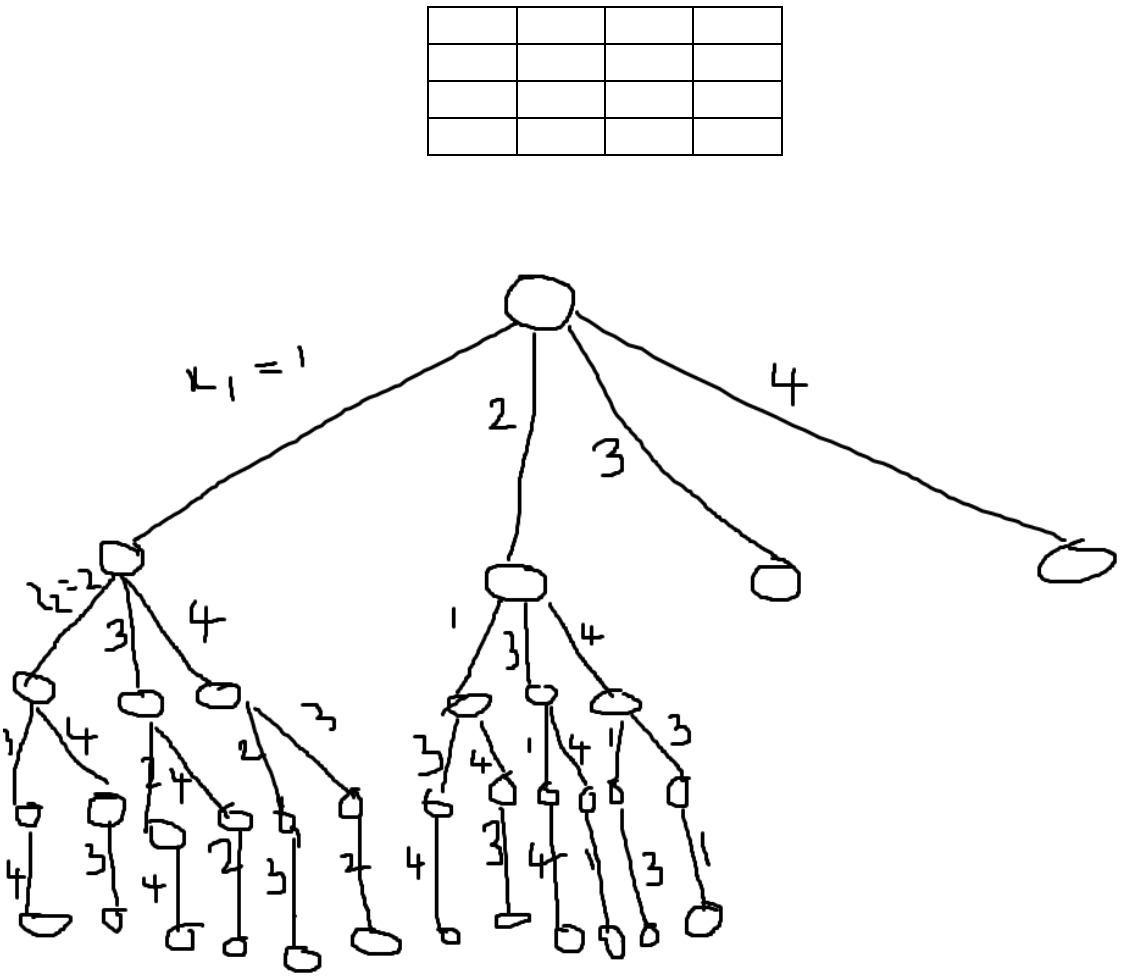
N-Queens Problem

Brute force Method (Exhaustive Search): State space tree for 4-Queens problem

Place 4 queens Q1,Q2,Q3,Q4 such that no two queens attack each other



$1+4+4\times3+4\times3\times2+4\times3\times2\times1$

$= 1+ \sum_{i=0}^{N-1} \prod_{j=0}^{i} (N-j)$

For 4 queens total nodes generated = 65

Backtracking finds solution without generating all nodes.

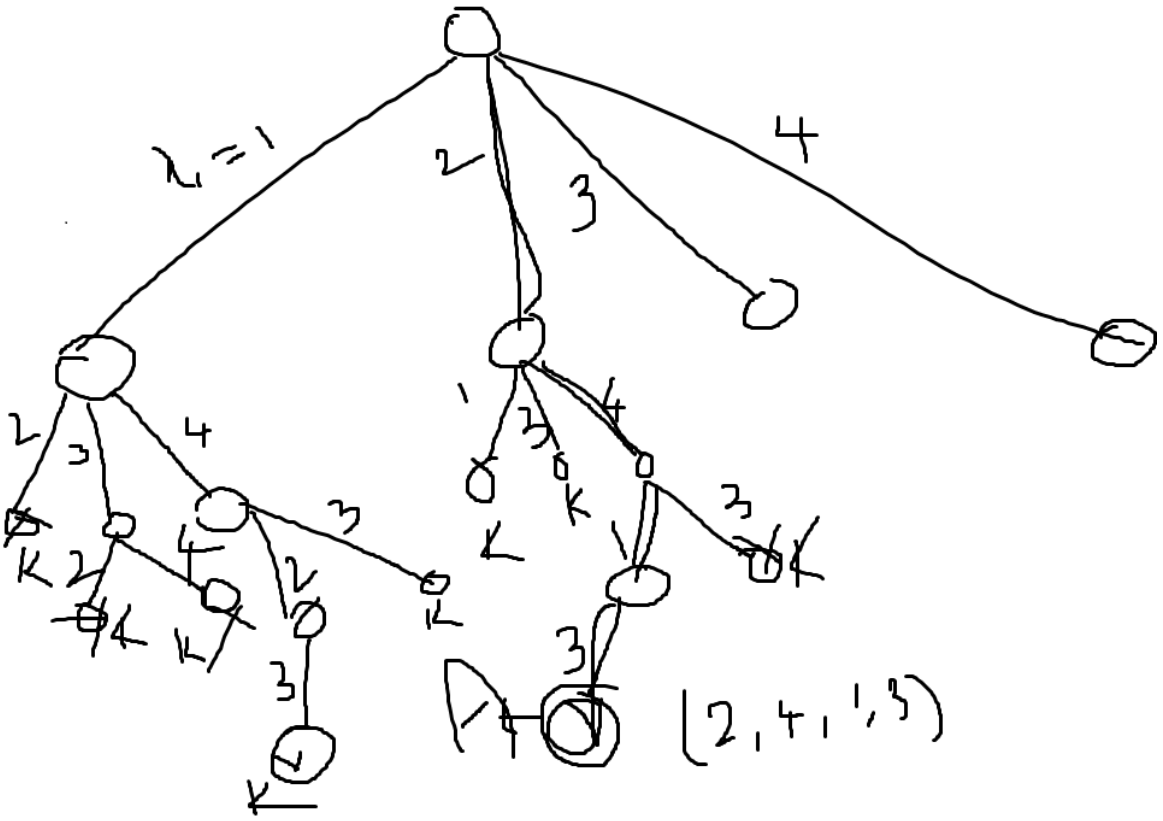
		Q1	

No two queens should be placed on the

i) Same row
ii) Same column
iii) Diagonal

In backtracking, if partially constructed solution will not lead to a solution then, it will stop exploring that path.

