ;

2.    cw := cp := 0  ; k := 1 ; fp := -1    // cw = current weight, cp = current
profit

3.  loop

4.  while  k ≤ n  and cw + W(k) ≤ M do // place k into knapsack

5.      cw := cw + W(k) ; cp := cp + P(k) ; Y(k) := 1 ; k := k+1
```

```
repeat
if k > n then fp := cp; fw := cw ; k := n ; X := Y   // update the solution
else  Y(k) := 0     // M is exceeded so object k does not fit
endif
while BOUND(cp,cw,k,M) ≤ fp do // after fp is set above, BOUND = fp
     while k <>  0 and Y(k) <>  1 do
         k := k -1   // find the last weight included in the knapsack
     repeat
     if k = 0 then return   endif    // the algorithm ends here
     Y(k) := 0 ; cw := cw - W(k) ; cp := cp - P(k)  // remove the k-th item
 repeat
 k := k+1
 repeat
 end knapsack
```

Algorithm : Backtracking solution to the 0/1 knapsack problem

Example: Consider knapsack problem : n = 8. (W_1, W_2, W_3, W_4, W_5, W_6, W_2, W_8) = (1, 11, 21, 23, 33, 43, 45, 55), P = (11, 21, 31, 33, 43, 53, 55, 65), m = 110. Solve the problem using backtracking approach.

Solution:

Let arrange all items in non-decreasing order of p[i] / w[i].

i	p[i]	w[i]	p[i]/w[i]
1	11	1	11
2	21	11	1.90
3	31	21	1.47
4	33	23	1.43
5	43	33	1.30
6	54	43	1.23
7	55	45	1.22
8	65	55	1.18

Here M = 110 and n = 8.

Initially, cp = cw = 0, fp = – 1, k = 0

For k = 1:

cp = cp + p_1 = 0 + 11 = 11
cw = cw + w_1 = 0 + 1 = 1
cw < M, so select item 1

k = k + 1 = 2

For k=1:
cp = cp + p_2 = 11 + 21 = 32
