UNIT - V

Backtracking:

The general method—8 queens problem—Sum of subsets—Graph coloring—Hamiltonian cycle—Knapsack problem.

BACKTRACKING

- It is one of the most general algorithm design techniques.

- Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation.

- To apply backtracking method, tne desired solution must be expressible as an n-tuple (x1…xn) where xi is chosen from some finite set Si.

- The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn).

- The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (m_{i+1}………..m_n) possible test vectors may be ignored entirely.

- Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.

- These constraints are classified as:

i) Explicit constraints.
ii) Implicit constraints.

1) Explicit constraints:

Explicit constraints are rules that restrict each Xi to take values only from a given set.

Some examples are,

$Xi \geq 0$ or Si = {all non-negative real nos.}
Xi =0 or 1 or Si={0,1}.
$Li \leq Xi \leq Ui$ or Si= {a: $Li \leq a \leq Ui$}

- All tupules that satisfy the explicit constraint define a possible solution space for I.

2) Implicit constraints:

The implicit constraint determines which of the tuples in the solution space I can actually satisfy the criterion functions.

Algorithm:

```
Algorithm IBacktracking (n)
// This schema describes the backtracking procedure .All solutions are generated in X[1:n]
//and printed as soon as they are determined.
 {
   k=1;
   While (k ≠ 0) do
   {
     if (there remains all untried
     X[k] ∈ T (X[1],[2],…..X[k-1]) and Bk (X[1],…..X[k])) is true ) then
     {
       if(X[1],……X[k] )is the path to the answer node)
       Then write(X[1:k]);
       k=k+1;             //consider the next step.
     }
   else k=k-1;              //consider backtracking to the previous set.
 }
}
```

- All solutions are generated in X[1:n] and printed as soon as they are determined.

- T(X[1]…..X[k-1]) is all possible values of X[k] gives that X[1],……..X[k-1] have already been chosen.

- B_k(X[1]………X[k]) is a boundary function which determines the elements of X[k] which satisfies the implicit constraint.

- Certain problems which are solved using backtracking method are,

    1. Sum of subsets.
    2. Graph coloring.
    3. Hamiltonian cycle.
    4. N-Queens problem.

SUM OF SUBSETS:

- We are given 'n' positive numbers called weights and we have to find all combinations of these numbers whose sum is M. this is called sum of subsets problem.

- If we consider backtracking procedure using fixed tuple strategy , the elements X(i) of the solution vector is either 1 or 0 depending on if the weight W(i) is included or not.

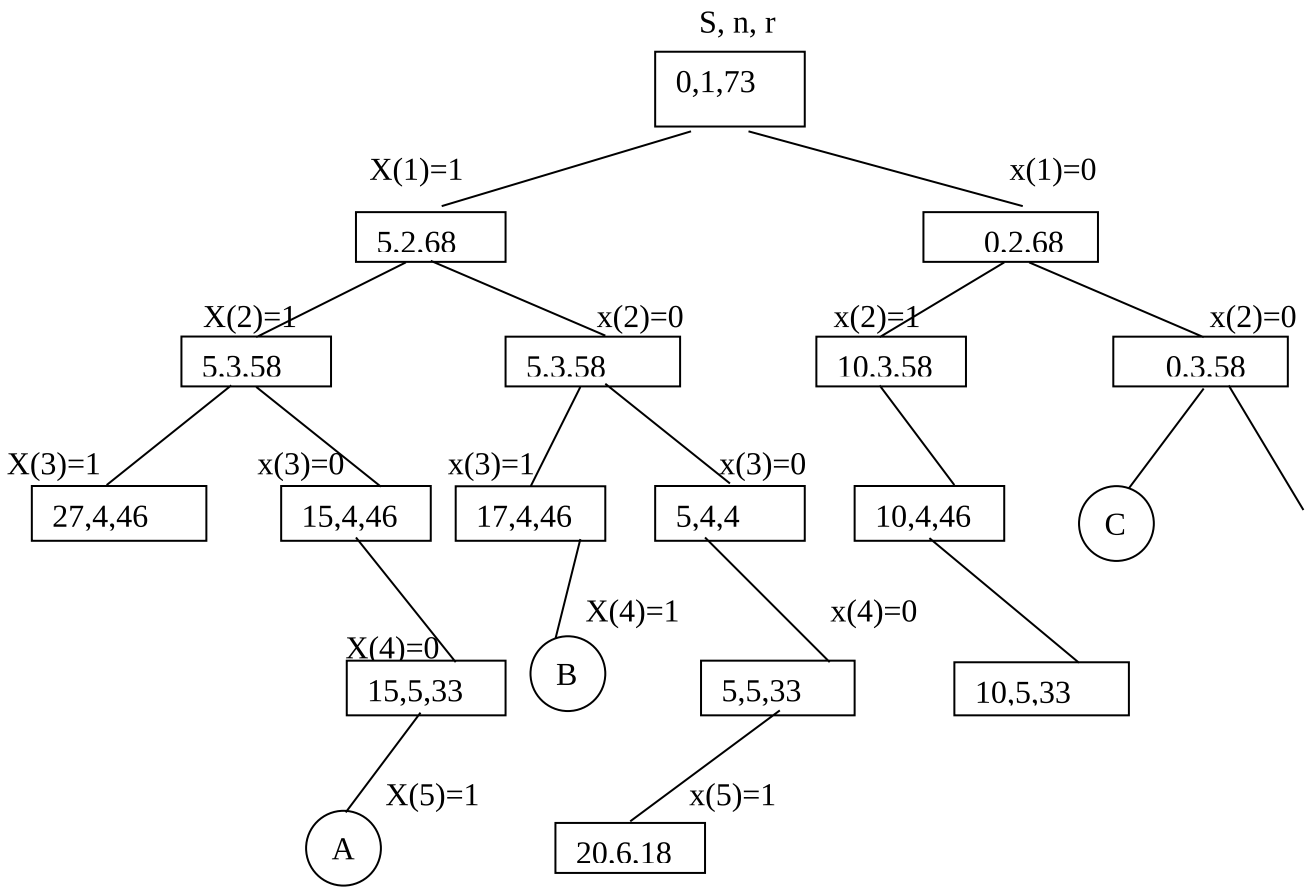
- If the state space tree of the solution, for a node at level I, the left child corresponds to X(i)=1 and right to X(i)=0.