State Space tree



Second solution: (3,1,4,2)

Algorithm

```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 : n]);
            else NQueens(k + 1, n);
        }
    }
}
Algorithm Place(k, i)
// Returns true if a queen can be placed in kth row and
// ith column. Otherwise it returns false. x[ ] is a
// global array whose first (k − 1) values have been set.
// Abs(r) returns the absolute value of r.
{
    for j := 1 to k − 1 do
        if ((x[j] = i) // Two in the same column
            or (Abs(x[j] − i) = Abs(j − k)))
                // or in the same diagonal
            then return false;
    return true;
}
```

For 4-queens problem NQueens(1,4) is the initial call

It makes a call to Place(1,1) find Q1 can be placed in column 1

Sum of Subsets Problem

W[1:6]= {5,10,12,13,15,18}= 73

Solution= (x1,x2,x5) = (1,1,0,0,1,0)

(x3,x6) = (0,0,1,0,0,1)

Variable solution

Fixed solution

N=6, **m=30**

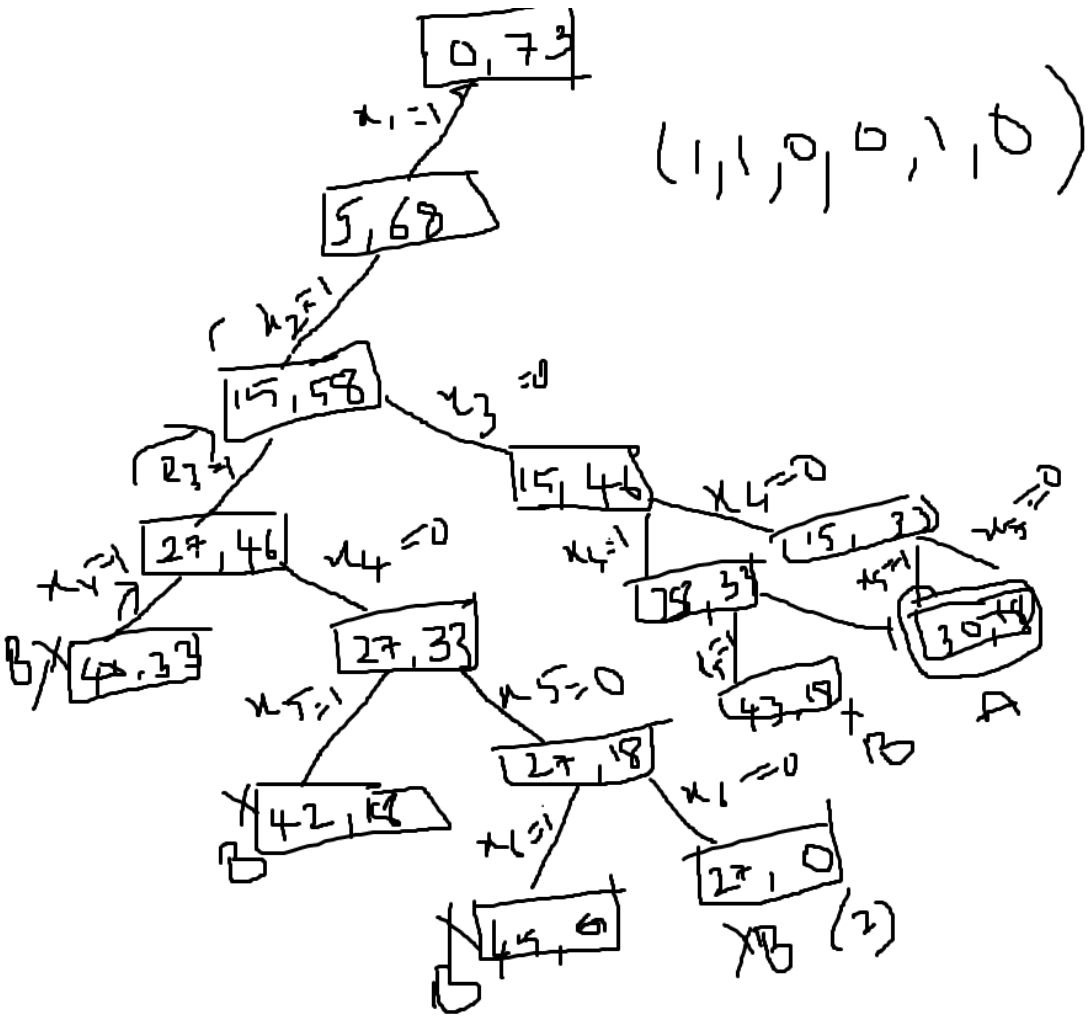
Bounding functions

1. Let k objects are already included in to the sum and when you add (k+1) object, then it should be less than total sum (m)

$$\sum_{i=1}^{k} wixi + wk + 1 \leq m$$

$$\sum_{i=1}^{k} wixi + \sum_{i=k+1}^{n} wi \geq m$$

2.



Let w={5, 7, 10, 12, 15, 18, 20} and m=35. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.

Total=87, m=35

(0,1,87)

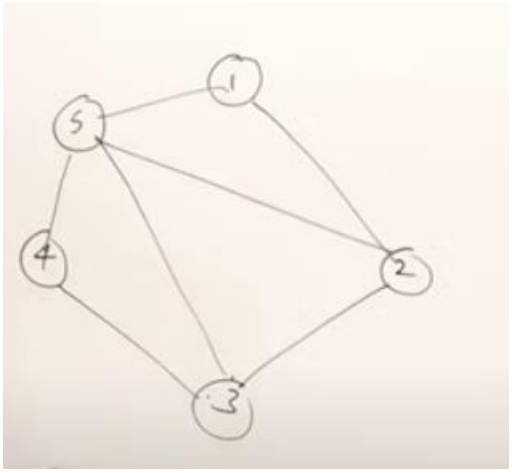
Solution vector (1,0,1,0,0,0,1)

Graph Coloring problem

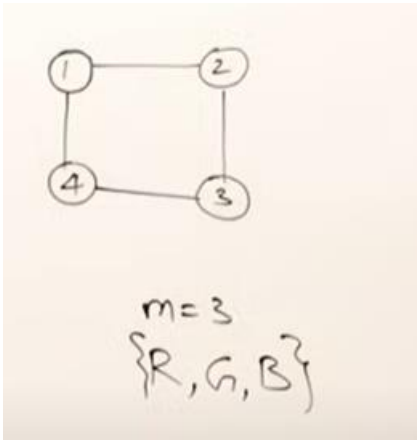
Colour the vertices of graph G=(V,E) such that no two adjacent vertices have the same colour.

