Visvesvaraya Technological University (VTU)

Subject Code:21CS42

Subject : DESIGN AND ANALYSIS OF ALGOROTHM

Created By:

Hanumanthu

Dep. Of CSE

Visit Our Official Website It is Available @ any time

http://searchcreators.org/

MODULE-05

Backtracking

In the backtracking method

1. The desired solution is expressible as an n tuple (x1,x2,...xn) where xi is chosen from some finite set Si.
2. The solution maximizes or minimizes or satisfies a criterion function C (x1, x2,...xn).

- The problem can be categorized into three categories.
- For Stance - For a problem P let C be the set of constraints for P. Let D be the set containing all solutions satisfying C then
- Finding whether there is any feasible solution ? - Is the decision problem. What is the best solution ? - Is the optimization problem.
- Listing of all the feasible solution - Is the enumeration problem.
- The basic idea of backtracking is to build up a vector, one component at a time and to test whether the vector being formed has any chance of success.
- The major advantage of backtracking algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be ignored.
- Backtracking algorithm determines the solution by systematically searching the solution space (i.e. set of all feasible solutions) for the given problem.
- Backtracking is a depth first search with some bounding function.

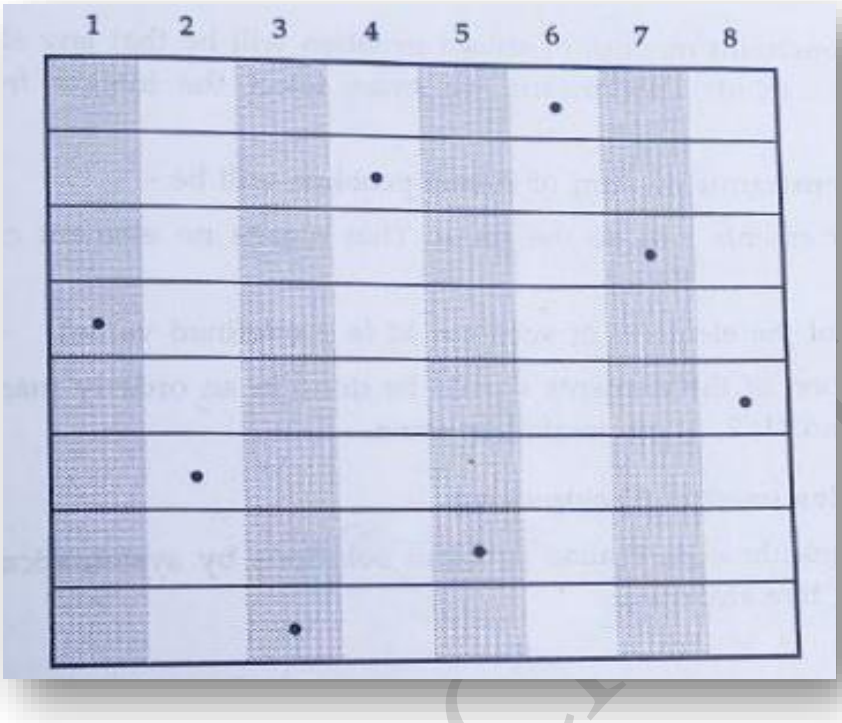
- Backtracking algorithm solves the problem using two types of constraints - Explicit constraint and Implicit constraints.
- Definition : Explicit constraints are the rules that restrict each element x, has to be chosen from given set only. Explicit constraints depends on particular instance I of the problem. All the tuples from solution set must satisfy the explicit constraints.
- Definition : Implicit constraints are the rules that decide which tuples in the solution space of I satisfy the criterion function. Thus the implicit constraints represent by which xi, in the solution set must be related with each other.

For example

- Example 1 : 8-Queen's problem –

  The 8-queen's problem can be stated The solution to 8-queens problem cart be obtained using backtracking method. As follows. Consider a chessboard of order 8x 8. The problem is to place 8 queens on this board such that no two queens can attack each other. That means no two queens can be placed on the same row, column or diagonal.

The solution can be given as below -



This 8 Queen's problem is solved by applying implicit and explicit constraints.

The explicit constraints show that the solution space Si must be (1, 2, 3, 4, 5, 6, 7, 8} with 1<_ i <_8. Hence solution space consists of 8_8 8-tuples.

The implicit constraint will be - 1) No two xi will be same. That means all the queens must lie in different columns.

2) No two queens can be on the same row, column or diagonal.

Hence the above solution can be represented as an 8-tuple {4, 6, 8, 2, 7, 1, 3, 5}.

Algorithm

Recursive Algorithm

```
Algorithm Backtrack()
//This is recursive backtracking algorithm
//a[k] is a solution vector. On entering (k–1) remaining next
//values can be computed.
//T(a1,a2,...ak) be the set of all values for a(i+1), such that
//(a1,a2,...ai+1) is a path to problem state.
// B(i+1) is a bounding function such that if B(i+1) (a1,a2,...ai+1) is false
//for a path (a1,a2,...ai+1) from root node to a problem state then
//path can not be extended to reach an answer node
{
for ( each ak that belongs to T((a1,a2,...ak-1) do
{
 if(Bk(a1,a2,...ak) = true) then // feasible sequence
 {
    if ((a1,a2,...ak) is a path to answer node then
      print(a[1],a[2],...a[k]);
    if (k<n) then
      Backtrack(k+1); //find the next set.
  }
 }
}
```

Iterative Algorithm

The non recursive backtracking algorithm can be given as –

```
Algorithm Non_Rec_Back(n)
// This is a non recursive version of backtracking
//a[1,...,n] is a solution vector and each a1,a2,...ak will be used to print solution.
{
 k:=1;
 while(k ≠ 0) do
```

```
{
    if(any a[k] that belongs to T(a[1],a[2],...a[k-1])remains untried)
    AND (B_k (a[1],a[2]...,a[k] is true) then

  {
   if((a[1],a[2]...a[k]) is a path to answer node) then
    write(a[1],a[2],..a[k]); // solution printed
   //consider next element of the set
   k:=k+1;
   else
    k : =k-1;  //backtrack to most recent value
  }
 }
}
```

N-Queens Problem

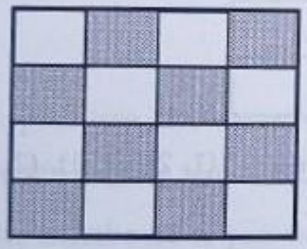
The n-queen's problem can be stated as follows.

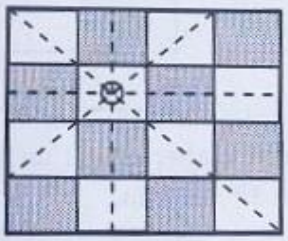
Consider a n x it chessboard on which we have to place it queens so that no two

queens attack each other by being in the same row or in the same column or on the same diagonal.

For example

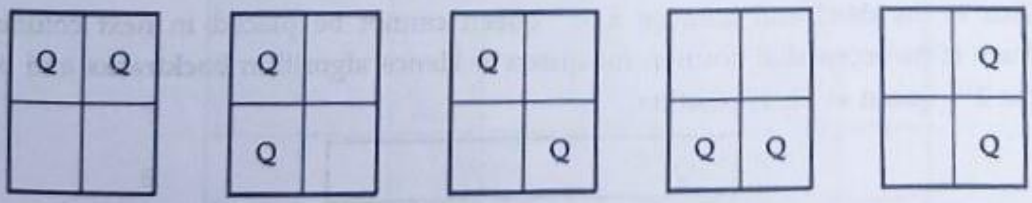
Consider 4 x 4 board





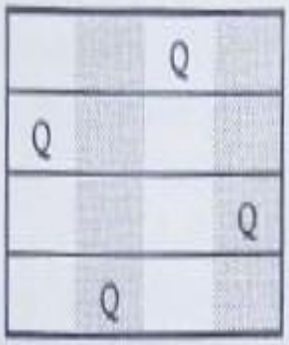
The next queen - if is placed on the paths marked by dotted lines then they can attack each other

- 2-Queen's problem is not solvable - Because 2-queens can be placed on 2 × 2 chessboard as



Illegal Illegal Illegal Illegal Illegal

- But 4-queen's problem is solvable.

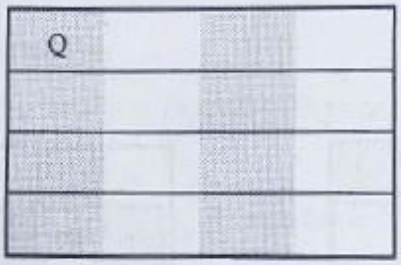


⇐ Note that no two queens can attack each other.

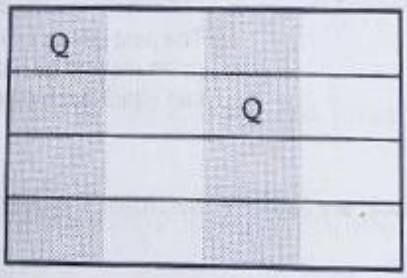
How to solve n-Queen's Problem?

Let us take 4-queens and 4×4 chessboard.

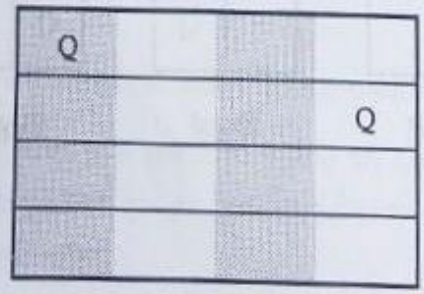
- Now we start with empty chessboard.
- Place queen 1 in the first possible position of its row i.e. on 1^{st} row and 1^{st} column.



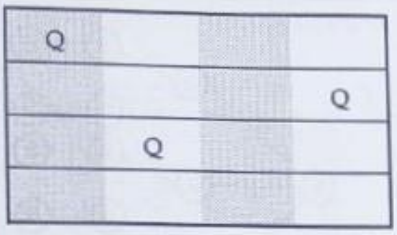
- Then place queen 2 after trying unsuccessful place - 1(1, 2), (2, 1), (2, 2) at (2, 3) i.e. 2^{nd} row and 3^{rd} column.



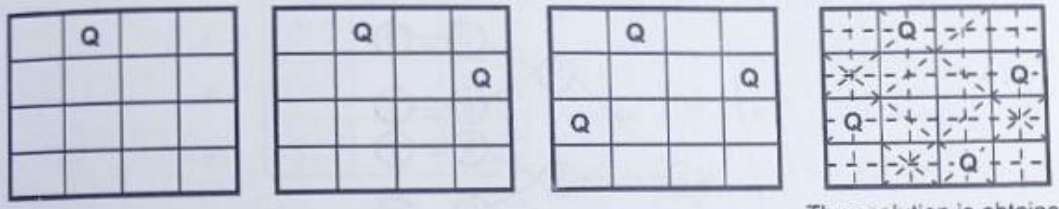
- This is the dead end because a 3^{rd} queen cannot be placed in next column, as there is no acceptable position for queen 3. Hence algorithm **backtracks** and places the 2^{nd} queen at (2, 4) position.



- The place 3^{rd} queen at (3, 2) but it is again another dead end as next queen (4^{th} queen) cannot be placed at permissible position.



- Hence we need to backtrack all the way upto queen 1 and move it to (1, 2).
- Place queen 1 at (1, 2), queen 2 at (2, 4), queen 3 at (3, 1) and queen 4 at (4, 3).

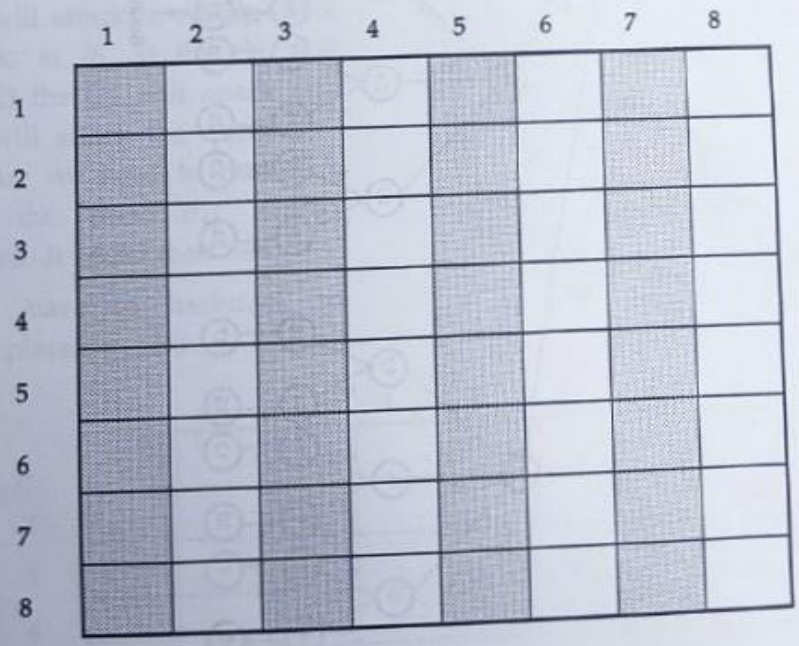


Thus solution is obtained.
(2, 4, 1, 3) in rowwise manner.

The state space tree of 4-queen's problem is shown in Fig. 5.2.1 (See on next page)

Now we will consider how to place **8-Queen's** on the chessboard.