Example:

- Given n=6,M=30 and W(1…6)=(5,10,12,13,15,18).We have to generate all possible combinations of subsets whose sum is equal to the given value M=30.

- In state space tree of the solution the rectangular node lists the values of s, k, r, where s is the sum of subsets,'k' is the iteration and 'r' is the sum of elements after 'k' in the original set.

- The state space tree for the given problem is,



I st solution is **A** -> 1 1 0 0 1 0
II nd solution is **B** -> 1 0 1 1 0 0

III rd solution is **C** -> 0 0 1 0 0 1

- In the state space tree, edges from level 'i' nodes to 'i+1' nodes are labeled with the values of Xi, which is either 0 or 1.
- The left sub tree of the root defines all subsets containing Wi.
- The right subtree of the root defines all subsets, which does not include Wi.

GENERATION OF STATE SPACE TREE:

- Maintain an array X to represent all elements in the set.
- The value of Xi indicates whether the weight Wi is included or not.
- Sum is initialized to 0 i.e., s=0.
- We have to check starting from the first node.
- Assign X(k)<- 1.
- If S+X(k)=M then we print the subset b'coz the sum is the required output.
- If the above condition is not satisfied then we have to check S+X(k)+W(k+1)<=M. If so, we have to generate the left sub tree. It means W(t) can be included so the sum will be incremented and we have to check for the next k.
- After generating the left sub tree we have to generate the right sub tree, for this we have to check S+W(k+1)<=M.B'coz W(k) is omitted and W(k+1) has to be selected.
- Repeat the process and find all the possible combinations of the subset.

Algorithm:

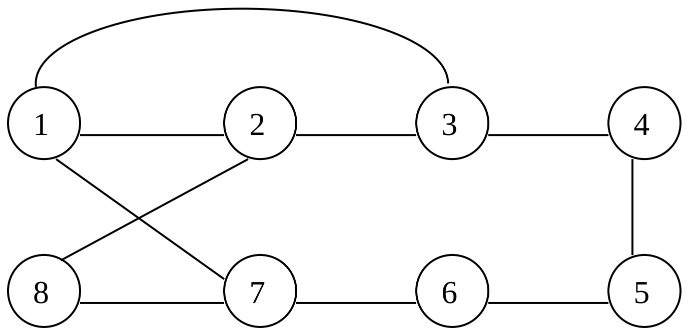
```
Algorithm sumofsubset(s,k,r)
{
//generate the left child. note s+w(k)<=M since Bk-1 is true.
X{k]=1;
If (S+W[k]=m) then write(X[1:k]); // there is no recursive call here as W[j]>0,1<=j<=n.
Else if (S+W[k]+W[k+1]<=m) then sum of sub (S+W[k], k+1,r- W[k]);
//generate right child and evaluate Bk.
If ((S+ r- W[k]>=m)and(S+ W[k+1]<=m)) then
{
  X{k]=0;
  sum of sub (S, k+1, r- W[k]);
}
}
```

HAMILTONIAN CYCLES:

- ❖ Let G=(V,E) be a connected graph with 'n' vertices. A HAMILTONIAN CYCLE is a round trip path along 'n' edges of G which every vertex once and returns to its starting position.

- ❖ If the Hamiltonian cycle begins at some vertex V1 belongs to G and the vertex are visited in the order of V1,V2…….Vn+1,then the edges are in E,1<=I<=n and the Vi are distinct except V1 and Vn+1 which are equal.

- ❖ Consider an example graph G1.



The graph G1 has Hamiltonian cycles:

->1,3,4,5,6,7,8,2,1 and
->1,2,8,7,6,5,4,3,1.

- ❖ The backtracking algorithm helps to find Hamiltonian cycle for any type of graph.

Procedure:

1. Define a solution vector X(Xi……..Xn) where Xi represents the I th visited vertex of the proposed cycle.

2. Create a cost adjacency matrix for the given graph.

3. The solution array initialized to all zeros except X(1)=1,b'coz the cycle should start at vertex '1'.

4. Now we have to find the second vertex to be visited in the cycle.

5. The vertex from 1 to n are included in the cycle one by one by checking 2 conditions,

    1.There should be a path from previous visited vertex to current vertex.
    2.The current vertex must be distinct and should not have been visited earlier.

6. When these two conditions are satisfied the current vertex is included in the cycle, else the next vertex is tried.

7. When the nth vertex is visited we have to check, is there any path from nth vertex to first 8vertex. if no path, the go back one step and after the previous visited node.

8. Repeat the above steps to generate possible Hamiltonian cycle.

Algorithm:(Finding all Hamiltonian cycle)

```
Algorithm Hamiltonian (k)
{
 Loop
        Next value (k)
If (x (k)=0) then return;
{
  If k=n then
      Print (x)
Else
Hamiltonian (k+1);
End if

}
Repeat
}

Algorithm Nextvalue (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 distinction.
  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);
}
```

8-QUEENS PROBLEM:

This 8 queens problem is to place n-queens in an 'N*N' matrix in such a way that no two queens attack each otherwise no two queens should be in the same row, column, diagonal.

Solution:

- The solution vector X (X1…Xn) represents a solution in which Xi is the column of the th row where I th queen is placed.
- First, we have to check no two queens are in same row.
- Second, we have to check no two queens are in same column.
- The function, which is used to check these two conditions, is [I, X (j)], which gives position of the I th queen, where I represents the row and X (j) represents the column position.
- Third, we have to check no two queens are in it diagonal.
- Consider two dimensional array A[1:n,1:n] in which we observe that every element on the same diagonal that runs from upper left to lower right has the same value.
- Also, every element on the same diagonal that runs from lower right to upper left has the same value.
- Suppose two queens are in same position (i,j) and (k,l) then two queens lie on the same diagonal , if and only if |j-l|=|I-k|.

STEPS TO GENERATE THE SOLUTION:

- Initialize x array to zero and start by placing the first queen in k=1 in the first row.
- To find the column position start from value 1 to n, where 'n' is the no. Of columns or no. Of queens.
- If k=1 then x (k)=1.so (k,x(k)) will give the position of the k th queen. Here we have to check whether there is any queen in the same column or diagonal.
- For this considers the previous position, which had already, been found out. Check whether
  X (I)=X(k) for column |X(i)-X(k)|=(I-k) for the same diagonal.
- If any one of the conditions is true then return false indicating that k th queen can't be placed in position X (k).
- For not possible condition increment X (k) value by one and precede d until the position is found.