Graph coloring is

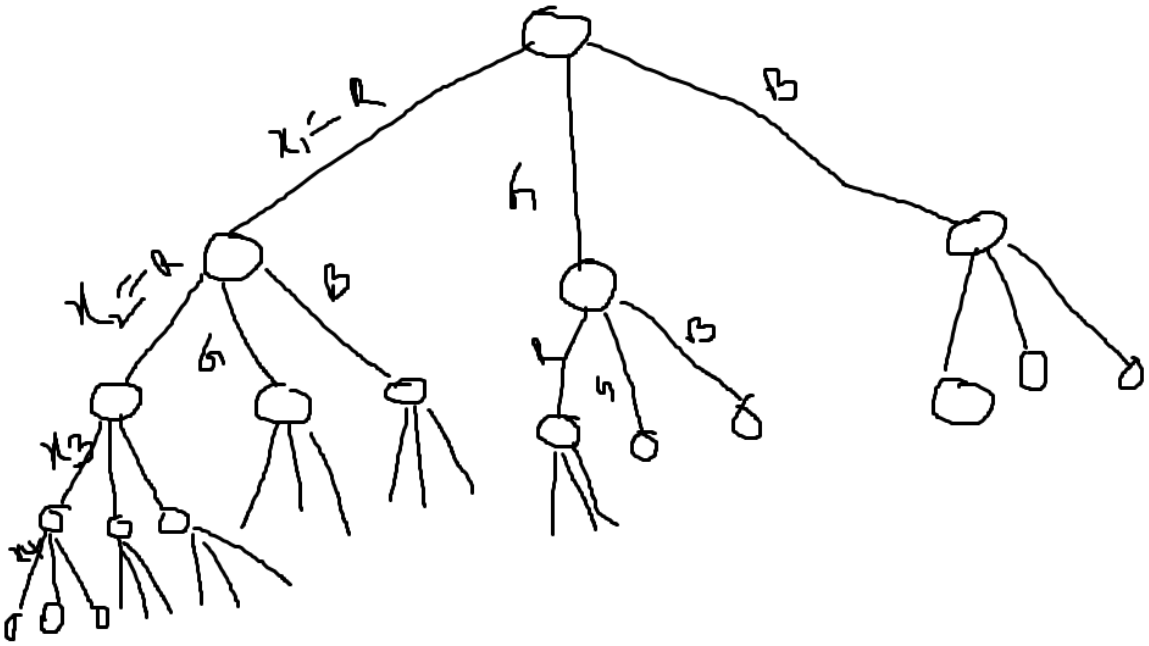
1. Optimization problem : find minimum no.of colours to colour a graph. This number is called as Chromatic number.
2. Decision problem: suppose some m no.of colours are given, we need to find out weather the graph is colourable with m colours or not.



Example:



Bruteforce method:

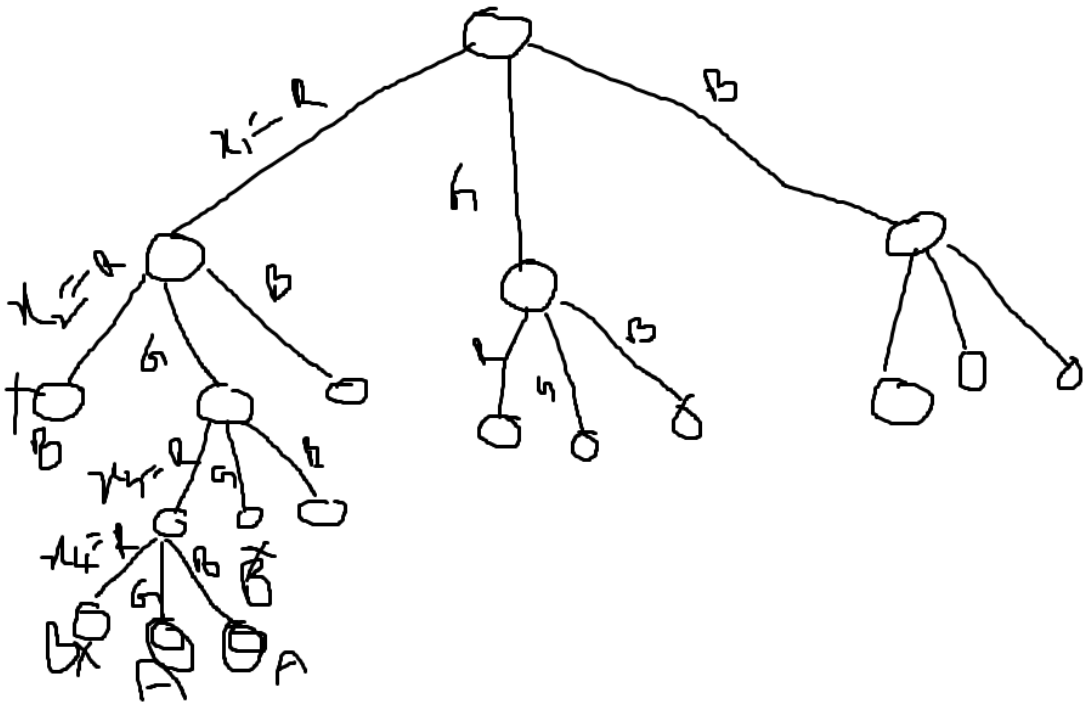


Total no.of nodes explored= 1+ 3+ (3 *3) + (3*3*3) + (3*3*3*3)=

Highest order term = 3^4

Generalize the above formula, if Chromatic number is ‘C’ then the time complexity = $O(C^n)$

Backtracking method:



Part of state space tree for graph coloring with m=3

Solutions = (R,G,R,G)

(R, G, R,B)