Initially the chessboard is empty.



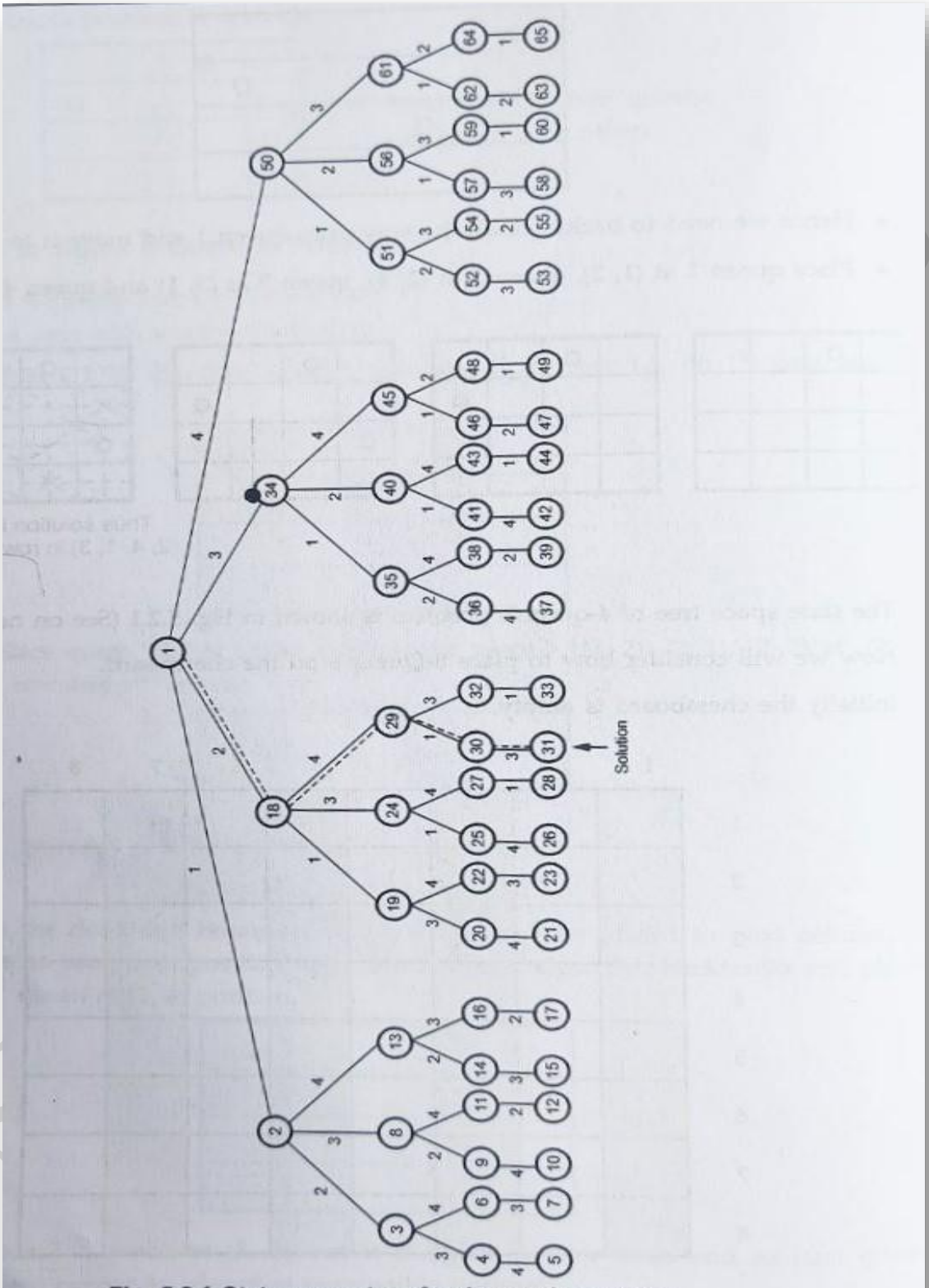
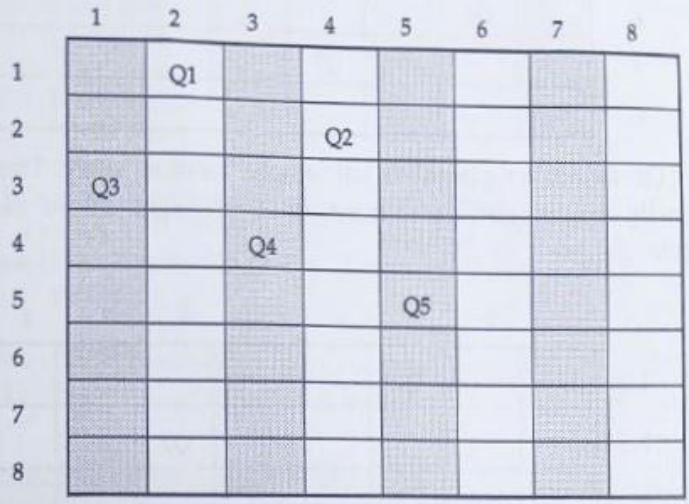
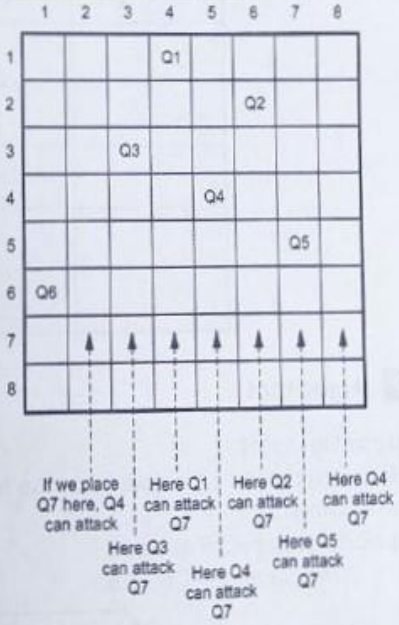


Fig. 5.2.1 State space tree for 4-queen's problem

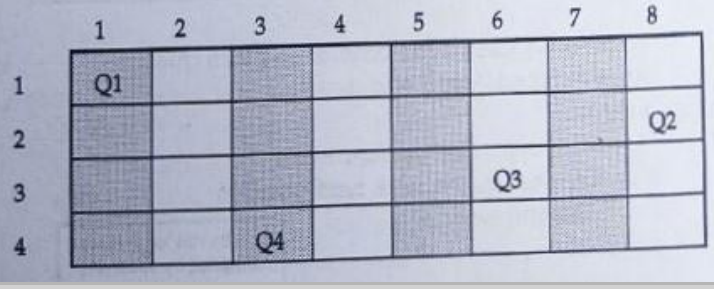
Now we will start placing the queens on the chessboard.

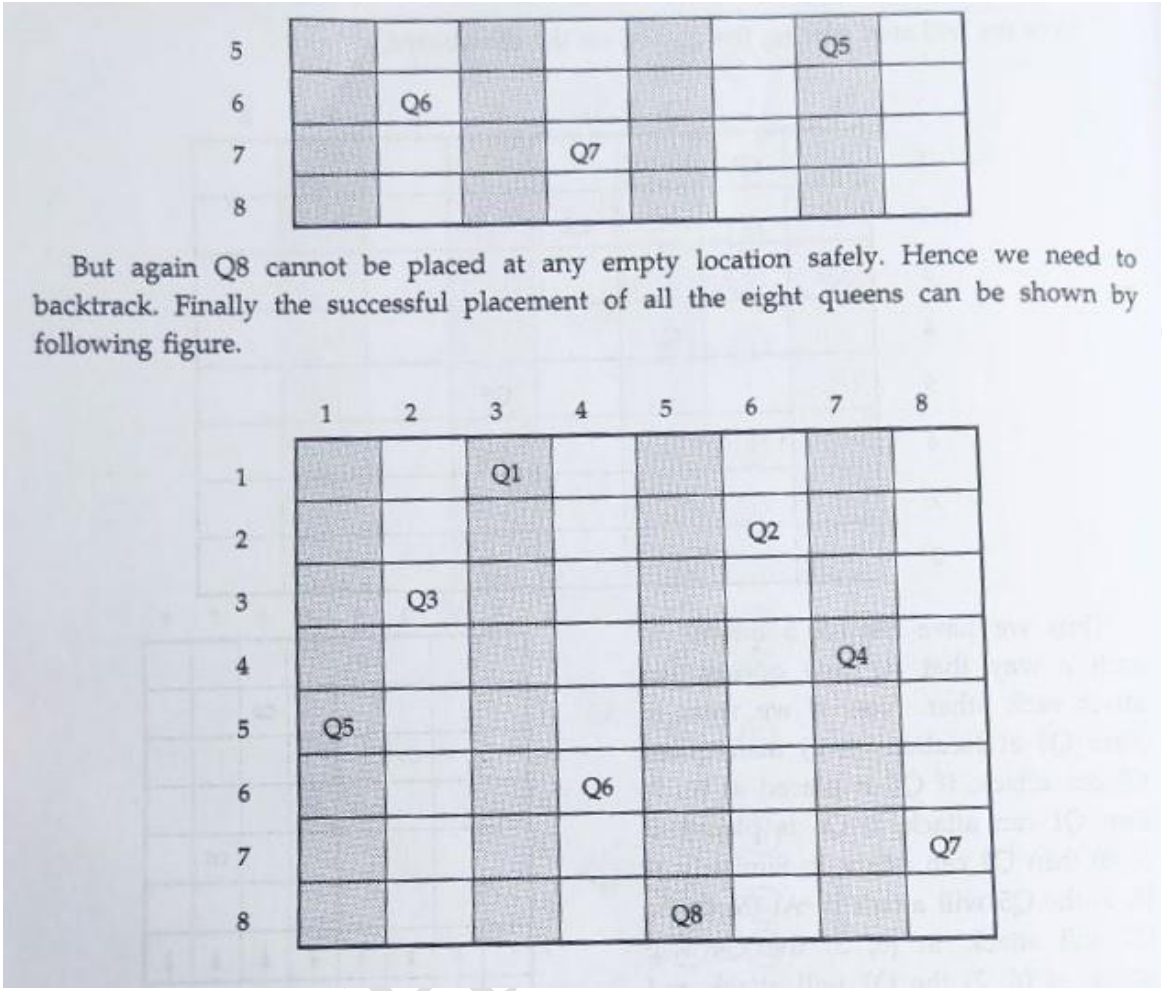


Thus we have placed 5 queens in such a way that no two queens can attack each other. Now if we want to place Q6 at location (6, 6) then queen Q5 can attack, if Q6 is placed at (6, 7) then Q1 can attack, if Q6 is placed at (6, 8) then Q2 can attack it. Similarly at (6, 5) the Q5 will attack it. At (6, 4) the Q2 will attack, at (6, 3) the Q4 will attack, at (6, 2) the Q1 will attack and at (6, 1) Q3 will attack the queen Q6. This shows that we need to backtrack and change the previously placed queens positions. It could then be -



Hence we have to backtrack to adjust already placed queens.





But again Q8 cannot be placed at any empty location safely. Hence we need to backtrack. Finally the successful placement of all the eight queens can be shown by following figure.

Algorithm

```
Algorithm Queen(n)
//Problem description : This algorithm is for implementing n
//queen's problem
//Input : total number of queen's n.
        for column ←1 to n do
        {
            if(place(row,column))then
            {
             board[row]column//no conflict so place queen
            if(row=n)then//dead end
        print_board(n)
            //printing the board configuration
            else//try next queen with next position
                Queen(row+1,n)
            }
        }
```

This function checks if two queens are on the same diagonal or not.

Row by row each queen is placed by satisfying constraints.

```
Algorithm place(row,column)
//Problem Description : This algorithm is for placing the
//queen at appropriate position
//Input : row and column of the chessboard
//output : returns 0 for the conflicting row and column
//position and 1 for no conflict.
        for i ← 1 to row-1 do
        { //checking for column and diagonal conflicts
          if(board[i] = column)then
            return 0
        else if(abs(board[i]- column) = abs(i - row))then
            return 0
        }
        //no conflicts hence Queen can be placed
    return 1
```

Same column by 2 queen's

This formula gives that 2 queens are on same diagonal

Sum of Subsets Problem

Problem Statement

Let, S = ISI,…. S,,) be a set of n positive integers, then we have to find a subset whose stun is equal to given positive integer d.

It is always convenient to sort the set's elements in ascending order.

That is, S1 5 S2 5…5 SI,

Let us first write a general algorithm for sum of subset problem.

Algorithm

Let, S be a set of elements and d is the expected sum of subsets. Then -Step 1 : Start with an empty set.

Step 2 : Add to the subset, the next element from the list.

Step 3 : If the subset is having sum d then stop with that subseti6 solution.

Step 4 : If the subset 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.

Step 5 : If the subset is feasible then repeat step 2.

Step 6 : If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

This problem can be well understood with some example.

Example:-

Consider a set S = (5, 10, 12, 13, 15, 181 and d = 30. Solve it for obtaining sum of subset.

Solution.