- ❖ If the position X (k)$\leq$ n and k=n then the solution is generated completely.
- ❖ If k<n, then increment the ‘k’ value and find position of the next queen.
- ❖ If the position X (k)>n then k th queen cannot be placed as the size of the matrix is ‘N*N’.
- ❖ So decrement the ‘k’ value by one i.e. we have to back track and after the position of the previous queen.

Algorithm:

```
Algorithm place (k,I)
//return true if a queen can be placed in k th row and I th column. otherwise it returns //
//false .X[] is a global array whose first k-1 values have been set. Abs® returns the
//absolute value of r.
{
  For j=1 to k-1 do
     If ((X [j]=I)              //two in same column.
     Or (abs (X [j]-I)=Abs (j-k)))
Then return false;
Return true;
}
```

Algorithm Nqueen (k,n)

```
//using backtracking it prints all possible positions of n queens in ‘n*n’ chessboard. So
 //that they are non-tracking.
{
     For I=1 to n do
         {
           If place (k,I) then
             {
                X [k]=I;
                 If (k=n) then write (X [1:n]);
                  Else nquenns(k+1,n)    ;
           }
      }
}
```

Example: 4 queens.

Two possible solutions are

	Q		
			Q
Q			
		Q	

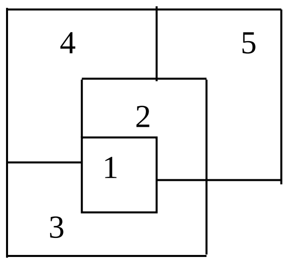
		Q	
Q			
			Q
	Q		

Solutin-1	Solution 2
(2 4 1 3)	(3 1 4 2)

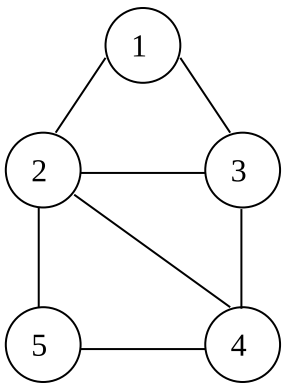
GRAPH COLORING:

- Let 'G' be a graph and 'm' be a given positive integer. If the nodes of 'G' can be colored in such a way that no two adjacent nodes have the same color. Yet only 'M' colors are used. So it's called M-color ability decision problem.
- The graph G can be colored using the smallest integer 'm'. This integer is referred to as chromatic number of the graph.
- A graph is said to be planar iff it can be drawn on plane in such a way that no two edges cross each other.
- Suppose we are given a map then, we have to convert it into planar. Consider each and every region as a node. If two regions are adjacent then the corresponding nodes are joined by an edge.

Consider a map with five regions and its graph.



1 is adjacent to 2, 3, 4.
2 is adjacent to 1, 3, 4, 5
3 is adjacent to 1, 2, 4
4 is adjacent to 1, 2, 3, 5
5 is adjacent to 2, 4



Steps to color the Graph:

- ❖ First create the adjacency matrix graph(1:m,1:n) for a graph, if there is an edge between i,j then C(i,j) = 1 otherwise C(i,j) =0.

- ❖ The Colors will be represented by the integers 1,2,…..m and the solutions will be stored in the array X(1),X(2),………..,X(n) ,X(index) is the color, index is the node.

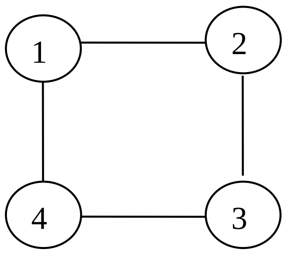
- ❖ He formula is used to set the color is,
  X(k) = (X(k)+1) % (m+1)

- ❖ First one chromatic number is assigned ,after assigning a number for 'k' node, we have to check whether the adjacent nodes has got the same values if so then we have to assign the next value.

- ❖ Repeat the procedure until all possible combinations of colors are found.

- ❖ The function which is used to check the adjacent nodes and same color is,
  If((Graph (k,j) == 1) and X(k) = X(j))

Example:



N= 4
M= 3

Adjacency Matrix:

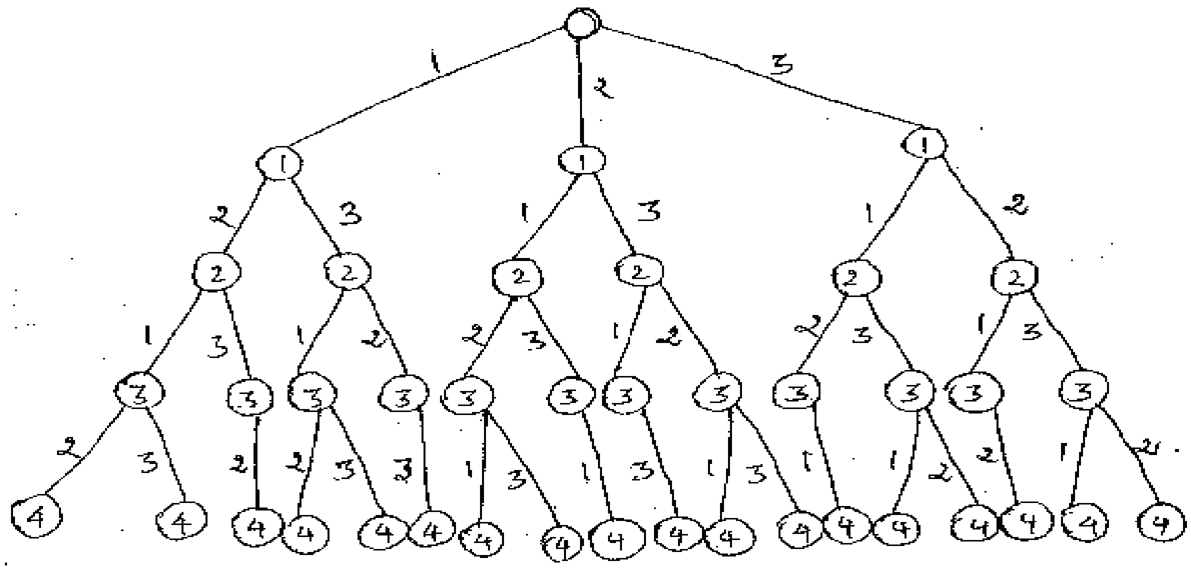
$$\begin{vmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{vmatrix}$$

→ Problem is to color the given graph of 4 nodes using 3 colors.