Algorithm:

```
Algorithm mColoring(k)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. k is the index
// of the next vertex to color.
{
    repeat
    {// Generate all legal assignments for x[k].
        NextValue(k); // Assign to x[k] a legal color.
        if (x[k] = 0) then return; // No new color possible
        if (k = n) then     // At most m colors have been
                            // used to color the n vertices.
            write (x[1 : n]);
        else mColoring(k + 1);
    } until (false);
}
```

```
Algorithm NextValue(k)
// x[1], ..., x[k − 1] have been assigned integer values in
// the range [1, m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0, m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex k. If no such color exists, then x[k] is 0.
{
    repeat
    {
        x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
        if (x[k] = 0) then return; // All colors have been used.
        for j := 1 to n do
        {   // Check if this color is
            // distinct from adjacent colors.
            if ((G[k, j] ≠ 0) and (x[k] = x[j]))
            // If (k, j) is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if (j = n + 1) then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

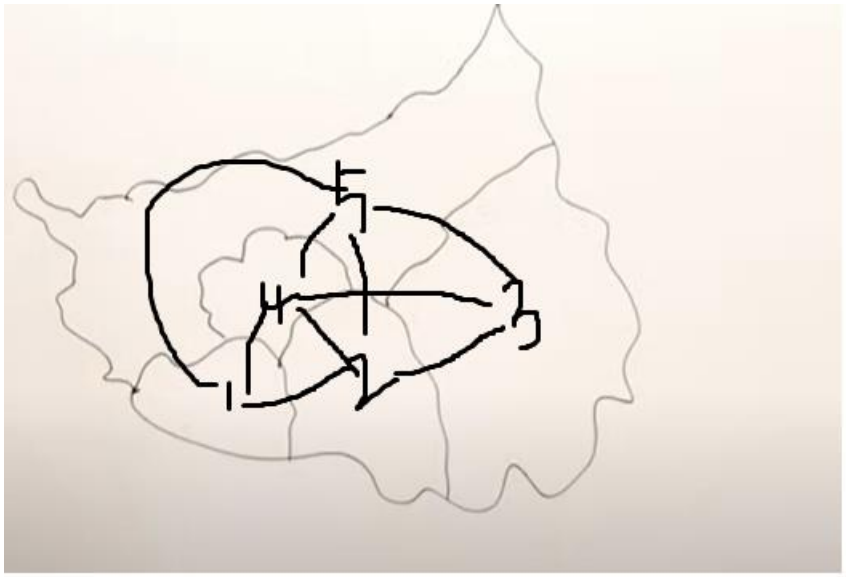
Initially mColoring is called for 1st vertex as mColoring(1). Nextvalue assigns a value to the vertex 1. No colour assignment is possible it will return 0.

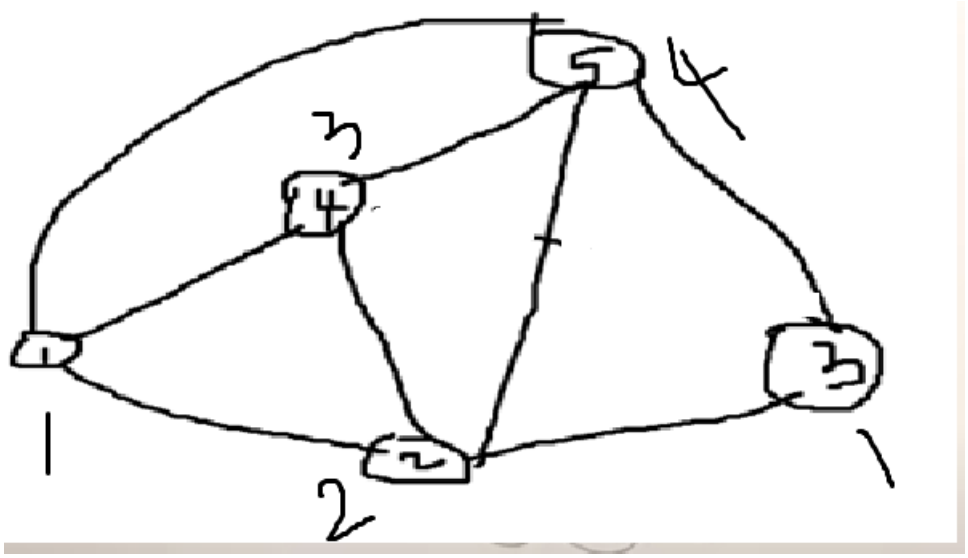
M=2

0,1,2

K=

Applications: Map colouring, Register allocation





Find the minimum no.of colours required to colour this graph.

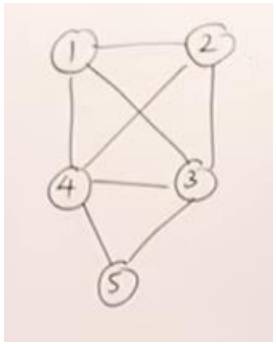
X=X+8

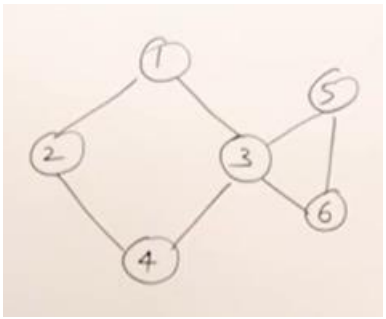
Y=X+Y-4

Z=2X+3Y

Hamiltonian Cycles

Find all the cycles by starting at a vertex and visit the remaining vertices exactly once and come back to the starting vertex.





(a) **(b)**

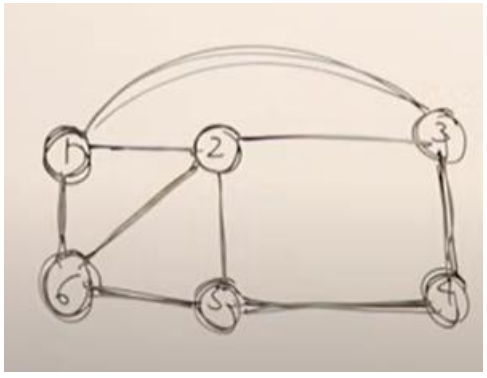
In graph (a) we have more than one Hamiltonian cycles

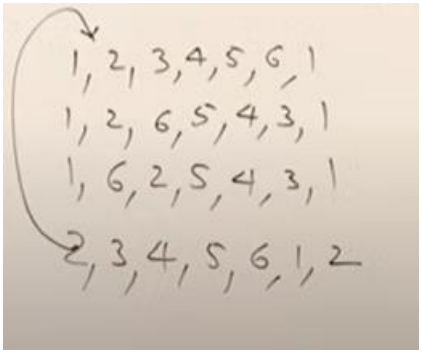
1-4-5-3-2-1

1-2-3-5-4-1

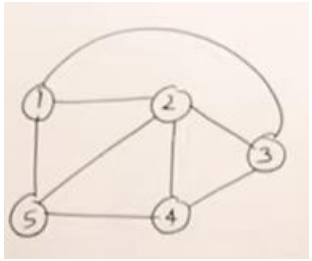
In graph (b) there is no Hamiltonian cycle.

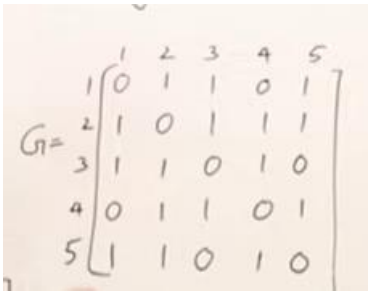
When a graph G=(V,E) is given, find Hamiltonian cycle existing.





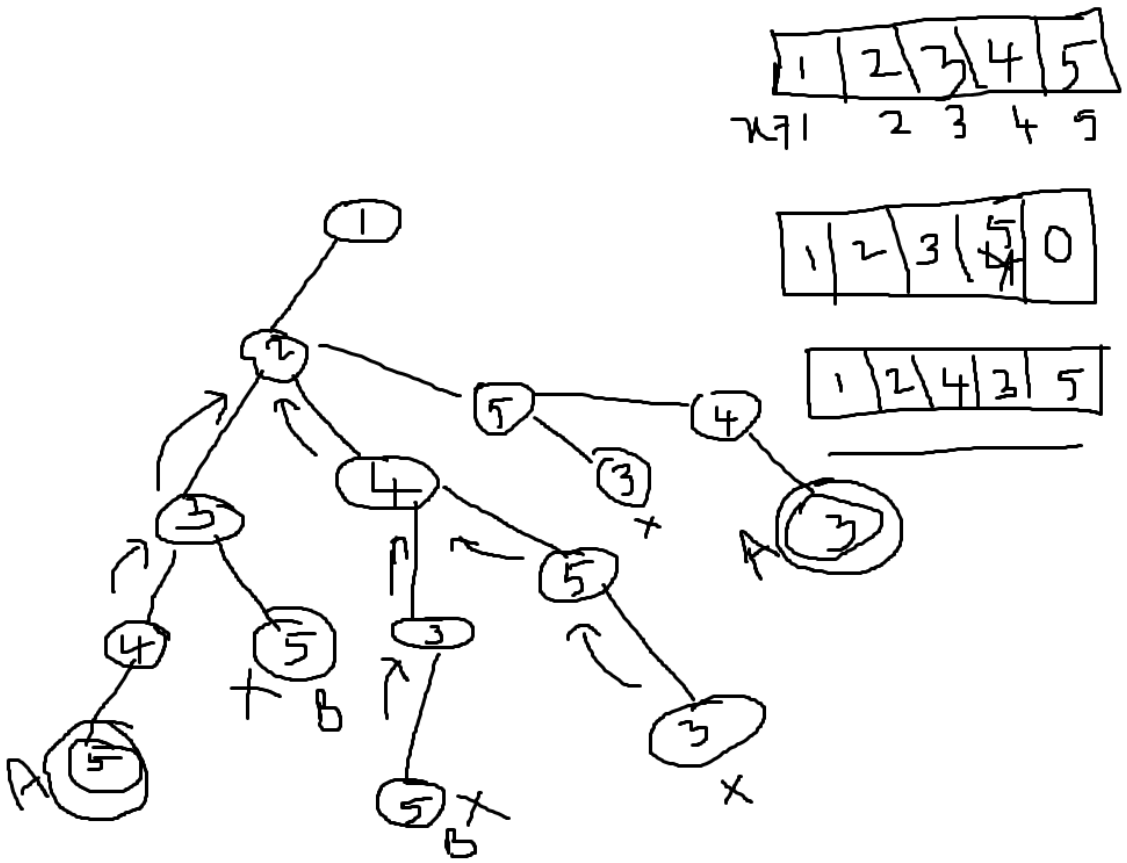
Example:





1	2	3	4	5

solution vector x



Hamiltonian cycles: (1,2,3,4,5,1)

(1,2,5,4,3,1)

Hamiltonian(2) is the first call as we start from the vertex 1.

Algorithm

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1 : n, 1 : n]. All cycles begin at node 1.
{
    repeat
    { // Generate values for x[k].
        NextValue(k); // Assign a legal next value to x[k].
        if (x[k] = 0) then return;
        if (k = n) then write (x[1 : n]);
        else Hamiltonian(k + 1);
    } until (false);
}
```