Initially subset = { }	Sum = 0	–
5	5	Then add next element.
5, 10	15 ∵ 15 < 30	Add next element.
5, 10, 12	27 ∵ 27 < 30	Add next element.
5, 10, 12, 13	40	Sum exceeds d = 30 hence backtrack.
5, 10, 12, 15	42	Sum exceeds d = 30 ∴ Backtrack

5, 10, 12, 18	45	Sum exceeds d ∴ not feasible. Hence backtrack.
5, 10		
5, 10, 13	28	
5, 10, 13, 15	33	Not feasible ∴ Backtrack.
5, 10		
5, 10, 15	30	Solution obtained as sum = 30 = d

∴ The state space tree can be drawn as follows.

{5, 10, 12, 13, 15, 18}

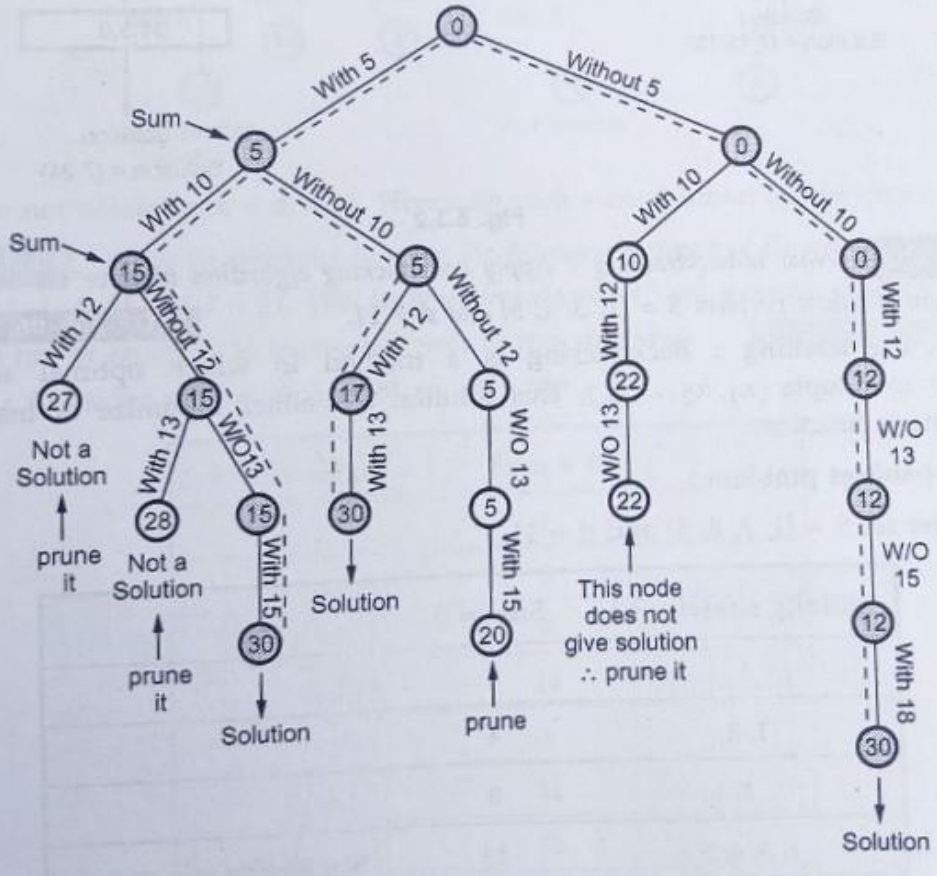
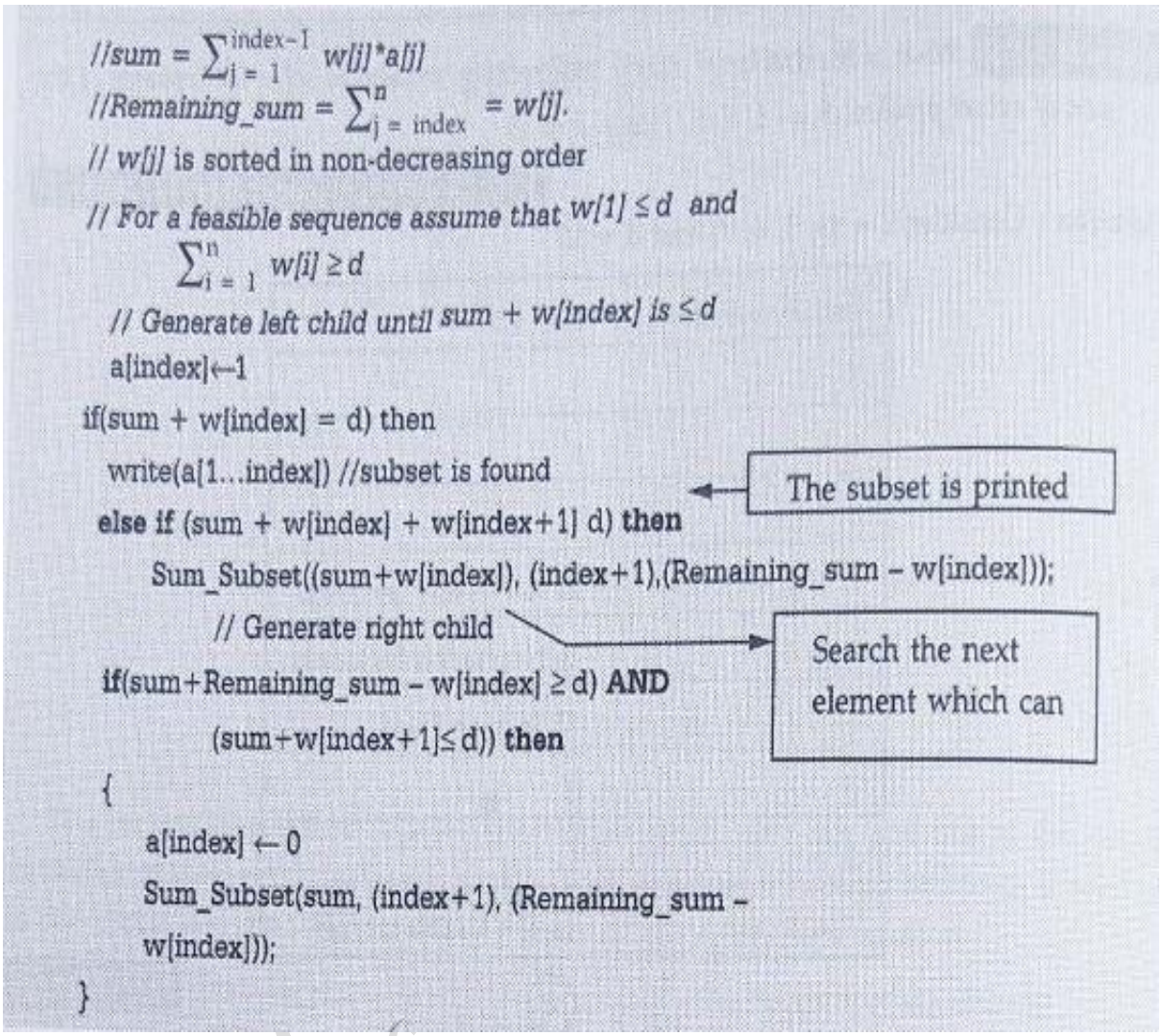


Fig. 5.3.1 State space tree for sum of subset

Algorithm

```
Algorithm Sum_Subset(sum, index, Remaining_sum)
//sum is a variable that stores the sum of all the
//selected elements
//index denctes the index of chosen element from the given
//Remaining_sum is initially sum of all the elements.
//selection of some element from the set subtracts the
//chosen value
//from Remaining_sum each time.
//w[1...n] represents the set containing n elements
//a[j] represents the subset where 1 ≤ j ≤ index
```



```
//sum = Σ_{j=1}^{index-1} w[j]*a[j]
//Remaining_sum = Σ_{j=index}^{n} = w[j].
// w[j] is sorted in non-decreasing order
// For a feasible sequence assume that w[1] ≤ d and
        Σ_{i=1}^{n} w[i] ≥ d
  // Generate left child until sum + w[index] is ≤ d
  a[index]←1
if(sum + w[index] = d) then
  write(a[1...index]) //subset is found
 else if (sum + w[index] + w[index+1] d) then
      Sum_Subset((sum+w[index]), (index+1),(Remaining_sum – w[index]));
            // Generate right child
  if(sum+Remaining_sum – w[index] ≥ d) AND
            (sum+w[index+1]≤ d)) then
  {
      a[index] ← 0
      Sum_Subset(sum, (index+1), (Remaining_sum –
      w[index]));
}
```

The subset is printed

Search the next element which can

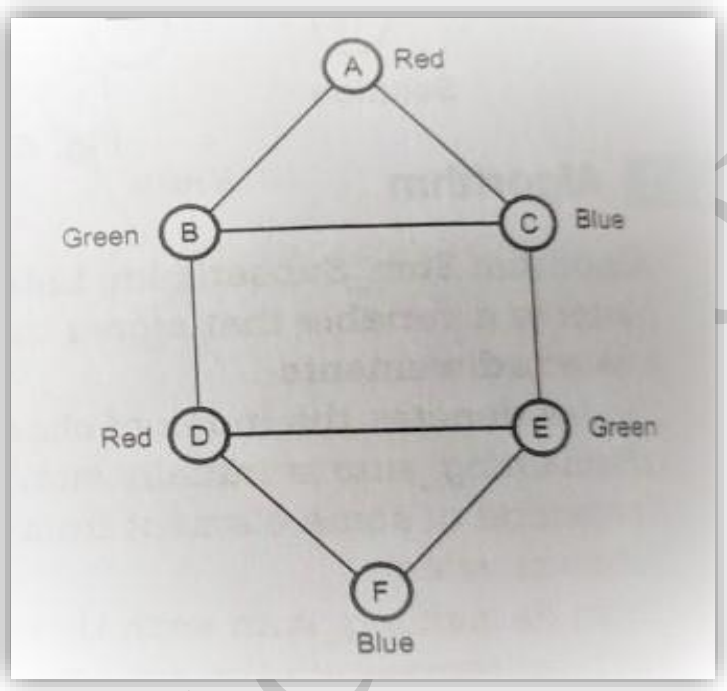
Graph Coloring

Graph coloring is a problem of coloring each vertex in graph in such a way that no two adjacent vertices have same color and yet m-colors are used.

This problem is also called m-coloring problem. If the degree of given graph is d then we can color it with d + 1 colors.

The least number of colors needed to color the graph is called its chromatic number.

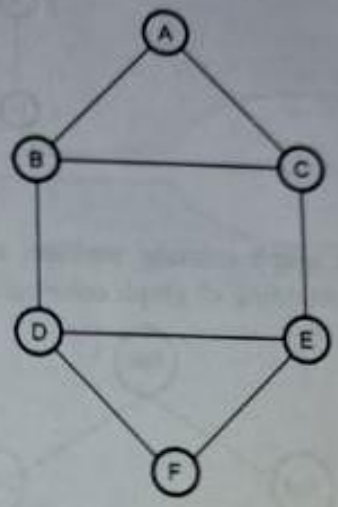
For example : Consider a graph



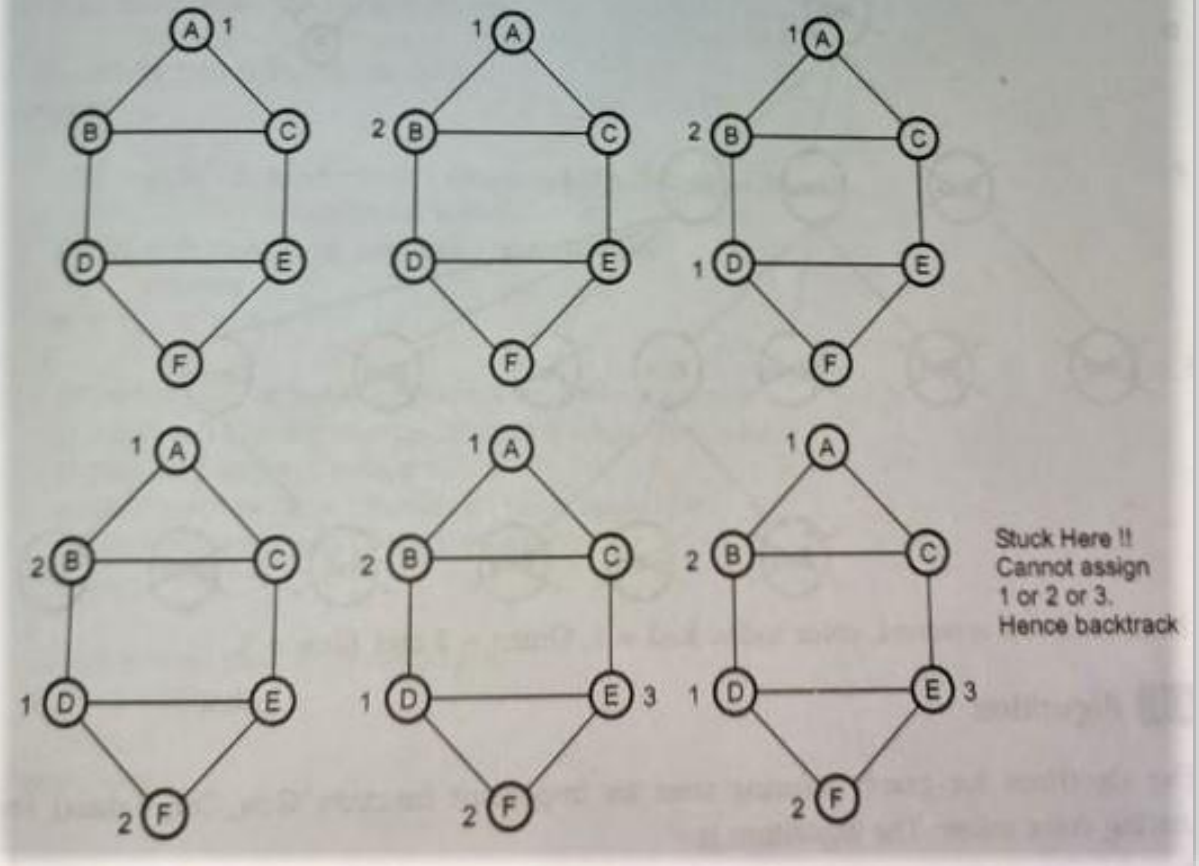
As given in Fig. 5.4.1 we require three colors to color the graph. Hence the chromatic number of given graph is 3. We can use backtracking technique to solve the graph coloring problem as follows -

Step 1 :

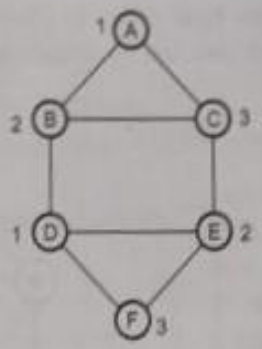
A Graph G consists of vertices from A to F. There are three colors used Red, Green and Blue. We will number them out. That means 1 indicates Red, 2 indicates Green and 3 indicates Blue color.