→Node-1 can take the given graph of 4 nodes using 3 colors.

→ The state space tree will give all possible colors in that ,the numbers which are inside the circles are nodes ,and the branch with a number is the colors of the nodes.

State Space Tree:



Algorithm:

Algorithm mColoring(k)

// 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 assignment 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                // Almost '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 values have distinct integers. A value for X[k] is determined in the //range[0,m].X[k] is assigned the next highest numbers color while maintaining //distinctness form 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 color.
         If((G[k,j] $\neq$ 0)and(X[k] = X[j]))
            // If (k,j) is an edge and if adjacent vertices have the same color.
         Then break;
         }

      if(j=n+1) then return;     //new color found.
   } until(false);    //otherwise try to find another color.
}
```

→ The time spent by Nextvalue to determine the children is θ(mn)

→Total time is = θ(m^n n).

Knapsack Problem using Backtracking:

- The problem is similar to the zero-one (0/1) knapsack optimization problem is dynamic programming algorithm.

- We are given 'n' positive weights Wi and 'n' positive profits Pi, and a positive number 'm' that is the knapsack capacity, the is problem calls for choosing a subset of the weights such that,

$$\sum_{1 \le i \le n} WiXi \le m \quad \text{and} \quad \sum_{1 \le i \le n} PiXi \text{ is Maximized.}$$

Xi →Constitute Zero-one valued Vector.

- The Solution space is the same as that for the sum of subset's problem.

- Bounding functions are needed to help kill some live nodes without 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.

- The profits and weights are assigned in descending order depend upon the ratio.

(i.e.) Pi/Wi $\geq$ P(I+1) / W(I+1)

Solution :

- After assigning the profit and weights ,we have to take the first object weights and check if the first weight is less than or equal to the capacity, if so then we include that object (i.e.) the unit is 1.(i.e.) K→ 1.

- Then We are going to the next object, if the object weight is exceeded that object does not fit. So unit of that object is ‘0’.(i.e.) K=0.
- Then We are going to the bounding function ,this function determines an upper bound on the best solution obtainable at level K+1.

- Repeat the process until we reach the optimal solution.

Algorithm:

Algorithm Bknap(k,cp,cw)

// ‘m’ is the size of the knapsack; ‘n’ → no.of weights & profits. W[]&P[] are the //weights & weights. P[I]/W[I] ≥ P[I+1]/W[I+1].
//fw→Final weights of knapsack.
//fp→ final max.profit.
//x[k] = 0 if W[k] is not the knapsack,else X[k]=1.

```
{
    // Generate left child.
     If((W+W[k] ≤m) then
     {
         Y[k] =1;
          If(k<n) then Bnap(k+1,cp+P[k],Cw +W[k])
            If((Cp + p[w] > fp) and (k=n)) then

              {
                fp = cp + P[k];
                fw = Cw+W[k];
               for j=1 to k do X[j] = Y[j];
              }
  }

 if(Bound(cp,cw,k) ≥fp) then
 {
     y[k] = 0;
    if(k<n) then Bnap (K+1,cp,cw);
   if((cp>fp) and (k=n)) then
     {
         fp = cp;
          fw = cw;
            for j=1 to k do X[j] = Y[j];
```

```
        }
    }
}
```

Algorithm for Bounding function:

```
Algorithm Bound(cp,cw,k)
// cp→ current profit total.
//cw→ current weight total.
//k→the index of the last removed item.
//m→the knapsack size.