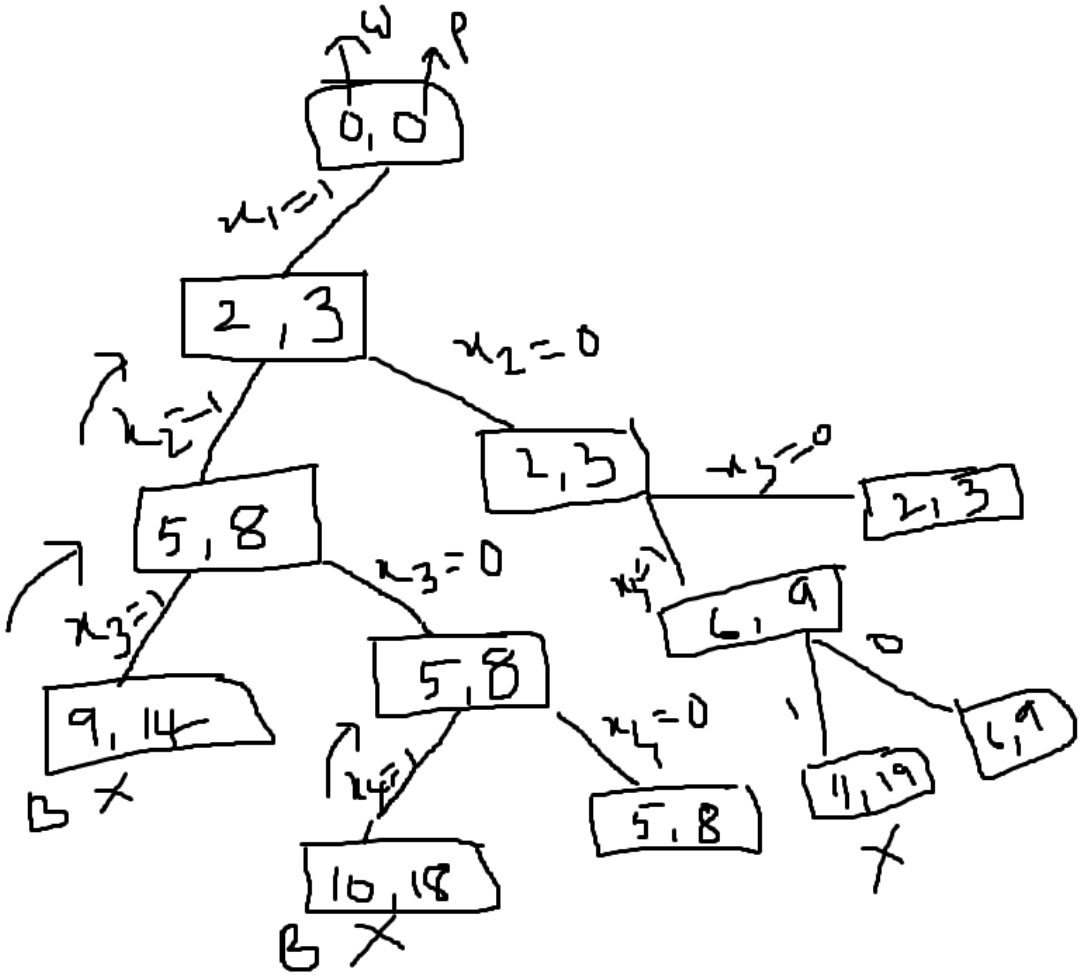
```
Algorithm NextValue(k)
// x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
// no vertex has as yet been assigned to x[k]. After execution,
// x[k] is assigned to the next highest numbered vertex which
// does not already appear in x[1 : k − 1] and is connected by
// an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
// in addition x[k] is connected to x[1].
{
    repeat
    {
        x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
        if (x[k] = 0) then return;
        if (G[x[k − 1], x[k]] ≠ 0) then
        { // Is there an edge?
            for j := 1 to k − 1 do if (x[j] = x[k]) then break;
                                // Check for distinctness.
            if (j = k) then // If true, then the vertex is distinct.
                if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
                    then return;
        }
    } until (false);
}
```