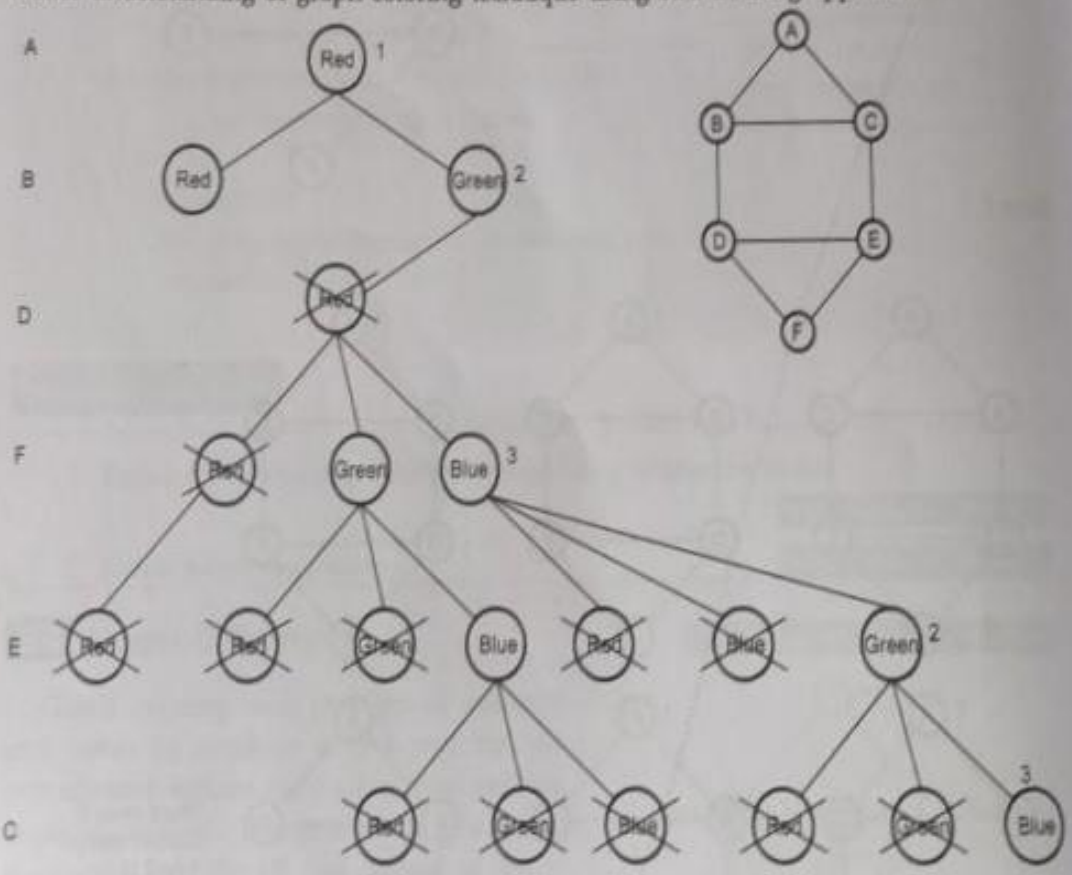
Step 2 :



Step 3 :



Thus the graph coloring problem is solved. The state-space tree can be drawn for better understanding of graph coloring technique using backtracking approach -



Here we have assumed, color index Red = 1, Green = 2 and Blue = 3.

Algorithm

The algorithm for graph coloring uses an important function **Gen_Col_Value()** for generating color index. The algorithm is -

```
Algorithm Gr_color(k)

//The graph G[1 : n,1 : n] is given by adjacency matrix.
//Each vertex is picked up one by one for coloring
//x[k] indicates the legal color assigned to the vertex
{
repeat
{   // produces possible colors assigned

     Gen_Col_Value(k);

if(x[k] = 0) then
   return; //Assignment of new color is not possible.
if(k=n) then // all vertices of the graph are colored
  write(x[1:n]); //print color index
else
 Gr_color(k+1) // choose next vertex
}until(false);
}
```

Takes O(nm) time

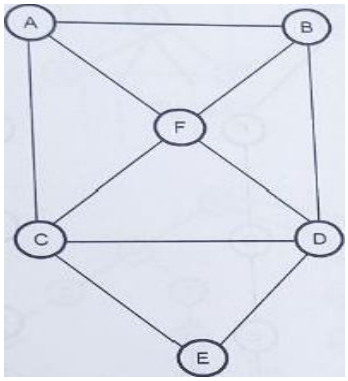
The algorithm used assigning the color is given as below

```
Algorithm Gen_Col_Value(k)
 //x[k] indicates the legal color assigned to the vertex
 // If no color exists then x[k] = 0
{
// repeatedly find different colors
 repeat
  {
    x[k] ← (x[k]+1) mod (m+1) ; //next color index when it is
                          //highest index
    if(x[k] = 0) then // no new color is remaining
           return;
   for (j ← 1 to n) do
   {
    // Taking care of having different colors for adjacent
    // vertices by using two conditions i) edge should be
    // present between two vertices
   // ii) adjacent vertices should not have same color
    if(G[k,j]!=0) AND (x[k]→x[j])) then
     break;
   }
   //if there is no new color remaining
   if ( j= n+1) then
      return;
  }until(false);
}
```

Hamiltonian Cycles

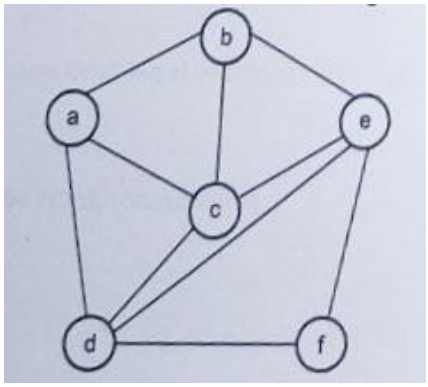
Problem -

Given an undirected connected graph and two nodes x and y then find a path from x to y visiting each node in the graph exactly once.



For example: Consider the graph G -

Then the Hamiltonian cycle is A - B - D - E - C - F - A. This problem can be solved using backtracking approach. The state space tree is generated in order to find all the Hamiltonian cycles in the graph. Only distinct cycles are output of this algorithm. The Hamiltonian cycle can be identified as follows - (See Fig. 5.5.2 on next page.) In Fig. 5.5.2 clearly the backtrack approach is adopted. For instance A - B - D - F – C - E; here we get stuck. For returning to A we have to revisit at least one vertex. Hence we backtracked and from D node another path is chosen A-B-D-E-C-F-A which is Hamiltonian cycle.



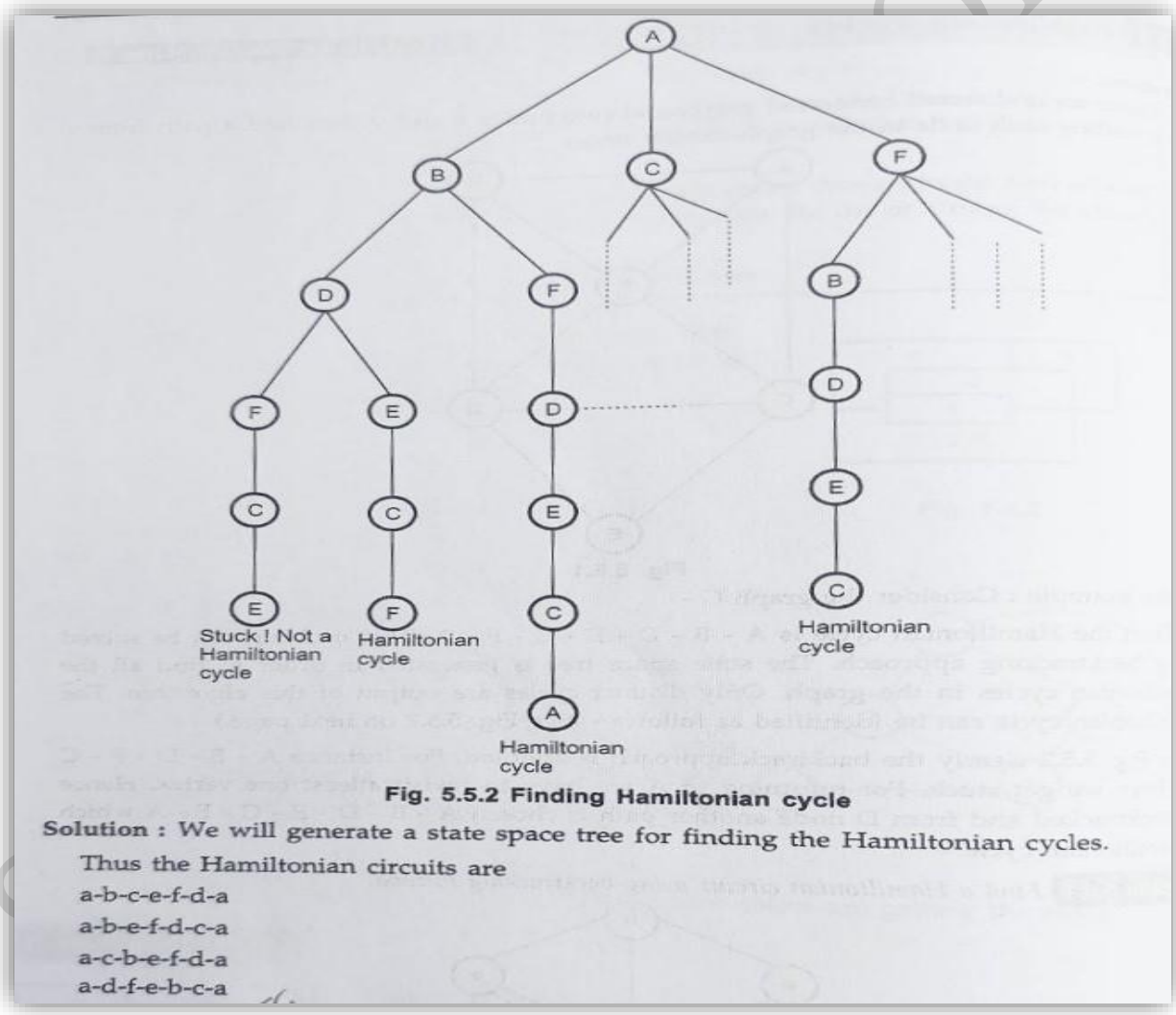


Fig. 5.5.2 Finding Hamiltonian cycle

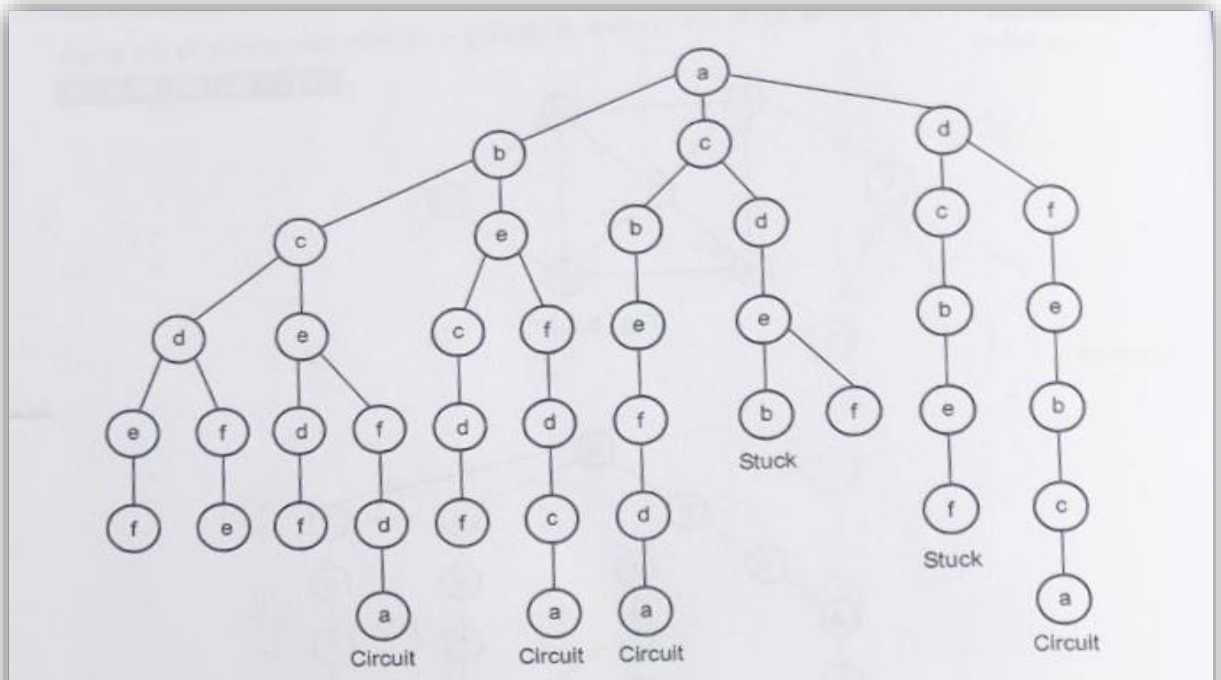
Solution : We will generate a state space tree for finding the Hamiltonian cycles.