{
    b=cp;
    c=cw;
    for I =- k+1 to n do
  {
        c= c+w[I];
      if (c<m) then b=b+p[I];
          else return b+ (1-(c-m)/W[I]) * P[I];
}
return b;
}
```

Example:

M= 6	Wi = 2,3,4		4 2 2
N= 3	Pi = 1,2,5	Pi/Wi (i.e.)	5 2 1

Xi = 1 0   1

The maximum weight is 6

The Maximum profit is (1*5) + (0*2) + (1*1)

$\rightarrow$ 5+1

$\rightarrow$ 6.

Fp = (-1)

- $1 \le 3$ & $0+4 \le 6$

  cw = 4,cp = 5,y(1) =1

  k = k+2

- $2 \le 3$ but 7>6

  so y(2) = 0

- So bound(5,4,2,6)

B=5
C=4
I=3 to 3
C=6
$6 \neq 6$
So return 5+(1-(6-6))/(2*1)

- 5.5 is not less than fp.
  So, k=k+1 (i.e.) 3.

  3=3 & $4+2 \leq 6$

  cw= 6,cp = 6, y(3)=1.
  K=4.

- If 4> 3 then
  Fp =6,fw=6,k=3 ,x(1) 1 0 1
  The solution Xi $\rightarrow$ 1 0 1

  Profit $\rightarrow$ 6
  Weight $\rightarrow$6.

UNIT – V

BRANCH AND BOUND -- THE METHOD

The design technique known as ***branch and bound*** is very similar to backtracking (seen in unit 4) in that it searches a tree model of the solution space and is applicable to a wide variety of discrete combinatorial problems.

Each node in the combinatorial tree generated in the last Unit defines a *problem state*. All paths from the root to other nodes define the ***state space*** of the problem.

Solution states are those problem states *'s'* for which the path from the root to *'s'* defines a tuple in the solution space. The leaf nodes in the combinatorial tree are the solution states.

Answer states are those solution states *'s'* for which the path from the root to *'s'* defines a tuple that is a member of the set of solutions (i.e.,it satisfies the implicit constraints) of the problem.

The tree organization of the solution space is referred to as the ***state space tree.***

A node which has been generated and all of whose children have not yet been generated is called a ***live node.***

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

A ***dead node*** is a generated node, which is not to be expanded further or all of whose children have been generated.

Bounding functions are used to kill live nodes without generating all their children.

Depth first node generation with bounding function is called backtracking. State generation methods in which the *E*-node remains the *E*-node until it is dead lead to ***branch-and-bound method.***

The term branch-and-bound 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.

In branch-and-bound terminology **Breadth first search(BFS)**- like state space search will be called FIFO (First In First Output) search as the list of live nodes is a first -in-first -out list(or queue).

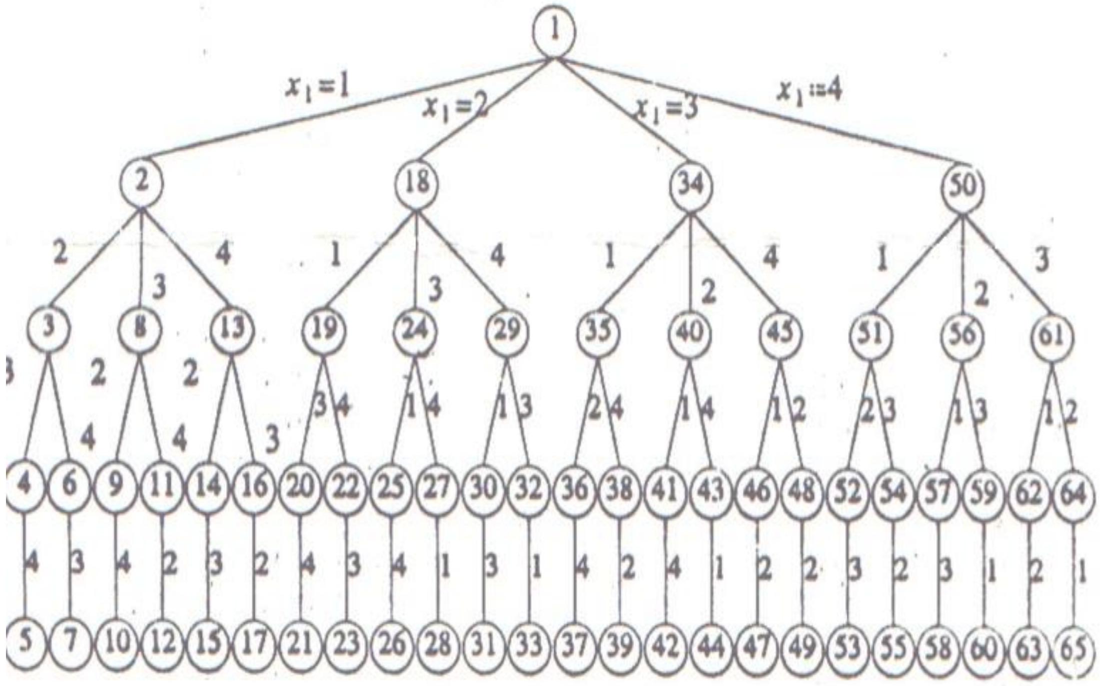
A ***D-search*** (depth search) state space search will be called LIFO (Last In First Out) search, as the list of live nodes is a list-in-first-out list (or stack).

Bounding functions are used to help avoid the generation of sub trees that do not contain an answer node.

The branch-and-bound algorithms search a tree model of the solution space to get the solution. However, this type of algorithms is oriented more toward optimization. An algorithm of this type specifies a real -valued cost function for each of the nodes that appear in the search tree.

Usually, the goal here is to find a configuration for which the cost function is minimized. The branch-and-bound algorithms are rarely simple. They tend to be quite complicated in many cases.

Example 8.1[4-queens] Let us see how a FIFO branch-and-bound algorithm would search the state space tree (figure 7.2) for the 4-queens problem.



Initially, there is only one live node, node1. This represents the case in which no queen has been placed on the chessboard. This node becomes the E-node.

It is expanded and its children, nodes2, 18, 34 and 50 are generated.

These nodes represent a chessboard with queen1 in row 1and columns 1, 2, 3, and 4 respectively.

The only live nodes 2, 18, 34, and 50.If the nodes are generated in this order, then the next E-node are node 2.

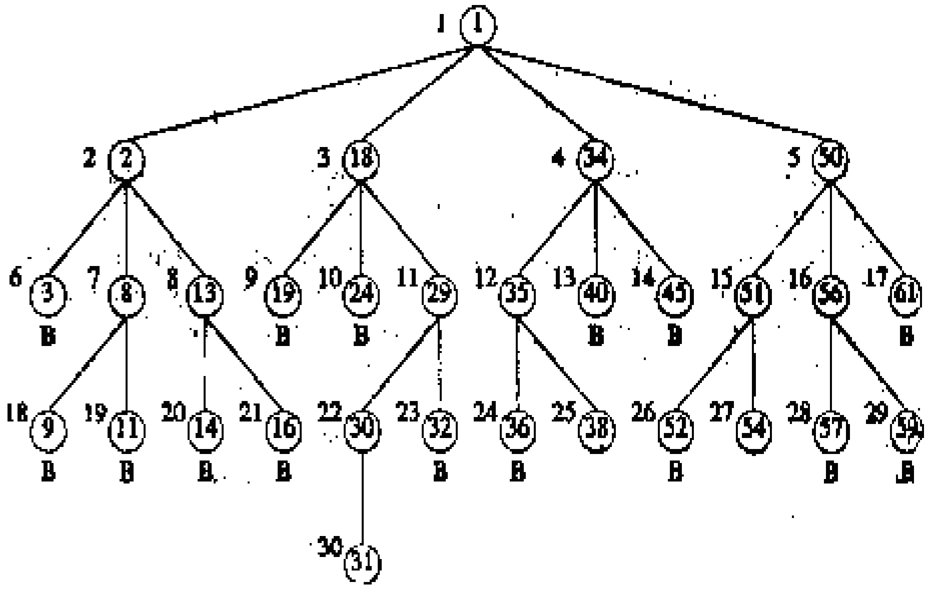
It is expanded and the nodes 3, 8, and 13 are generated. Node 3 is immediately killed using the bounding function. Nodes 8 and 13 are added to the queue of live nodes.

Node 18 becomes the next *E*-node. Nodes 19, 24, and 29 are generated. Nodes 19 and 24 are killed as a result of the bounding functions. Node 29 is added to the queue of live nodes.