0/1 Knapsack problem with Backtracking

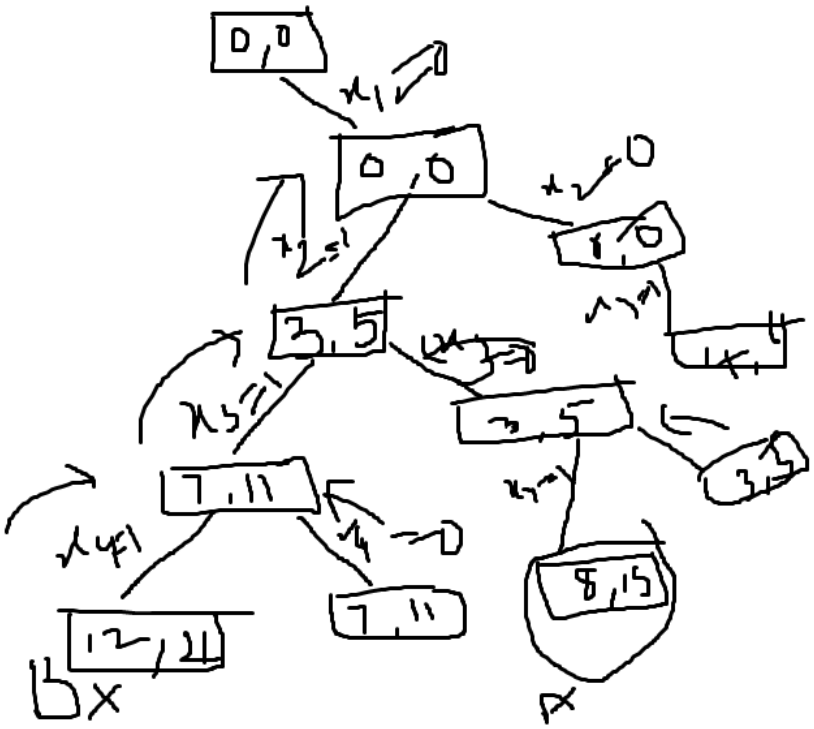
(w1,w2,w3,w4) = (2,3,4,5)

(p1,p2,p3,p4) = (3,5,6,10), m=8, n=4

Find all the solutions where the sum of weights included ≤m and find optimal solution where profit is maximum.



(0,1,0,1)



Branch and Bound

GENERAL METHOD

The backtracking algorithm is effective for decision problems, but it is not designed for optimization problems.

This drawback is refined in case of branch and bound technique. In this we use bounding function, i.e. similar to backtracking.

The difference between backtracking and branch and bound is – In backtracking, if we get a solution then we will terminate the search procedure, where as in branch and bound, we will continue the process until we get an optimal solution. Branch and bound is applicable only for minimization problems.

Definition: Generation of nodes in BFS with bounding function is called Branch and Bound.

The main difference between Backtracking and Branch and Bound

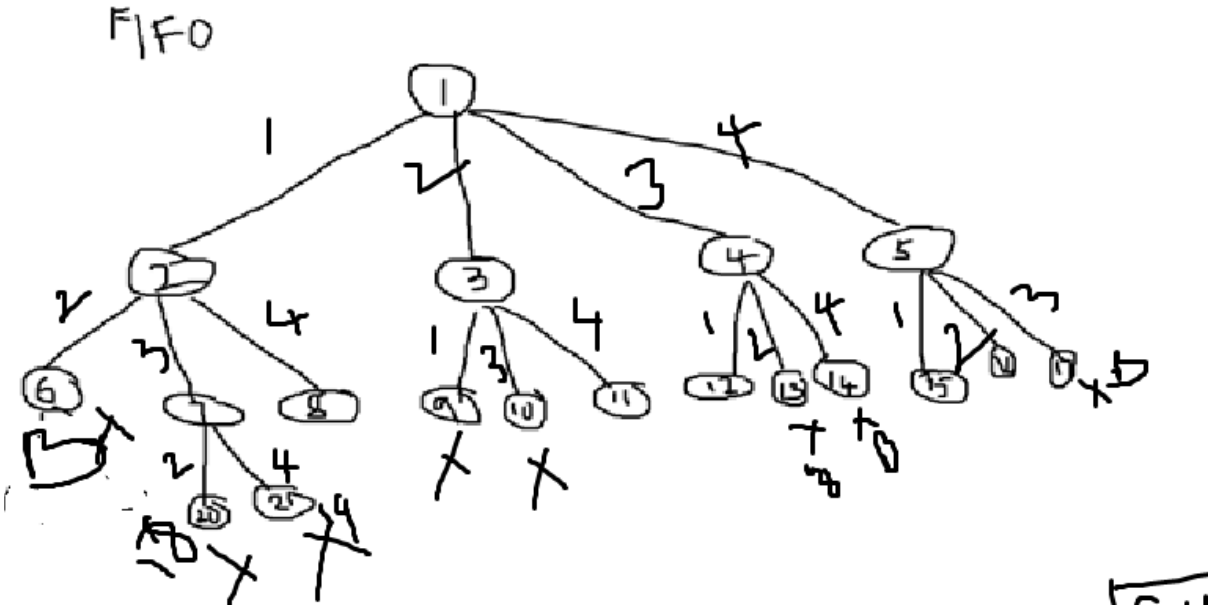
• In backtracking, we have to generate one child at a time for the current node by using DFS, and check the conditions for the current solution. If the conditions are not satisfied then bound the node and move back to the previous step otherwise move to next step.

• In Branch and Bound, we have to generate all children for the current parent node and check the condition in BFS method. If it is not satisfied then bound the solution.

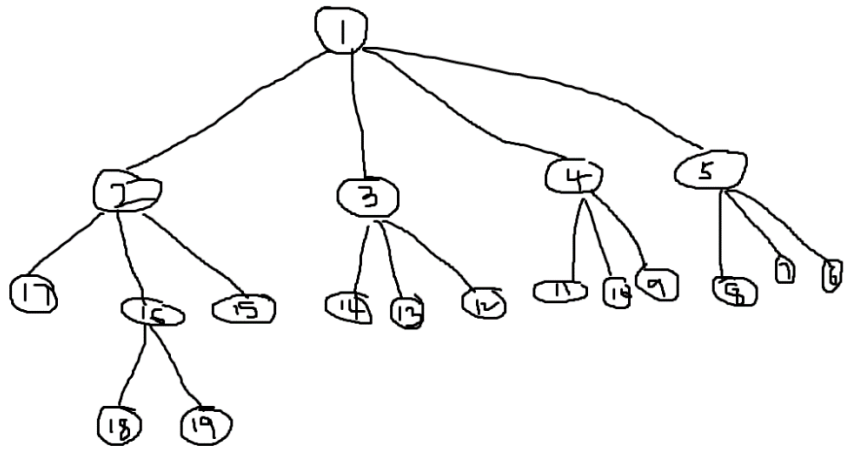
• In Branch and Bound, a BFS like state space search will be called FIFO. Search as the list of live nodes is a FIFO list.