Thus the Hamiltonian circuits are

a-b-c-e-f-d-a
a-b-e-f-d-c-a
a-c-b-e-f-d-a
a-d-f-e-b-c-a

Algorithm

```
Algorithm H_Cycle(k)
 // This is recursive backtracking algorithm for finding Hamiltonian cycle
//The graph G[1:n,1:n] is adjacency matrix used for graph G
//The cycle begins from the vertex 1
{
 repeat
 {
  Next Vertex(k);//generates next legal vertex for determining cycle
  if(x[k]=0) then
    return;
  if(k=n) then
   write(x[1:n])  // print the Hamiltonian cycle
  else
     H Cycle(k+1);
 } until(false);
}
```



The algorithm for generating next vertex which helps in finding Hamiltonian cycle is as given below -

```
Algorithm Next_Vertex(k)
// This algorithm finds the Hamiltonian path by choosing appropriate
// vertex each time. The x[1:k-1] is a Hamiltonian path.
//Each time the x[k] is assigned to a next highest numbered vertex
// which is not visited earlier. If x[k]=0 then no vertex is assigned
// to x[k].
//n denotes total number of vertices
{
 while(1)
 {
      //obtain next vertex
   x[k] : =x[k+1] mod (n+1);
   if(x[k]=0) then
      return;
  if(G[x[k-1],x[k]]=1) then // if edge between vertex k and
                            // k-1 is present
  {
    for (j=1 to k - 1 ) do //for every adjacent vertex
      if(x[j]=x[k]) then
        break;// not a distinct vertex
    if(j=k) then // obtained a distinct vertex
    {
```

```
    if((k<n) OR ((k=n) AND G[x[n],x[1]]=1)) then
      return;//return a distinct vertex
  }
 }
}
```

Assignment Problem

Problem statement : There are n people to whom n Jobs are to be assigned so that total cost of assignment is as small as possible. The assignment problem is specified by n by n matrix and each element in each row has to be selected in such a way that no two selected elements are in the same column and their sum is smallest as far as possible.

Let us take one example and understand this problem.

Job1	Job2	Job3	Job4	
10	2	7	8	Person a
6	4	3	7	Person b
5	8	1	8	Person c
7	6	10	4	Person d

If we select minimum value from each row then we get,

10	(2)	7	8
6	4	3	7
5	8	1	8
7	6	10	(4)

$\Rightarrow 2 + 3 + 1 + 4 = 10$

Hence set lower bound = 10.

Now the state space tree can be drawn step by step.

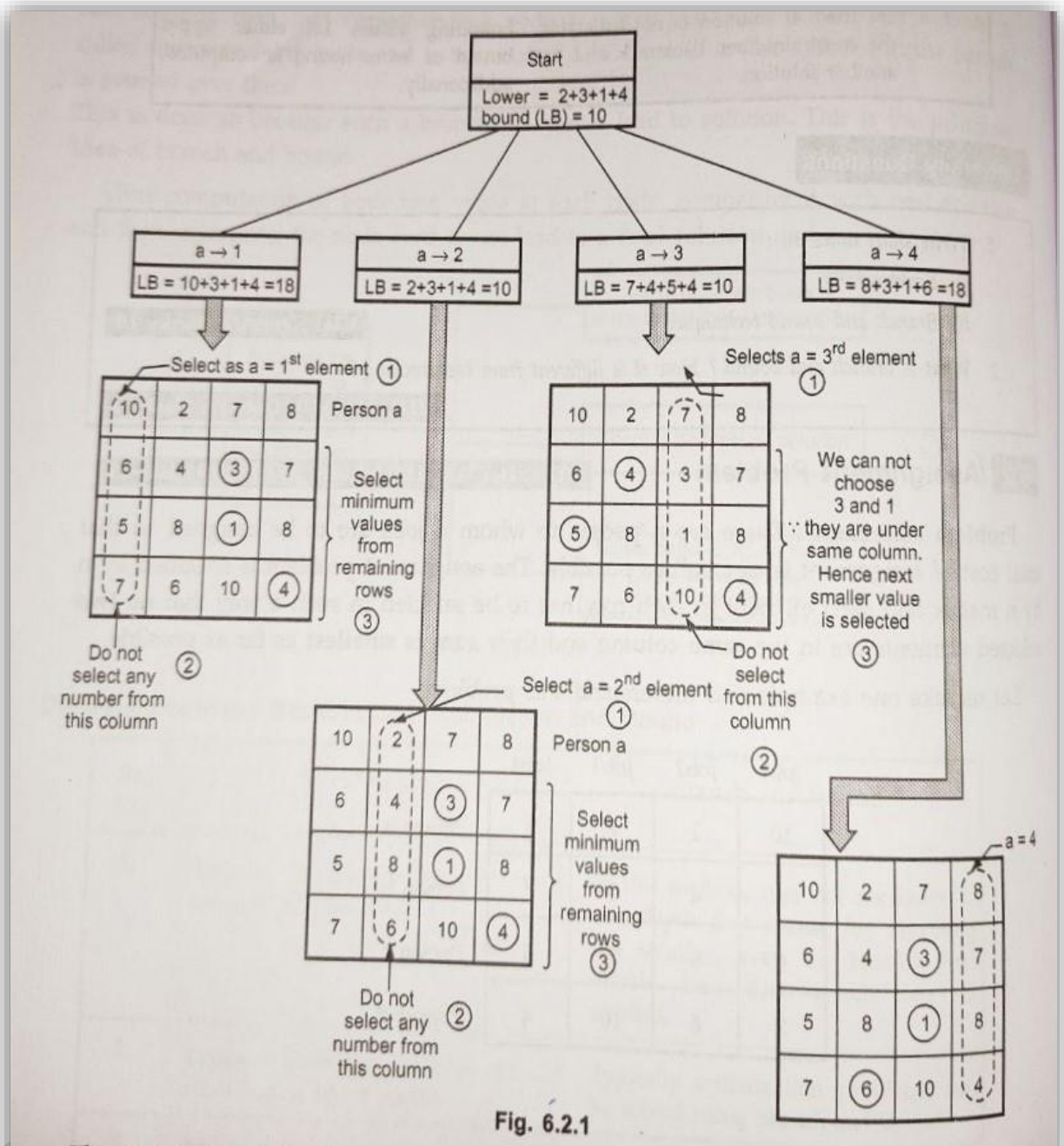


Fig. 6.2.1

Thus we have set jobs for person a, then we select jobs for person b then for person c and finally for person d.

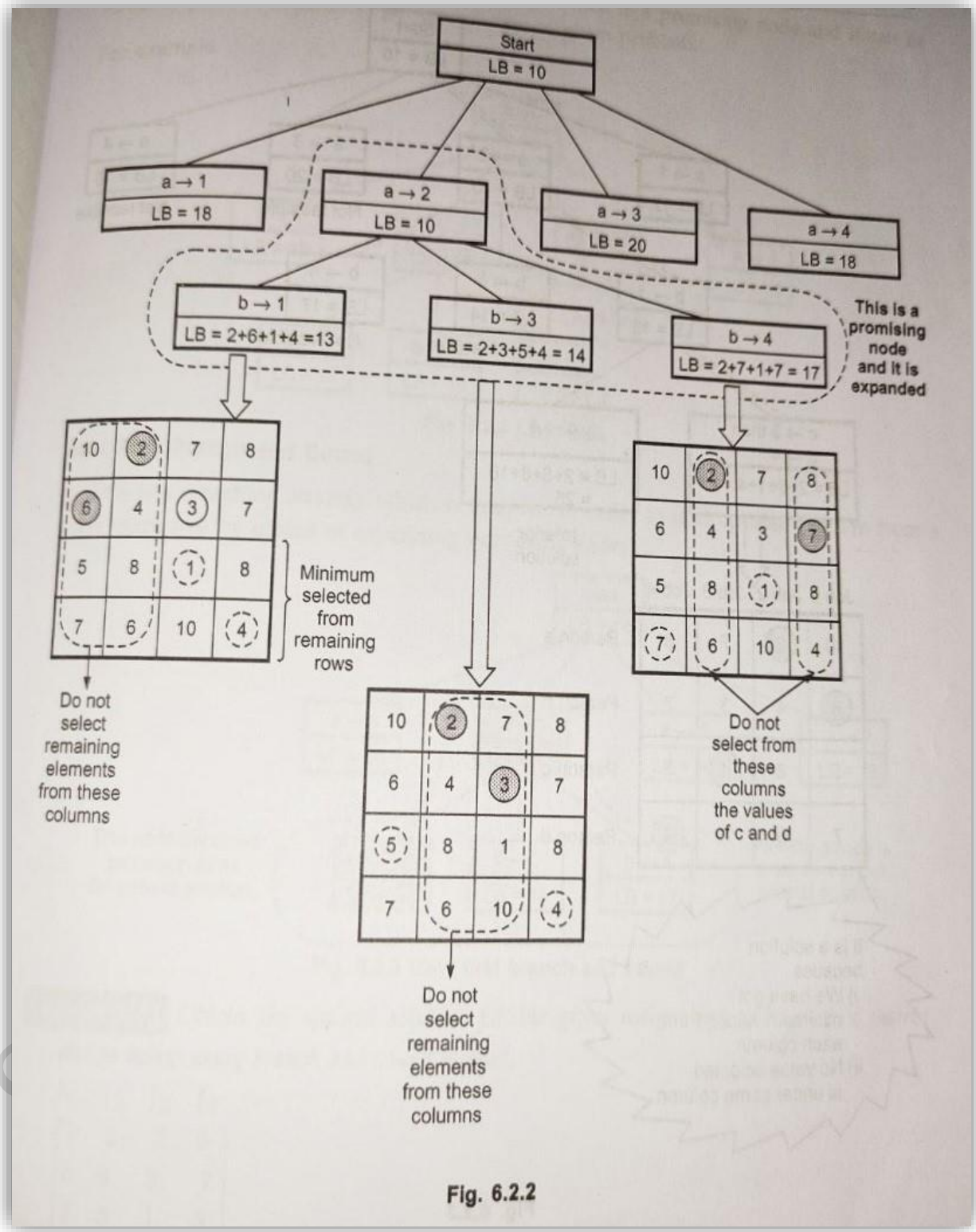

Start
LB = 10
a → 1
LB = 18
a → 2
LB = 10
a → 3
LB = 20
a → 4
LB = 18
b → 1
LB = 2+6+1+4 =13
b → 3
LB = 2+3+5+4 = 14
b → 4
LB = 2+7+1+7 = 17
This is a promising node and it is expanded
Minimum selected from remaining rows
Do not select remaining elements from these columns
Do not select from these columns the values of c and d
Do not select remaining elements from these columns