Now the *E*-node is node 34.Figure 8.1 shows the portion of the tree of Figure 7.2 that is generated by a FIFO branch-and-bound search. Nodes that are killed as a result of the bounding functions are a "B" under them.

Numbers inside the nodes correspond to the numbers in Figure 7.2.Numbers outside the nodes give the order in which the nodes are generated by FIFO branch-and-bound.

At the time the answer node, node 31, is reached, the only live nodes remaining are nodes 38 and 54.



Least Cost (LC) Search:

In both LIFO and FIFO branch-and-bound the selection rule for the next E-node is rather rigid and in a sense blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

Thus, in Example 8.1, when node 30 is generated, it should have become obvious to the search algorithm that this node will lead to answer node in one move. However, the rigid FIFO rule first requires the expansion of all live nodes generated before node 30 was expanded.

The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{c}$ (.) for live nodes. The next *E*-node is selected on the basis of this ranking function.

If in the 4-queens example we use a ranking function that assigns node 30 a better rank than all other live nodes, then node 30 will become *E*-node, following node 29.The remaining live nodes will never become *E*-nodes as the expansion of node 30 results in the generation of an answer node (node 31).

The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the live node. For any node *x*, this cost could be

(1) The number of nodes on the sub-tree *x* that need to be generated before any answer node is generated or, more simply,

(2) The number of levels the nearest answer node (in the sub-tree *x*) is from *x*

Using cost measure (2), the cost of the root of the tree of Figure 8.1 is 4 (node 31 is four levels from node 1).The costs of nodes 18 and 34,29 and 35,and 30 and 38 are respectively 3, 2, and 1.The costs of all remaining nodes on levels 2, 3, and 4 are respectively greater than 3, 2, and 1.

Using these costs as a basis to select the next E-node, the E-nodes are nodes 1, 18, 29, and 30 (in that order).The only other nodes to get generated are nodes 2, 34, 50, 19, 24, 32, and 31.

The difficulty of using the ideal cost function is that computing the cost of a node usually involves a search of the sub-tree x for an answer node. Hence, by the time the cost of a node is determined, that sub-tree has been searched and there is no need to explore x again. For this reason, search algorithms usually rank nodes only based on an estimate $\hat{g}$ (.) of their cost.

Let $\hat{g}$ (x) be an estimate of the additional effort needed to reach an answer node from x. node x is assigned a rank using a function (.) such that $\hat{c}$ (x) =f (h(x)) + $\hat{g}$ (x), where h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.

A search strategy that uses a cost function $\hat{c}$ (x) =f (h(x)) + $\hat{g}$ (x), to select the next e-node would always choose for its next e-node a live node with least (.).Hence, such a strategy is called an LC-search (least cost search).

Cost function c (.) is defined as, if x is an answer node, then c(x) is the cost (level, computational difficulty, etc.) of reaching x from the root of the state space tree. If x is not an answer node, then c(x) =infinity, providing the sub-tree x contains no answer node; otherwise c(x) is equals the cost of a minimum cost answer node in the sub-tree x.

It should be easy to see that $\hat{c}$ (.) with f (h(x)) =h(x) is an approximation to c (.). From now on (x) is referred to as the cost of x.

Bounding:

A branch -and-bound searches the state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node.

We assume that each answer node x has a cost c(x) associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC.

A cost function $\hat{c}$ (.) such that $\hat{c}$ (x) <=c(x) is used to provide lower bounds on solutions obtainable from any node x. If upper is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $\hat{c}$ (x)>upper may be killed as all answer nodes reachable from x have cost c(x)>= $\hat{c}$ (x)>upper. The starting value for upper can be set to infinity.

Clearly, so long as the initial value for upper is no less than the cost of a minimum-cost answer node, the above rule to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node .Each time a new answer is found ,the value of upper can be updated.