• A D search like state space tree- search will be called LIFO. Search as the list of live nodes is a LIFO list.

• FIFO – first generate all child nodes for the root node and select the node which is first generated and repeat the process.



LIFO BB for 4-Queens problem



Least Cost Search

• In branch and bound method the basic idea is selection of E-node. The selection of E-node should so perfect that we will reach to answer node quickly.

• Using FIFO and LIFO branch and bound method the selection of E-node is very complicated and somewhat blind.

• For speeding up the search process we need to intelligent ranking function for live nodes. Each time, the next E-node is selected on the basis of this ranking function. For this ranking function additional computation (normally called as cost) is needed to answer node from the live node.

• The Least Cost (LC) search is a kind of search in which least cost is involved for reaching to answer node. At each E-node the probability of being an answer node is checked.

• BFS and D-search are special cases of LC search.

• Each time the next E-node is selected on the basis of the ranking function (smallest c^(x)). Let g^(x) be an estimate of the additional effort needed to reach an answer node from x. Let h(x) to be the cost of reaching x from the root and f(x) to be any non-decreasing function such that

• c^(x) = f(h(x)) + g^(x)

• If we set g^(x) = 0 and f(h(x)) to be level of node x then we have BFS.

• If we set f(h(x)) = 0 and g^(x) ≤ g^(y) whenever y is a child of x then the search is a Dsearch.

• In LC search, the cost function c(.) can be defined as

i) If x is an answer node the c(X) is the cost computed by the path from x to root in the state space tree.

ii) If x is not an answer node such that sub tree of x node is also not containing the answer node then c(x)= infinite

iii. Otherwise c(x) is equal to the cost of minimum cost answer node in subtree x. c^(.) with f(h(x)) = h(x) can be an approximation of c(.).

Bounding

• As we know that the bounding functions are used to avoid the generation of sub trees that do not contain the answer nodes. In bounding lower bounds and upper bounds are generated at each node

A cost function c^(x) is such that c^(x) ≤ c(x) is used to provide the lower bounds on solution obtained from any node x.

• Let upper is an upper bound on cost of minimum-cost solution. In that case, all the live nodes with c^(x) > upper can be killed.

• At the start the upper is usually set to α. After generating the children of current E-node, upper can be updated by minimum cost answer node. Each time a new answer node can be obtained.

```
listnode = record { listnode *next, *parent; float cost; }

{

Algorithm LCSearch(t)

 //Search t for an answer node.

{

if *t is an answer node then output *t and return;

 E : = t; //E-node

Initialize the list of live nodes to be empty;

repeat

{

 for each child x of E do
```

```
        {
                if x is an answer node then output the path
                from x to t and return;
                Add(x); //x is a new live node.
                (x→parent) := E; //Pointer for path to root.
        }
        if there are no more live nodes then
        {
                write ("No answer node"); return;
        }
         E:=Least();
      } until(false);
}
```

0/1 Knapsack problem with LCBB

LC BRANCH AND BOUND SOLUTION

The LC Branch and Bound solution can be obtained using fixed tuple size formulation. The steps to be followed for LCBB solution are

1. Draw the state space tree.

2. Compute C^(.) and u(.) for each node.

3. If C^(x) > upper kill node x.

4. Otherwise, the minimum cost c^(x) becomes E-node. Generate children for E-node.

5. Repeat step 3 and 4 until all the nodes get covered.

6. The minimum cost c^(x) becomes the answer node. Trace the path in backward direction from x to root for solution subset

Example:

Consider knapsack instance n=4 with capacity m = 15

Object j	p_i	w_i
1	10	2
2	10	4
4	12	6
4	18	9

Solution:

u() is computed by only considering integer part of the objects

c() is computed by considering fractional part of the objects

Algorithm: for computing u(x)

```
Algorithm U_Bound(total_profit, total_wt, k, m)
{
        pt := total_profit;
        wt := total_wt;
        for (i := k+1 to n) do
        {
                if (wt + w[i] <= m) then
                {
                pt := pt - p[i];
                wt := wt + w[i];
                }
        }
        return pt;
}
```

Travelling Sales Person Problem

Tour(x) is the path that begins at root and reaching to node x in a state space tree and returns to root. In Branch and Bound strategy cost of each node x is computed.

The TSP is solved by choosing the node with optimum cost. Hence, c(x) = cost of tour(x) where x is a leaf node.

The c^(x) is the approximation cost along the path from the root to x.

G=(V,E), adjacency matrix {i,j €V , C(i,j)= a, (i,j) € E, otherwise C(i,j)= α }

Row Reduction: To understand solving of TSP using Branch and Bound approach we will reduce the cost of the cost matrix M, by using following formula.

$$\text{Red\_Row}(M) = [M_{ij} - \min \{ M_{ij} \mid 1 \le j \le n \}] \text{ where } M_{ij} < \alpha$$

Column Reduction: Now we will reduce the matrix by choosing minimum from each column. The formula for column reduction of matrix is

$$\text{Red\_Col}(M) = [M_{ji} - \min \{ M_{ji} \mid 1 \le j \le n \}] \text{ where } M_{ji} < \alpha$$

Full Reduction:

Let M be the cost matrix for TSP for n vertices then M is called reduced if each row and column consists of either entirely α entries or else contain at least one zero. The full reduction can be achieved by applying both row reduction and column reduction. Thus total reduced cost will be = cost (Red_Row(M)) + cost (Red_Col(M))

Dynamic Reduction:

We obtained the total reduced cost as 25. That means all tours in the original graph have a length at least 25. Using dynamic reduction we can take the choice of edge i→j with optimum cost.

Steps in dynamic reduction technique:

1. Draw a state space tree with optimum cost at root node.

2. Obtain the cost of matrix for path i→j by making ithrow and jth column entries as α. Also set M[i][j] = α

3. Cost of corresponding node x with path i, j, is optimum cost + reduced cost + M[i][j].

4. Set node with minimum cost as E-node and generate its children. Repeat step 1 to 4 for completing tour with optimum cost.