Fig. 6.2.2

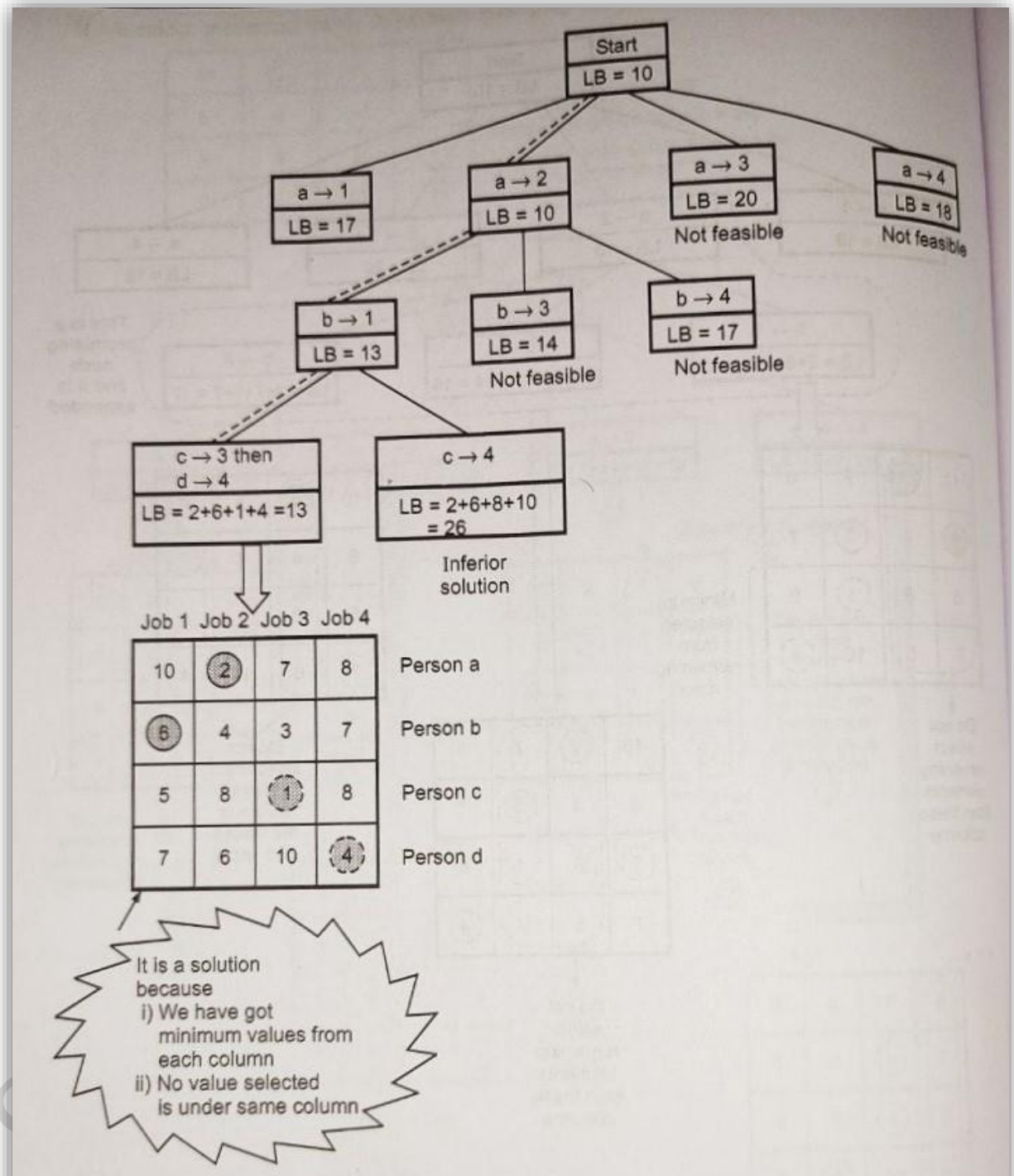


Fig. 6.2.3

Thus assignment problem can be solved with best fit branch and bound algorithm.

Let us understand some commonly used terminologies in branch and bound.

Live node - It is a node in state space tree which is a promising node and it can be further expanded in order to get the solution to given problem.

For example

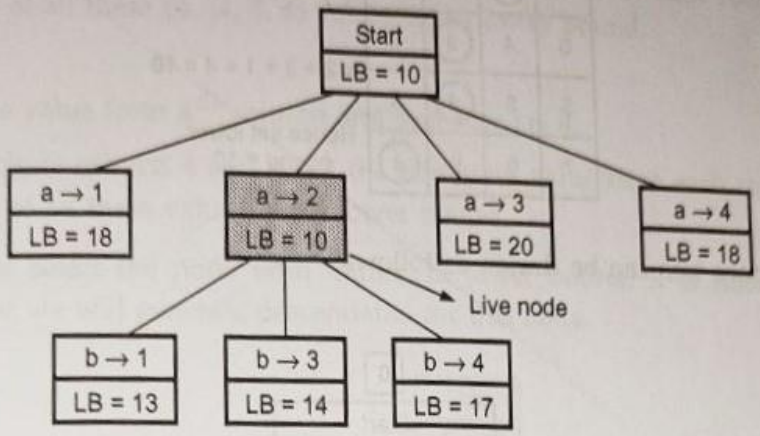


Fig. 6.2.4 Live node

Best First Branch and Bound

This is a searching strategy which is adopted in order to find optimal solution from a state space tree by means of expanding promising node.

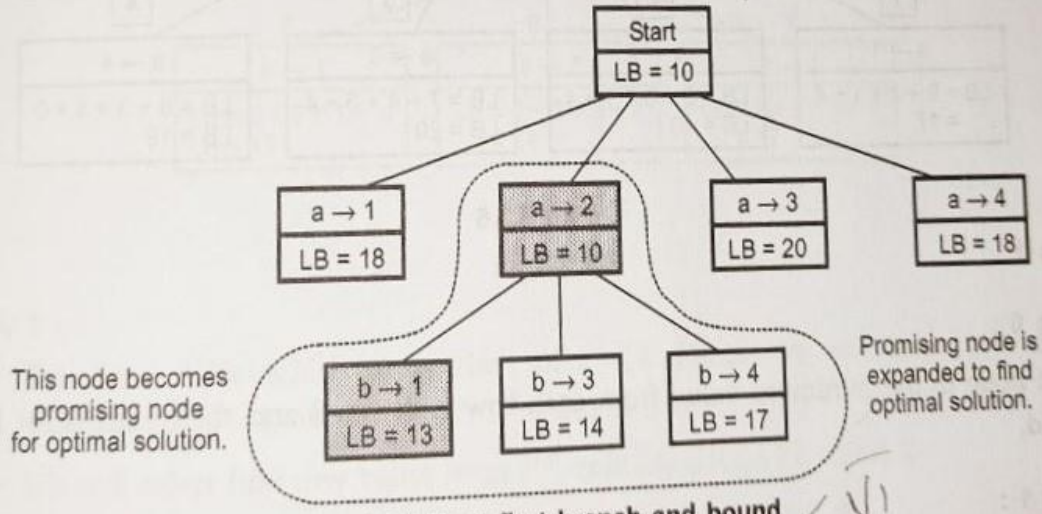


Fig. 6.2.5 Best first branch and bound

Travelling Sales Person Problem

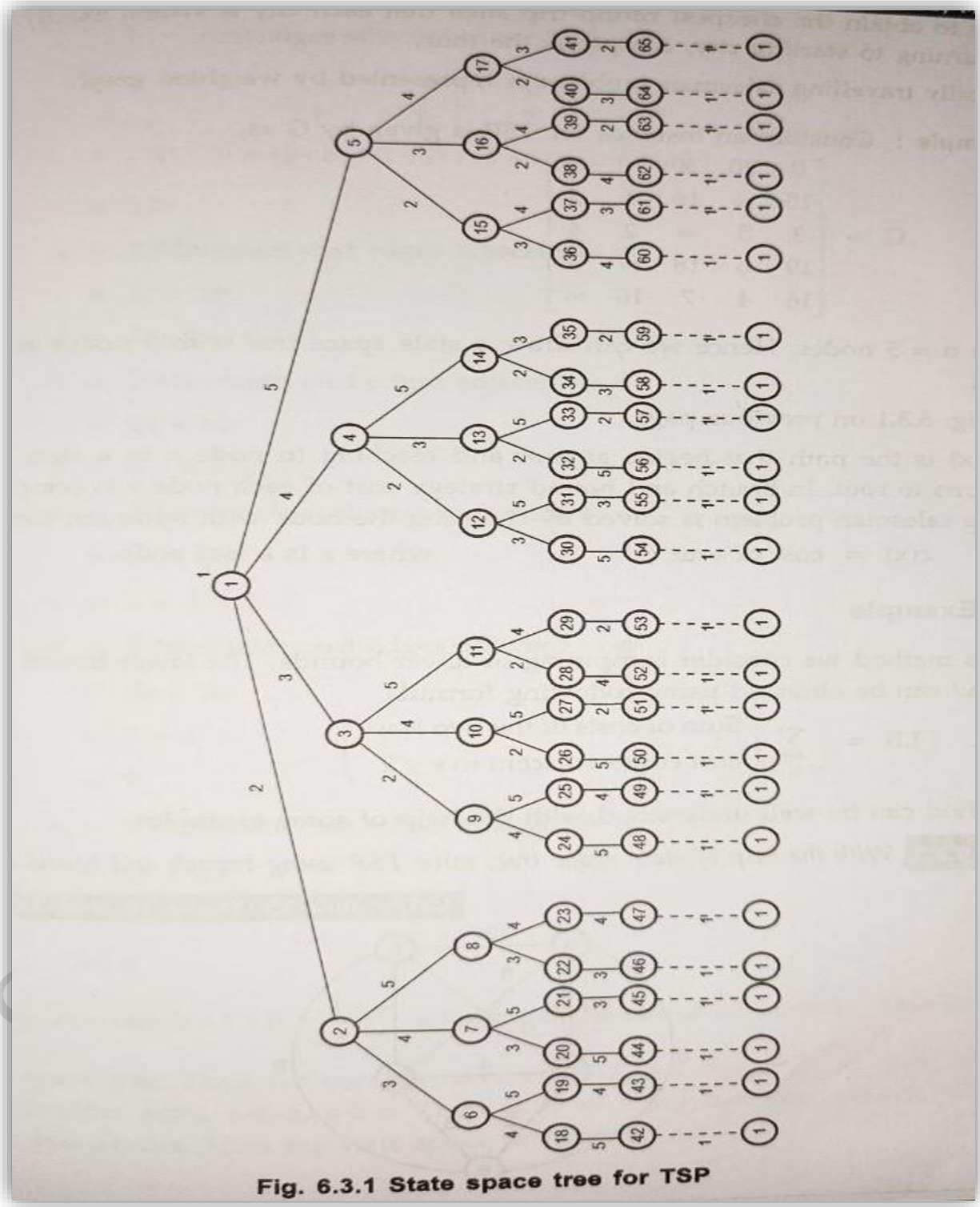


Fig. 6.3.1 State space tree for TSP

Problem Statement

If there are n cities and cost of travelling from any city to any other city is given then we have to obtain the cheapest round-trip such that each city is visited exactly once and then returning to starting city, completes the tour.

Typically travelling salesman problem is represented by weighted graph.

For example: Consider an instance for TSP is given by G as,

$$G = \begin{bmatrix} 0 & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

There n = 5 nodes. Hence we can draw a state space tree with 5 nodes as shown in Fig. 6.3.1. See Fig. 6.3.1 on previous page.

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 travelling salesman problem is solved by choosing the node with optimum cost. Hence,

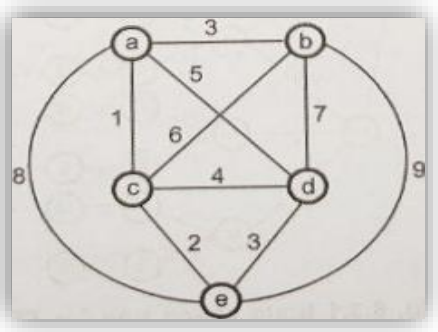
c(x) = cost of tour (x) where x is a leaf node.

Ex:-

In this method we consider computing of lower bounds. The lower bound is denoted by LB and can be obtained using following formula.

$$LB = \sum_{v \in V} \left(\begin{array}{l} \text{Sum of costs of the two least} \\ \text{cost edges adjacent to v} \end{array} / 2 \right)$$

This method can be well understood with the help of some examples.



Solution : We will first obtain Lower Bound LB as

$$LB = \left(\sum \begin{matrix}\text{Sum of costs of the two least} \\ \text{cost edges adjacent to v}\end{matrix} / 2 \right)$$

$$= \lceil [(1+3) + (3+6) + (1+2) + (3+4) + (2+3)] / 2 \rceil$$

$$= 14$$

Because

a = 2 Minimum cost edges adjacent to a

= ac + ab

a = 1 + 3 = 4

b = 2 Minimum cost edges adjacent to b

= ba + bc

b = 3 + 6 = 9

c = 2 Minimum cost edges adjacent to c

= ac + ce

c = 1 + 2 = 3

d = 2 Minimum cost edges adjacent to d

= de + dc

= 3 + 4

= 7

e = 2 Minimum cost edges adjacent to e

= ce + ed

= 2 + 3

= 5

Hence we have obtained LB = $\frac{1}{2}\sum_{v}$ adjacent distances of all vertices. This forms root of the state space tree. Then we consider a-b, a-c, a-d, a-e distances at level 1. Then at level 3 we consider a-b-c, a-b-d, a-b-e. Then at level 4 we consider a-b-c-d and a-b-c-e then a-b-d-c and a-b-d-e. Thus the state space tree can be

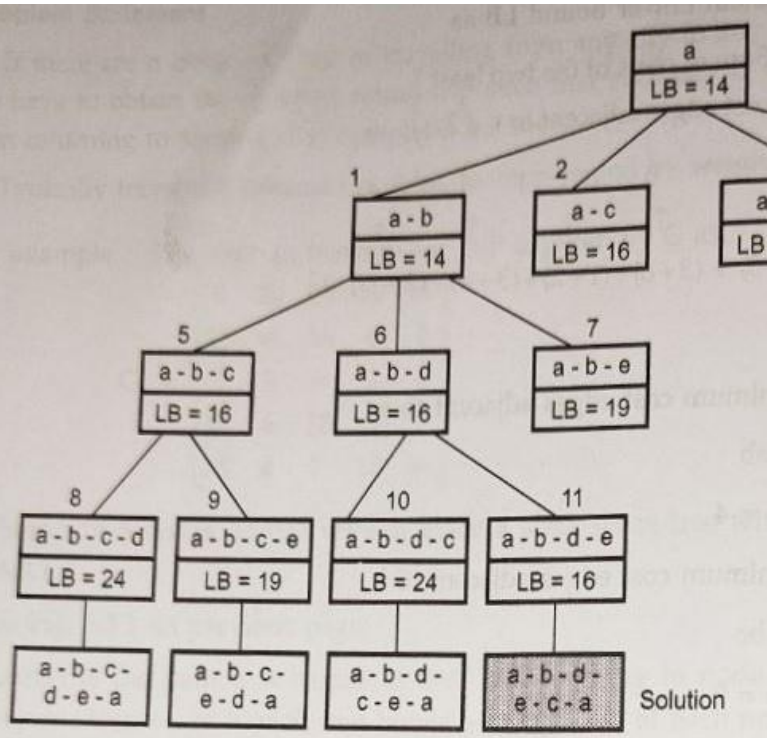


Fig. 6.3.2 State space tree

Explanation of above created state spaceTree

Consider node 1 : It says that consider distance a-b in computation of the corresponding vertices along with one minimum distance.