As an example optimization problem, consider the problem of job scheduling with deadlines. We generalize this problem to allow jobs with different processing times. We are given n jobs and one processor. Each job i has associated with it a three tuple (p_i d_i t_i).job i requires t_i units of processing time .if its processing is not completed by the deadline d_i , and then a penalty p_i is incurred.

The objective is to select a subset j of the n jobs such that all jobs in j can be completed by their deadlines. Hence, a penalty can be incurred only on those jobs not in j. The subset j should be such that the penalty incurred is minimum among all possible subsets j. such a j is optimal.

Consider the following instances: n=4,(p_1 , d_1 , t_1)=(5,1,1),(p_2 , d_2 , t_2)=(10,3,2),(p_3 , d_3 , t_3)=(6,2,1),and(p_4 , d_4 , t_4)=(3,1,1).The solution space for this instances consists of all possible subsets of the job index set{1,2,3,4}. The solution space can be organized into a tree by means of either of the two formulations used for the sum of subsets problem.

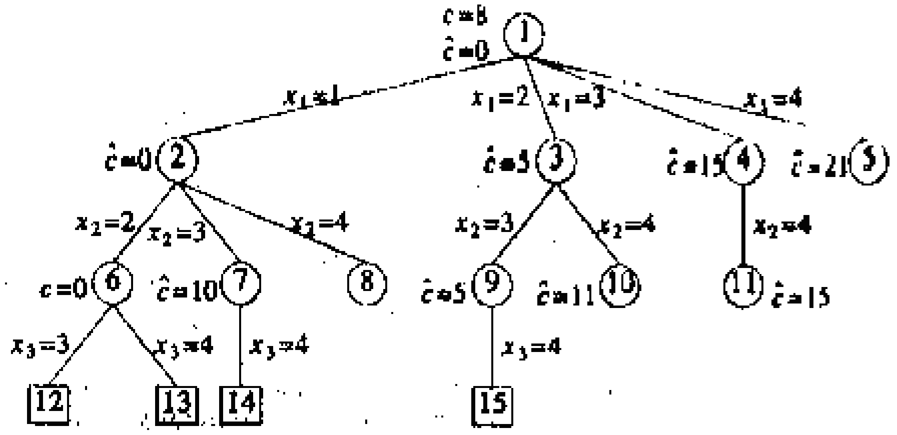


Figure 8.6 State space tree corresponding to variable tuple size formulation

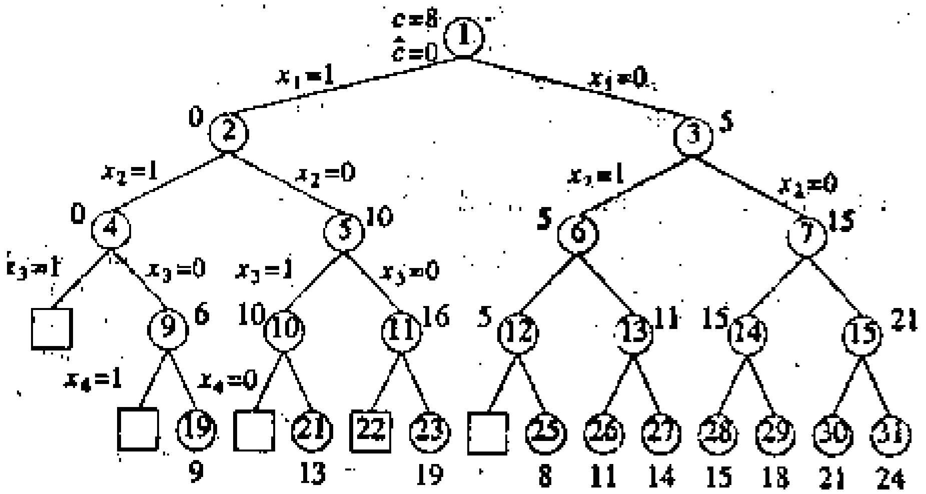


Figure 8.7

Figure 8.6 corresponds to the variable tuple size formulations while figure 8.7 corresponds to the fixed tuple size formulation. In both figures square nodes represent infeasible subsets. In figure 8.6 all non-square nodes are answer nodes. Node 9 represents an optimal solution and is the only minimum-cost answer node .For this node j= {2,3} and the penalty (cost) is 8. In figure 8.7 only non-square leaf nodes are answer nodes. Node 25 represents the optimal solution and is also a minimum-cost answer node. This node corresponds to j={2,3} and a penalty of 8. The costs of the answer nodes of figure 8.7 are given below the nodes.

TRAVELLING SALESMAN PROBLEM

INTRODUCTION:

It is algorithmic procedures similar to backtracking in which a new branch is chosen and is there (bound there) until new branch is choosing for advancing.

This technique is implemented in the traveling salesman problem [TSP] which are asymmetric (Cij <>Cij) where this technique is an effective procedure.

S**TEPS INVOLVED IN THIS PROCEDURE ARE AS FOLLOWS:**

STEP 0: Generate cost matrix C [for the given graph g]

STEP 1: [*ROW REDUCTION]*
For all rows do step 2

STEP: Find least cost in a row and negate it with rest of the elements.

STEP 3: *[COLUMN REDUCTION]*
Use cost matrix- Row reduced one for all columns do STEP 4.

STEP 4: Find least cost in a column and negate it with rest of the elements.

STEP 5: Preserve cost matrix C [which row reduced first and then column reduced] for the i th time.

STEP 6: Enlist all edges (i, j) having cost = 0.

STEP 7: Calculate effective cost of the edges. $\sum$ (i, j)=least cost in the i th row excluding (i, j) + least cost in the j th column excluding (i, j).

STEP 8: Compare all effective cost and pick up the largest l. If two or more have same cost then arbitrarily choose any one among them.

STEP 9: Delete (i, j) means delete ith row and jth column change (j, i) value to infinity. (Used to avoid infinite loop formation) If (i,j) not present, leave it.

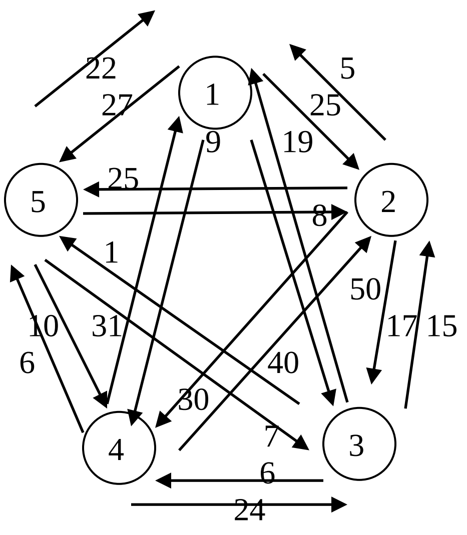
STEP 10: Repeat step 1 to step 9 until the resultant cost matrix having order of 2*2 and reduce it. (Both R.R and C.C)

STEP 11: Use preserved cost matrix Cn, Cn-1… C1

Choose an edge [i, j] having value =0, at the first time for a preserved matrix and leave that matrix.

STEP 12: Use result obtained in Step 11 to generate a complete tour.

EXAMPLE: *Given graph G*



MATRIX:

	1	2	3	4	5
1	α	25	40	31	27
2	5	α	17	30	25
3	19	15	α	6	1
4	9	50	24	α	6
5	22	8	7	10	α

PHASE I

STEP 1: Row Reduction C

C1 [ROW REDUCTION:

	1	2	3	4	5
1	α	0	15	6	2
2	0	α	12	25	20
3	18	14	α	5	0
4	3	44	18	α	0
5	15	1	0	3	α

STEP 3: C1 [Column Reduction]

	1	2	3	4	5
1	α	0	15	3	2
2	0	α	12	25	20
3	18	14	α	2	0
4	3	44	18	α	0
5	15	1	0	3	α

STEP 5:

Preserve the above in C1,

	1	2	3	4	5
1	α	0	15	3	2
2	0	α	12	22	20
3	18	14	α	2	0
4	3	44	18	α	0
5	15	1	0	3	α

STEP 6:

L= $\{(1,2), (2,1), (3,5), (4,5), (5,3), (5,4)\}$

STEP 7:

Calculation of effective cost [E.C]

(1,2) = 2+1 =3

(2,1) = 12+3 = 15