	α	20	30	10	11
	15	α	16	4	2
G =	3	5	α	2	4
	19	6	18	α	3
	16	4	7	16	α

Row reduction: R1-10, R2-2, R3-2, R4-3, R5-4

total cost of row reduction= 10+2+2+3+4 = 21

Row reduced matrix is:

α	10	20	0	1
13	α	14	2	0
1	3	α	0	2
16	3	15	α	0
12	0	3	12	α

Column Reduction: C1-1, C3-3

Total cost of column reductions= 1 +3=4

Column reduced matrix:

α	10	17	0	1
12	α	11	2	0
0	3	α	0	2
15	3	12	α	0
11	0	0	12	α

Total reduction cost C(1)= 21 + 4= 25

Dynamic reduction: start at vertex 1 and visit 2,3,4,5 based on the minimum cost

Set 1st row and 2nd column to α and (2,1) to α

α	α	α	α	α
α	α	11	2	0
0	α	α	0	2
15	α	12	α	0
11	α	0	12	α

Check weather the resulting matrix is having zero's in all rows and columns. Otherwise reduce the matrix. Here all rows and columns have zero.

C(2) = C(1) + Row reduction + Column reduction + G(1,2) = 25 +0+0+10= 35

Set 1st row and 3rd column to α and (3,1) to α

α	α	α	α	α
1	α	α	2	0
α	3	α	0	2
4	3	α	α	0
0	0	α	12	α

Check weather the resulting matrix is having zero's in all rows and columns. Otherwise reduce the matrix.

C1-11

C(3)= C(1) + Row reduction + Column reduction + G(1,3) = 25 +0+11+17 =53

Set 1st row and 4th column to α and (4,1) to α

α	α	α	α	α
12	α	11	α	0
0	3	α	α	2
α	3	12	α	0
11	0	0	α	α

C(4)= C(1) +G(1,4) = 25+0= 25

Set 1st row and 5th column to α and (5,1) to α

α	α	α	α	α
10	α	9	0	α
0	3	α	0	α
15	3	12	α	α
α	0	0	12	α

R2-2

C(5)= C(1) + RR + G(1,5)= 25+ 2 +1= 28

Set 4th row and 2nd column to α and (2,1) to α in the matrix at node 4

α	α	α	α	α
α	α	11	α	0
0	α	α	α	2
α	α	α	α	α
11	α	0	α	α

C(6)= C(4) + G(1,2) = 25+10= 35

Set 4th row and 3rd column to α and (3,1) to α in the matrix at node 4

α	α	α	α	α
1	α	α	α	0
α	1	α	α	0
α	α	α	α	α
0	0	α	α	α

R3-2

C1-11

C(7)= C(4) + RR + CR+ G(1,3) = 25+ 2+11+ 17 = 55

Set 4th row and 5th column to α and (5,1) to α in the matrix at node 4

α	α	α	α	α
1	α	0	α	α
0	3	α	α	α
α	α	α	α	α
α	0	0	α	α

R2-11

C(8)= C(4)+RR+G(1,5) = 25+ 11+1= 37

Set 2nd row and 3rd column to α and (3,1) to α in the matrix at node 6

α	α	α	α	α
α	α	α	α	α
α	α	α	α	0
α	α	α	α	α
0	α	α	α	α

R3-2, R5-11

C(9) = 35+ 13+ 17= 65

Set 2nd row and 5th column to α and (5,1) to α in the matrix at node 6

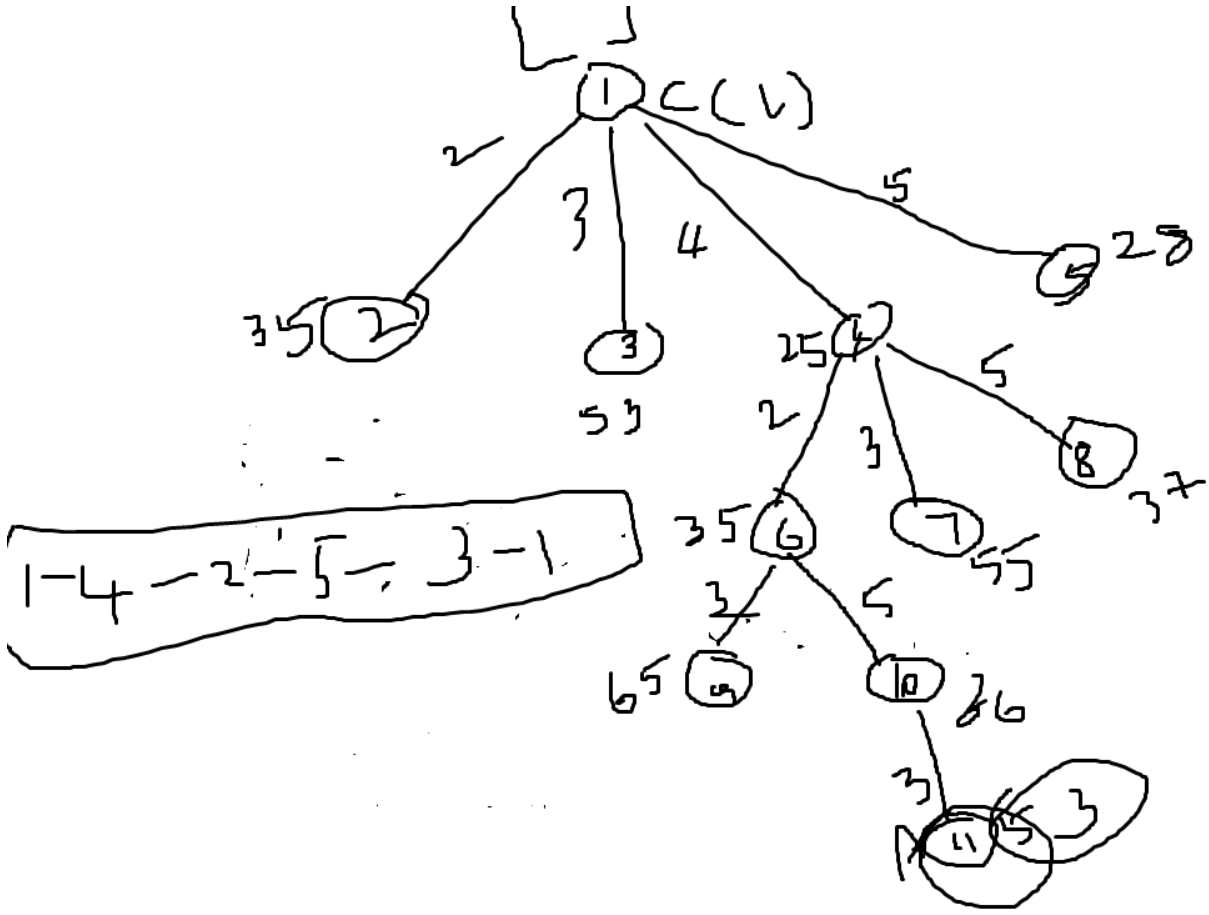
α	α	α	α	α
α	α	α	α	α
0	α	α	α	α
α	α	α	α	α
α	α	0	α	α

C(10)= 35+1= 36

Set 5th row and 3rd column to α and (3,1) to α in the matrix at node 10

α	α	α	α	α
α	α	α	α	α
α	α	α	α	α
α	α	α	α	α
α	α	α	α	α

C(11)= 36+17= 53


C(1)
2
3
4
5
35
53
25
55
37
65
86
1-4-2-5-3-1