$\therefore$ $a = (a - b) + (a - c) = 3 + 1$

$b = (a - b) + (b - c) = 3 + 6$

$c = (a - c) + (c - e) = 1 + 2 \rightarrow$ can not consider (a - b) because an edge (a - b) is not adjacent to c.

$d = (d - e) + (c - d) = 3 + 4 \rightarrow$ can not consider (a - b)

$e = (c - e) + (d - e) = 2 + 3 \rightarrow$ can not consider (a - b)

$\therefore$ $[(3 + 1) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)]/2$

LB = 14 is for node 1.

Consider node 2 : It says that consider distance a-c in computation of corresponding vertices.

a = (a - b) + (a - c) = 3 + 1

b = (a - b) + (b - c) = 3 + 6 → can not consider (a - c) here because (a - c) is not adjacent to vertex b.

c = (a - c) + (c - e) = 1 + 2

d = (d - e) + (c - d) = 3 + 4 → can not consider (a - c) here

e = (c - e) + (d - e) = 2 + 3 → can not consider (a - c) here

∴ LB = [(3 + 1) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)]/2

LB = 14 is for node 2.

But can not be considered because b is before c.

Consider node 3 : It says that consider distance a-d in computation of corresponding vertices.

a = (a - c) + (a - d) = 15

b = (a - b) + (b - c) = 3 + 6 → can not consider (a - d)

c = (a - c) + (c - e) = 1 + 2 → can not consider (a - d)

d = (d - e) + (a - d) = 3 + 5

e = (c - e) + (d - e) = 2 + 3

∴ LB = [(1 + 5) + (3 + 6) + (1 + 2) + (3 + 5) + (2 + 3)]/2

= 32/2

LB = 16

Consider node 4 : This node say include edge a - e wherever possible.

a = (a - c) + (a - e) = 1 + 8 = 9

b = (a - b) + (b - c) = 3 + 6 = 9 → not possible to consider (a - e)

c = (a - c) + (c - e) = 1 + 2 = 3 → not possible to consider (a - e)

d = (c - d) + (d - e) = 4 + 3 = 7 → not possible to consider (a - e)

e = (c - e) + (a - e) = 2 + 8 = 10

∴ LB = (9 + 9 + 3 + 7 + 10)/2

LB = 19

Consider node 5 : This node says path a-b-c i.e. include edges (a – b), (b – c) wherever possible.

a = (a - b) + (a - c)
= 3 + 1 → can not include (b - c) here because (b - c) is not
= 4 adjacent to a.

b = (a - b) + (b - c)
= 3 + 6
= 9

c = (a - c) + (b - c) → Min. distance (a - c) is included but (a - b)
= 1 + 6 but (a - b) can not be included as it is not adjacent to c.
= 7

d = (d - e) + (d - c)
= 3 + 4
= 7

e = (c - e) + (d - e)
= 2 + 3
= 5

∴ LB = [4 + 9 + 7 + 7 + 5]/2
= 32/2

LB = 16

Similarly we can compute LB at node 6, 7, 8, 9, 10.

Consider node 8 : It says a-b-c-d that means include (a - b), (b - c), (c - d) whichever is minimum and whichever is applicable. As this is the leaf node and from this node we try to reach to source node. That is after a-b-c-d we go to e and from e-to-a. Hence

a → (a - b) + (a - e) = 3 + 8 = 11

b → (a - b) + (b - c) = 3 + 6 = 9

c → (b - c) + (c - d) = 6 + 4 = 10

d → (c - d) + (d - e) = 4 + 3 = 7

e → (a - e) + (d - e) = 8 + 3 = 11

$\therefore$ LB = [11 + 9 + 10 + 7 + 11]/2

LB = 48/2

LB = 24

Consider node 11 : It says a-b-d-e. That means include (a - b), (b - d), (d - e) in computation.

a = (a - b) + (a - c) = 3 + 1 = 4

b = (a - b) + (b - d) = 3 + 7 = 10

c = (a - c) + (c - e) = 1 + 2 = 3

d = (b - d) + (d - e) = 7 + 3 = 10

e = (c - e) + (d - e) = 2 + 3 = 5

$\therefore$ LB = (4 + 10 + 3 + 10 + 5)/2

= 32/2

= 16

At node 11 we get optimum tour i.e. a-b-d-e.

Hence the optimum cost tour of TSP is **a-b-d-e-c-a** with **cost 16.**

Knapsack Problem

In this section we will discuss "How Knapsack problem be solved using branch and bound?" The problem can be stated as follows,

"If we are given n objects and a Knapsack or a bag in which the object i with weight W_i is to be placed. The Knapsack has a capacity W. Then the profit value that can

be earned is VI. Then objective is to obtain filling of Knapsack with maximum profit earned." But it should not exceed weight W of the Knapsack.

To fill the given Knapsack we select the object with some weight and having some profit. This selected object is put in the Knapsack. Thus the Knapsack is filled up with selected objects. Note that the Knapsack's capacity W should not be exceeded. Hence the first item gives best pay off per weight unit. And last one gives the worst pay off per weight unit.

$$v_1/w_1 \geq v_2/w_2 \geq v_3/w_3 \ldots\ldots v_n/w_n$$

First of all we compute upper bound of the tree.

We design a state space tree by inclusion or exclusion of some items.

The upper bound can be computed using following formula.

$$ub = v + (W - w)\,(v_{i+1}/w_{i+1})$$

Consider 4 items as,

Item	Weight	Value	Value/Weight
1	4	5 40	10
n2	7	$ 42	6
3	5	$ 25	5

4	3	$ 12	4

W= Capacity of Knapsack = W = 10

and find the knapsack solution using Branch and Bound and draw state space tree.

Solution : We will first compute the upper bond by using above given formula -

$$ub = v + (W - w)\,(v_{i+1}/w_{i+1})$$

Initially $v = 0$, $w = 0$ and $v_{i+1} = v_1 = 40$ and $w_{i+1} = w_1 = 4$. The capacity $W = 10$.

$$\therefore \quad ub = 0 + (10 - 0)\,(40/4)$$

$$= (10)\,(10)$$

$$ub = 100\ \$$$

Now we will construct a state space tree by selecting different items.

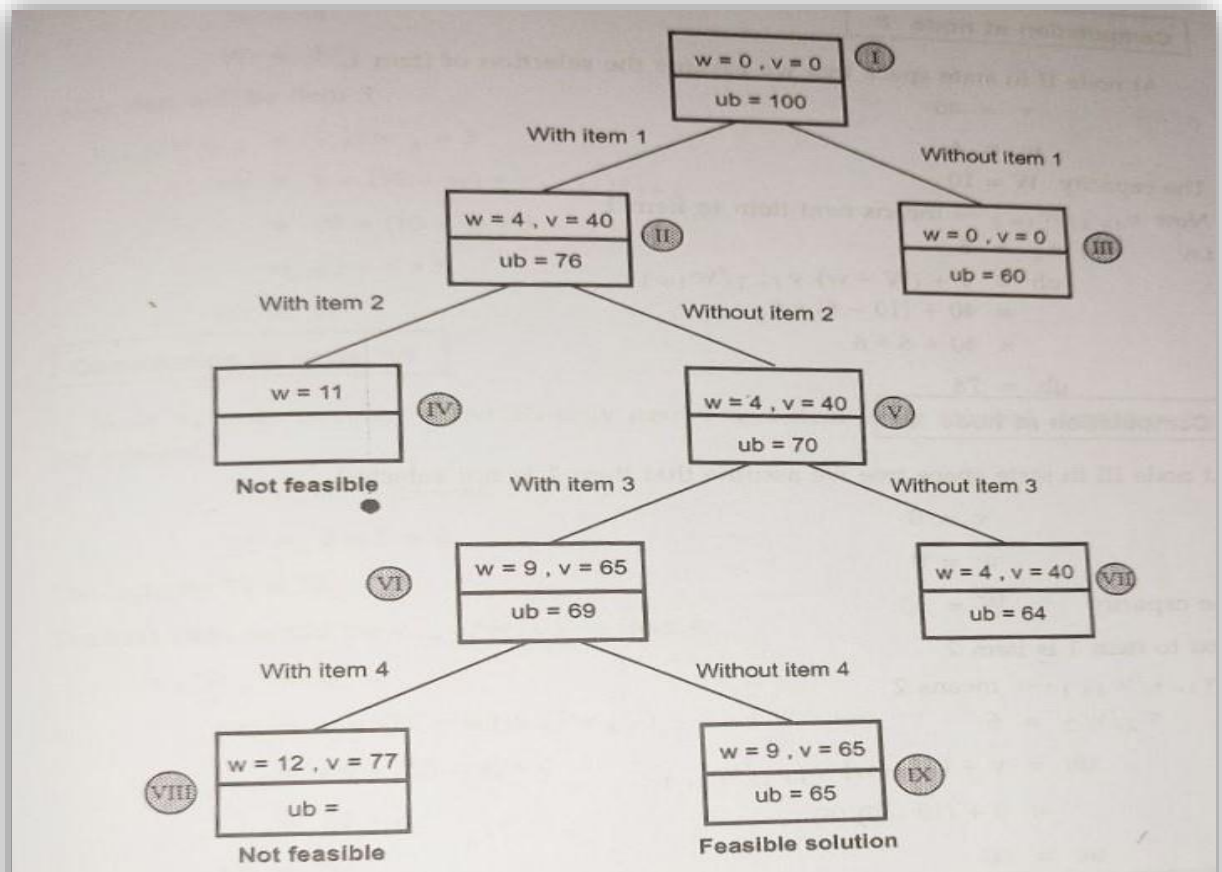


Fig. 6.4.1 State space tree for Knapsack problem

Computation at node 1 i.e. root of state space tree.

Initially, $w = 0$, $v = 0$ and $v_{i+1}/w_{i+1} = v_1/w_1 = 40/4 = 10$.

The capacity $W = 10$.

$$\therefore \quad ub = v + (W - w)\, v_{i+1}/w_{i+1}$$

$$= 0 + (10 - 0)\,(10)$$

$$\therefore \quad ub = 100$$

Computation at node II

At node II in state space tree we assume the selection of item 1.

$\therefore$ $v = 40$

$w = 4$

The capacity $W = 10$

Now v_{i+1}/w_{i+1} → means next item to item 1

i.e. $v_2/w_2 = 6$

$\therefore$ $ub = v + (W - w)\, v_{i+1}/w_{i+1}$

$= 40 + (10 - 6) * 6$

$= 40 + 6 * 6$

$ub = 76$

Computation at node III

At node III in state space tree we assume that item 1 is not selected.

$\therefore$ $v = 0$

$w = 0$

The capacity $W = 10$

Next to item 1 is item 2

$\therefore$ v_{i+1}/w_{i+1} → means 2

i.e. $v_2/w_2 = 6$

$\therefore$ $ub = v + (W - w)\, v_{i+1}/w_{i+1}$

$= 0 + (10 - 0)\,(6)$

$ub = 60$

Computation at node IV

This is a node at which we have selected item 1 and item 2 already

$\therefore$ $v_{i+1}/w_{i+1} = v_3/w_3 = 25/5 = 5$

$\therefore$ $v = 40 + 42 = 82$

$w = 4 + 7 = 11$

The capacity W = 10 as we get w = 11. Hence it is not feasible.

Computation at node V

At this node item 2 is not selected and only item 1 is selected.

$v = 40$

$\therefore$ $w = 4$

$W = 10$

Next item will be item 3

$\therefore v_{i+1}/w_{i+1} = v_3/w_3 = 5$

$\therefore$ $ub = v + (W - w) * v_{i+1}/w_{i+1}$

$= 40 + (10 - 4) * 5$

$= 40 + 6 * 5$

$ub = 70$

Computation at node VI

Node VI is an instance at which only item 1 and item 3 are selected. And item 2 is not selected.