cw = cw + w_2 = 1 + 11 = 12
cw < M, so select item 2

k = k + 1 = 2

For k = 2:
cp = cp + p_3 = 32 + 31 = 63
cw = cw + w_3 = 12 + 21 = 33
cw < M, so select item 3

k = k + 1 = 3

For k = 3:
cp = cp + p_4 = 63 + 33 = 96
cw = cw + w_4 = 33 + 23 = 56
cw < M, so select item 4

k = k + 1 = 4

For k = 4:
cp = cp + p_5 = 96 + 43 = 139
cw = cw + w_5 = 56 + 33 = 89
cw < M, so select item 5

k = k + 1 = 5

For k = 5:
cp = cp + p_6 = 139 + 53 = 192
cw = cw + w_6 = 89 + 43 = 132
cw > M, so reject item 6 and find upper bound

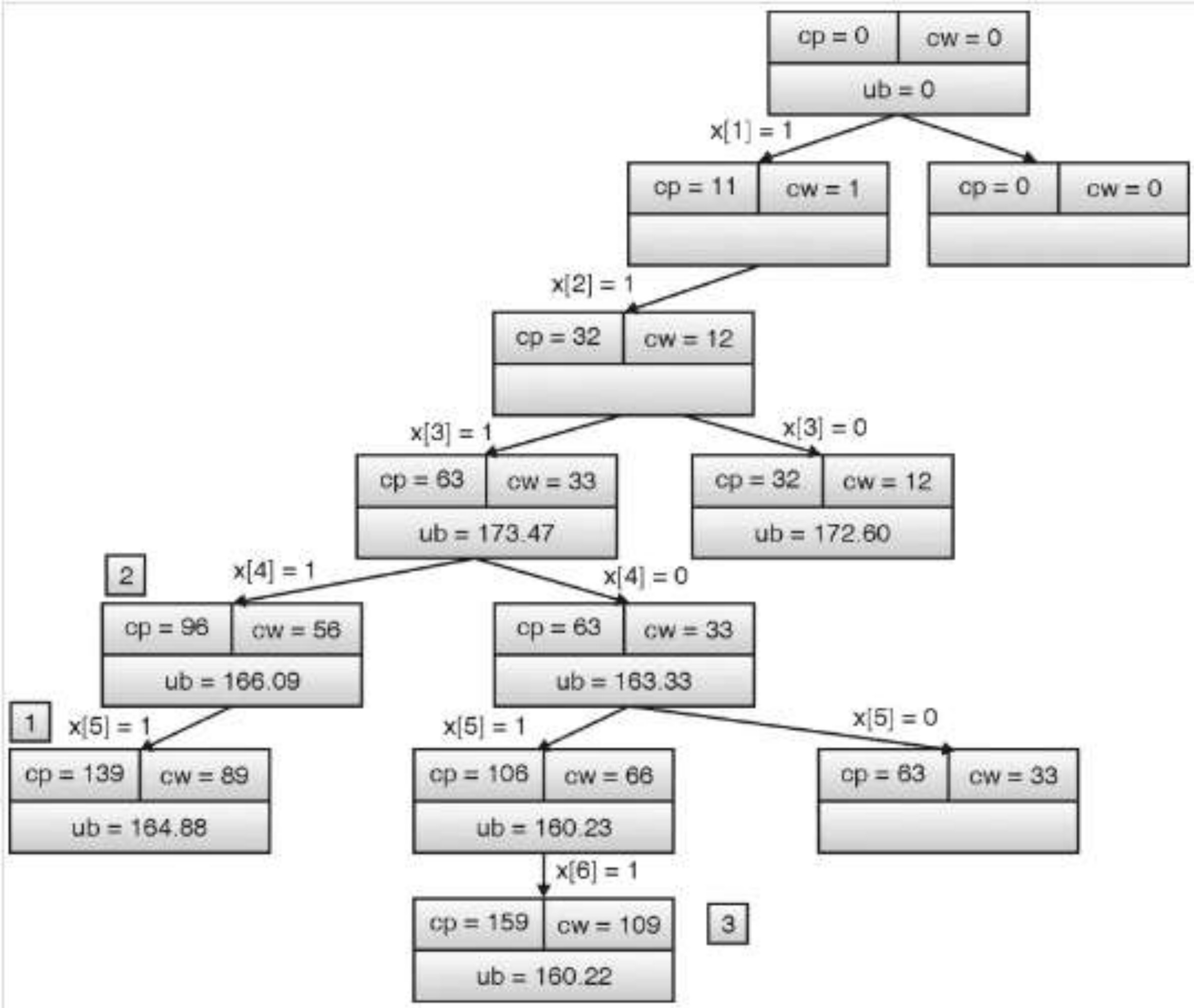
cp = cp – p_6 = 192 – 53 = 139
cw = cw – w_6 = 132 – 43 = 89
ub = cp + ((M – cw) / w_{i+1}) * p_{i+1}
b = cp + [(110 – 89) / 43] * 53 = 164.88

Inclusion of any item from {I_6, I_7, I_8} will exceed the capacity. So let's backtrack to item 4. The space tree would look like as shown in Fig. P. 6.7.2.

Upper bound at node 1:
ub = cp + ((M – cw) / w_{i+1}) * p_{i+1}
= 139 + [(110 – 89)] / 43 * 53 = 164.88

Upper bound at node 2:
= 96 + [(110 – 56) / 33] * 43 = 166.09

Upper bound at node 3:
= 63 + [(110 – 33) / 33] * 43 = 163.33


cp = 0
cw = 0
ub = 0
x[1] = 1
cp = 11
cw = 1
cp = 0
cw = 0
x[2] = 1
cp = 32
cw = 12
x[3] = 1
x[3] = 0
cp = 63
cw = 33
ub = 173.47
cp = 32
cw = 12
ub = 172.60
2
x[4] = 1
x[4] = 0
cp = 96
cw = 56
ub = 166.09
cp = 63
cw = 33
ub = 163.33
1
x[5] = 1
x[5] = 1
x[5] = 0
cp = 139
cw = 89
ub = 164.88
cp = 106
cw = 66
ub = 160.23
cp = 63
cw = 33
x[6] = 1
cp = 159
cw = 109
ub = 160.22
3