(3,5) = 2+0 =2

(4,5) = 3+0 = 3

(5,3) = 0+12 = 12

(5,4) = 0+2 = 2

STEP 8:

L having edge (2,1) is the largest.

STEP 9: Delete (2,1) from C1 and make change in it as (1,2) $\rightarrow$ α if exists.

Now Cost Matrix =

	2	3	4	5
1	α	15	3	2
3	14	α	2	0
4	44	18	α	0
5	1	0	0	α

STEP 10: The Cost matrix $\neq$ 2 x 2.

Therefore, go to step 1.

<u>PHASE II:</u>

STEP1<u>:</u> C2(R, R)

	2	3	4	5
1	α	13	1	0
3	14	α	2	0
4	44	18	α	0
5	1	0	0	α

<u>STEP 3:</u> C2 (C, R)

2 3 4 5

$$\begin{array}{c|c|c|c|c|}
\hline
1 & \alpha & 13 & 1 & 0 \\ \hline
3 & 13 & \alpha & 2 & 0 \\ \hline
4 & 43 & 18 & \alpha & 0 \\ \hline
5 & 0 & 0 & 0 & \alpha \\ \hline
\end{array}$$

STEP 5: Preserve the above in C2

C2 =

$$\begin{array}{c|c|c|c|c|}
 & 2 & 3 & 4 & 5 \\ \hline
1 & \alpha & 13 & 1 & 0 \\ \hline
3 & 13 & \alpha & 2 & 0 \\ \hline
4 & 43 & 18 & \alpha & 0 \\ \hline
5 & 0 & 0 & 0 & \alpha \\ \hline
\end{array}$$

STEP 6:

L= {(1,5), (3,5), (4,5), (5,2), (5,3), (5,4)}

STEP 7: calculation of E.C.

(1,5) = 1+0 =1
(3,5) = 2+0 =2
(4,5) = 18+0 =18
(5,2) = 0+13 =13
(5,3) = 0+13 =13
(5,4) = 0+1 =1

STEP 8: L having an edge (4,5) is the largest.

STEP 9: Delete (4,5) from C2 and make change in it as (5,4) = α if exists.

Now, cost matrix

$$\begin{array}{|c|c|c|}
2 & 3 & 4 \\ \hline
\alpha & 13 & 1 \\ \hline
\end{array}$$

$$\begin{array}{c|c|c|c|}
\cline{2-4}
1 & 13 & \alpha & 2 \\
\cline{2-4}
3 & 0 & 0 & \alpha \\
\cline{2-4}
5 & & & \\
\end{array}$$

STEP 10: THE cost matrix $\neq$ 2x2 hence go to step 1

PHASE III:

STEP 1: C3 (R, R)

$$\begin{array}{c|c|c|c|}
 & 2 & 3 & 4 \\
\cline{2-4}
1 & \alpha & 12 & 0 \\
\cline{2-4}
3 & 11 & \alpha & 0 \\
\cline{2-4}
5 & 0 & 0 & \alpha \\
\cline{2-4}
\end{array}$$

STEP 3: C3 (C, R)

$$\begin{array}{c|c|c|c|}
 & 2 & 3 & 4 \\
\cline{2-4}
1 & \alpha & 12 & 0 \\
\cline{2-4}
3 & 11 & \alpha & 0 \\
\cline{2-4}
5 & 0 & 0 & \alpha \\
\cline{2-4}
\end{array}$$

STEP 5: preserve the above in C2

STEP 6: L={(1,4), (3,4), (5,2), (5,3)}

STEP 7: calculation of E.C

(1,4)=12+0=12
(3,4)=11+0=11
(5,2)=0+11=11
(5,3)=0+12=12

STEP 8: Here we are having two edges (1,4) and (5,3) with cost = 12. Hence arbitrarily choose (1,4)

STEP 9: Delete (i,j) $\rightarrow$(1,4) and make change in it (4,1) = α if exists.

Now cost matrix is

2 3

$$\begin{array}{c|cc|}
2 & 11 & \alpha \\
3 & 0 & 0
\end{array}$$

<u>STEP 10</u>: We have got 2x2 matrix

C4 (RR)=

$$\begin{array}{c|cc|}
 & 2 & 3 \\
3 & 0 & \alpha \\
5 & 0 & 0
\end{array}$$

C4 (C, R) =

$$\begin{array}{c|cc|}
 & 2 & 3 \\
3 & 0 & \alpha \\
5 & 0 & 0
\end{array}$$

Therefore,C4 =

$$\begin{array}{c|cc|}
 & 2 & 3 \\
3 & 0 & \alpha \\
5 & 0 & 0
\end{array}$$

<u>STEP 11:</u> LIST C1, C2, C3 AND C4

C4

$$\begin{array}{c|cc|}
 & 2 & 3 \\
3 & 0 & \alpha \\
5 & 0 & 0
\end{array}$$

C3

$$\begin{array}{c|ccc|}
 & 2 & 3 & 4 \\
1 & \alpha & 12 & 0 \\
3 & 11 & \alpha & 0 \\
5 & 0 & 0 & \alpha
\end{array}$$

C2 =

$$\begin{array}{c|cccc|}
1 & \alpha & 13 & 1 & 0 \\
3 & 13 & \alpha & 2 & 0 \\
4 & 43 & 18 & \alpha & 0 \\
5 & 0 & 0 & 0 & \alpha
\end{array}$$

C1 =

	1	2	3	4	5
1	α	0	15	3	2
2	0	α	12	22	20
3	18	14	α	2	0
4	3	44	18	α	0
5	15	1	0	0	α

<u>STEP 12:</u>

i) Use C4 =

	2	3
3	0	α
5	0	0

Pick up an edge (I, j) =0 having least index

Here (3,2) =0

Hence, T← (3,2)

Use C3 =

	2	3	4
1	α	12	0
3	11	α	0
5	0	0	α

Pick up an edge (i, j) =0 having least index

Here (1,4) =0

Hence, T←(3,2), (1,4)

Use C2=

	2	3	4	5
1	α	13	1	0
3	13	α	2	0
4	43	18	α	0
5	0	0	0	α

Pick up an edge (i, j) with least cost index.

Here (1,5) $\rightarrow$ not possible because already chosen index i (i=j)
(3,5) $\rightarrow$ not possible as already chosen index.
(4,5) $\rightarrow$ 0

Hence, T$\leftarrow$ (3,2), (1,4), (4,5)

Use C1 =

$$
\begin{array}{c|c|c|c|c|c|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & \alpha & 0 & 15 & 3 & 2 \\
\hline
2 & 0 & \alpha & 12 & 22 & 20 \\
\hline
3 & 18 & 14 & \alpha & 2 & 0 \\
\hline
4 & 3 & 44 & 18 & \alpha & 0 \\
\hline
5 & 15 & 1 & 0 & 0 & \alpha \\
\hline
\end{array}
$$

Pick up an edge (i, j) with least index

(1,2) $\rightarrow$ Not possible
(2,1) $\rightarrow$ Choose it

HENCE T$\leftarrow$ (3,2), (1,4), (4,5), (2,1)

<u>SOLUTION:</u>

From the above list
3—2—1—4—5

This result now, we have to return to the same city where we started (Here 3).

<u>Final result</u>:
3—2—1—4—5—3

<u>Cost is 15+15+31+6+7=64</u>