$\therefore$ $v = 40 + 25 = 65$

$w = 4 + 5 = 9$

The capacity W = 10.

The next item would be $v_{i+1}/w_{i+1} \rightarrow$ item 4

$\therefore$ $v_4/w_4 = 4$

$\therefore$ $ub = v + (W - w)\ v_4/w_4$

$= 65 + (10 - 9) * 4$

$= 65 + 4$

$ub = 69$

Computation at node VII

Node VII is an instant at which item 1 is selected, item 2 and item 3 are not selected.

$\therefore$ $v = 40$

$w = 4$

$W = 10$

The next item being selected is item 4.

$v_{i+1}/w_{i+1} = v_4/w_4 = 4$

$\therefore$ $ub = v + (W - w) * v_{i+1}/w_{i+1}$

$= 40 + (10 - 4) * 4$

$= 40 + 24$

$ub = 64$

Computation at node VIII

At node VIII, we consider selection of item 1, item 3, item 4. There is no next item given problem statement. $\therefore v_{i+1}/w_{i+1} = 0$

$\therefore \quad w = 4 + 5 + 3 = 12 \rightarrow$ But this is exceeding capacity W = 10.

$v = 40 + 25 + 12 = 77$

But as weight of selected items exceed the capacity W this is **not a feasible solution.**

Computation at node IX

At node IX, we consider selection of item 1, and item 3.. There is no next item given.

$\therefore v_{i+1}/w_{i+1} = 0$

$\therefore \quad w = 4 + 5 = 9$

$v = 40 + 25 = 65$

$W = 10$

$\therefore \quad ub = v + (W - w) * v_{i+1}/w_{i+1}$

$= 65 + (10 - 9) * 0$

$ub = 65$

The node IX is a node indicating maximum profit of selected items with maximum weight of item = 9 i.e. < capacity of Knapsack (W = 10).

Thus for the given instance of Knapsack's problem we get {i[redacted]m 4} with maximum weight 9 and maximum profit gained = 65$ as a solution.

NP-Complete NP-Hard Problems

Basic Concepts

There are two groups in which a problem can be classified. The first group consists of the problems that can be solved in polynomial time.

For example : searching of an element from the list O(logn), sorting of elements 0(logn).

The second group consists of problems that can be solved in non-deterministic

Polynomial time. For example: Knapsack problem 0(2n/2) and Travelling Salesperson problem (0(n22n)).

- Any problem for which answer is either yes or no is called decision problem. The algorithm for decision problem is called decision algorithm.
- Any problem that involves the identification of optimal cost (minimum or
- Maximum) is called optimization problem. The algorithm for optimization problem is called optimization algorithm.
- Definition of P - Problems that can be solved in polynomial time. ("p" stands for polynomial). The polynomial time is nothing but the time expressed in terms of polynomial.
- Examples - Searching of key element, Sorting of elements, All pair shortest path.
- Definition - It stands for "non-deterministic polynomial time". Note that
- NP does not stand for "non-polynomial".
- Examples - Travelling Salesperson problem, Graph coloring problem, Knapsack problem, Hamiltonian circuit problems.
- The NT class problems can be further categorized into NP-complete and NP hard problems.
- A problem D is called NP-complete if -
    1. It belongs to class NP
    2. Every problem in NP can also be solved in polynomial time.
- If an NP-hard problem can be solved in polynomial time then all NP-complete problems can also be solved in polynomial time)

- All NP-complete problems are NP-hard but all NP-hard problems cannot be NP-complete.

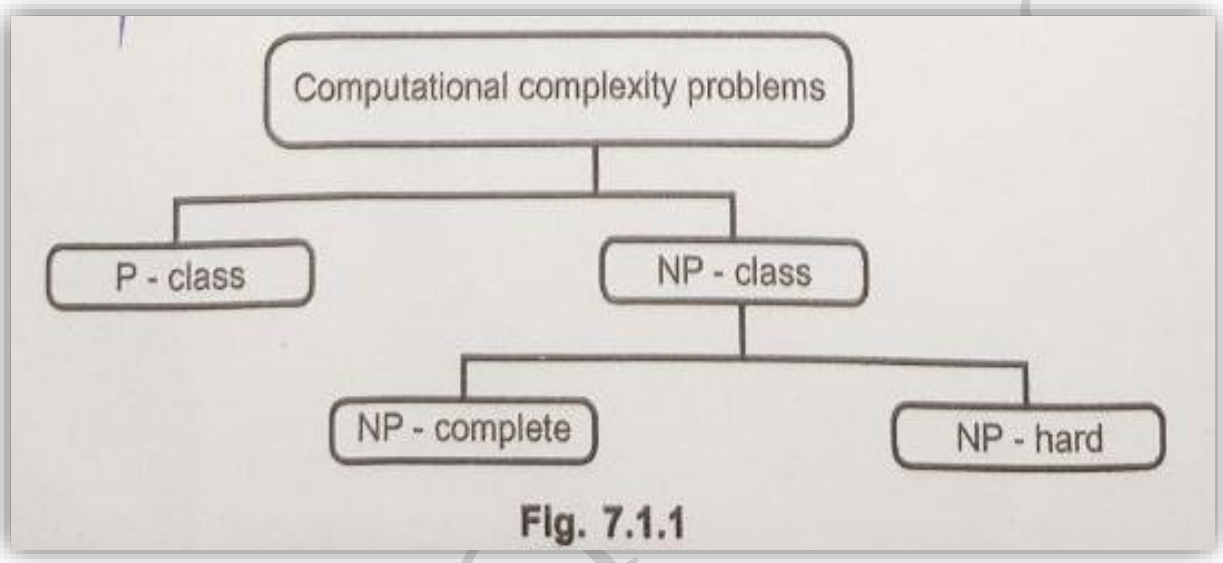


Fig. 7.1.1

- The NP class problems are the decision problems that can be solved by non-deterministic polynomial algorithm.
- Computational Complexity - The computational problems is an infinite collection of instances with a solution for every instance.
  - In computational complexity the solution to the problem can be "yes" or "no" type. Such type of problems are called decision problem.
  - The computational problems can be function problem. The function problem is a computational problem where single output is expected for every input.
  - The output of such type of problems is more complex than the decision problem. The computational problems also consists of a class of problems, whose output can be obtained in polynomial time.
- Complexity classes - The complexity classes is a set of problems of related

complexity. It includes function problems, P classes, NP classes, optimization problem.

- Intractability - Problems that can be solved by taking a long time for their Solutions is known as intractable problems.
- If NP is not same as P then NP complete problems are called intractable problems.

Non-Deterministic Algorithms

- The algorithm in which every operation is uniquely defined is called deterministic algorithm.
- The algorithm in which every operation may not have unique result, rather there can be specified set of possibilities for every operation. Such an algorithm is called non-deterministic algorithm. Non deterministic means that no particular rule is followed to make the guess.
- The non-deterministic algorithm is a two stage algorithm -
  - Non-deterministic (Guessing) stage - Generate an arbitrary string that can be thought of as a candidate solution.
  - Deterministic ("Verification") stage - In this stage it takes as input the candidate solution represents actual solution.

```
Algorithm Non_Determin()
// A[1:n] is a set of elements
// we have to determine the index i of A at which element x is
//located.
{
  // The following for-loop is the guessing stage
        for i=1 to n do
          A[i] := choose(i);

  // Next is the verification(deterministic) stage
      if (A[i] = x) then
      {
            write(i);
            success();
      }
      write(0);
      fail();
}
```

```
Algorithm Non_DSort(A,n)
// A[1:n] is an array that stores n elements which are positive integers B[1:n] be an
auxiliary
// array in which elements are put at appropriate positions
{
// guessing stage
     for i←1 to n do
          B[i]←0;//initialize B to 0
     for i←1 to n do
     {
          j←choose(1,n)//select any element from the input set
          if(B[j]!=0) then
               fail();
          B[j]←A[i];
     }
//verification stage
     for i←1 to n-1 do
          if(B[i]>B[i+1]) then
               fail();// not sorted elements
          write(B[1:n]);// print the sorted list of elements
          success();
}
```

P,NP, NP-Complete and NP-Hard Classes

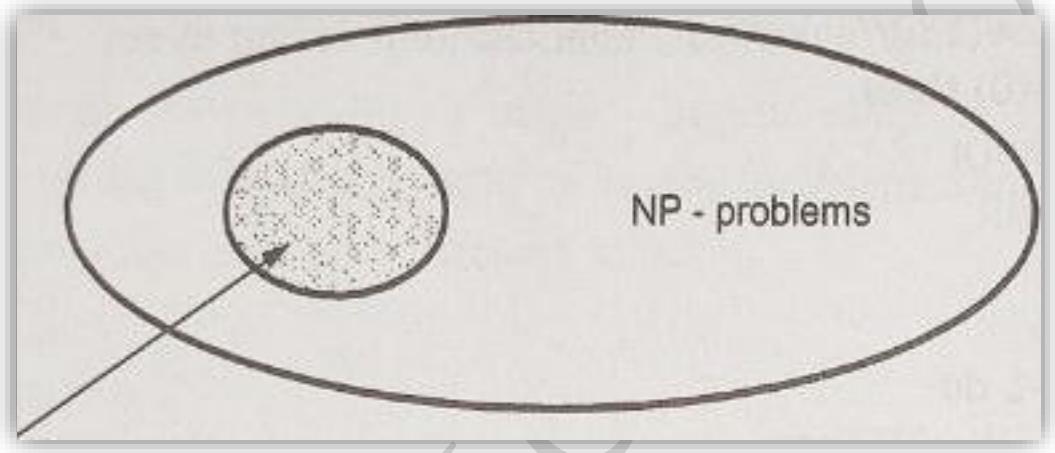
•As we know, P denotes the class of all deterministic polynomial language problems and NP denotes the class of all non-deterministic polynomial language problems. Hence

The question of whether or not

P = NP

holds, is the most famous outstanding problem in the computer science. Problems which are known to lie in P are often called as tractable. Problems which lie outside of P are often termed as intractable. Thus, the question of whether P = NP or P NP is the same as that of asking whether there exist problems in NP which are intractable or not.

The relationship between P and NP is depicted by following figure –



We don't know if P = NP. However in 1971 S.A. Cook proved that a particular NP problem known as SAT (Satisfiability of sets of boolean clauses) has the property that, if it is solvable in polynomial time, so are all NP problems. This is what is called a "NP-complete" problem.

Let A and B are two problems then problem A reduces to B if and only if there is a way to solve A by deterministic polynomial time algorithm using a deterministic algorithm that solves B in polynomial time.

A reduces to B can be denoted as A α B. In other words we can say that if there exists any polynomial time algorithm which solves B then we can solve A

in polynomial time. We can also state that if A α B and B α C then A α C.

A NP problem such that, if it is in P, then NP = P. If a (not necessarily NP)

problem has this same property then it is called "NP-hard".

Thus the class d NP-complete problem is the intersection of the NP and NP-hard classes.

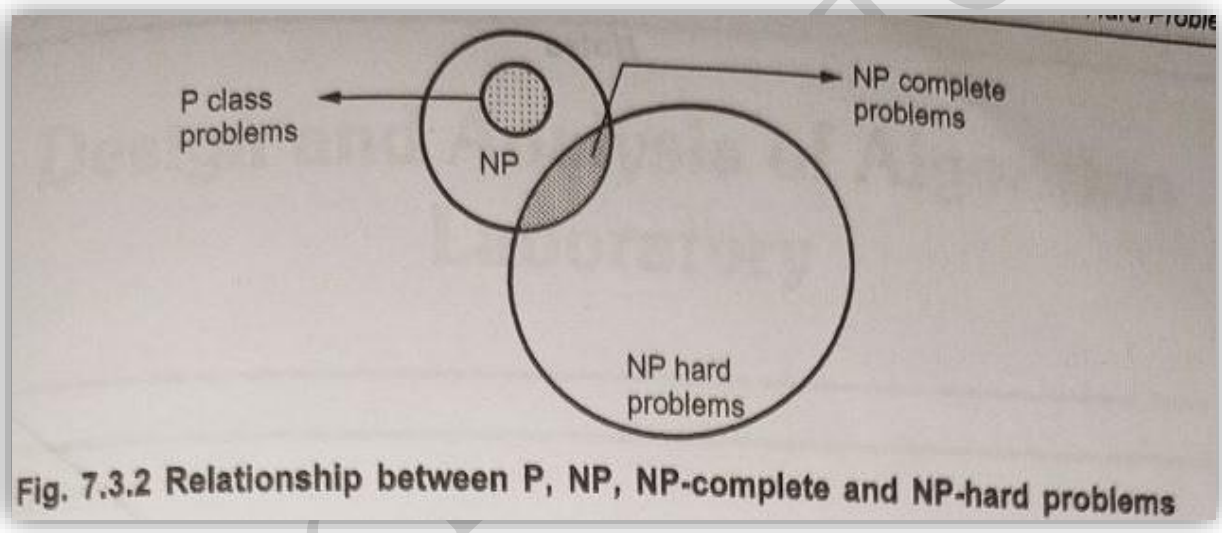


Fig. 7.3.2 Relationship between P, NP, NP-complete and NP-hard problems

Normally the decision problems are NP-complete but optimization problems are

NP-hard. However if problem A is a decision problem and B is optimization

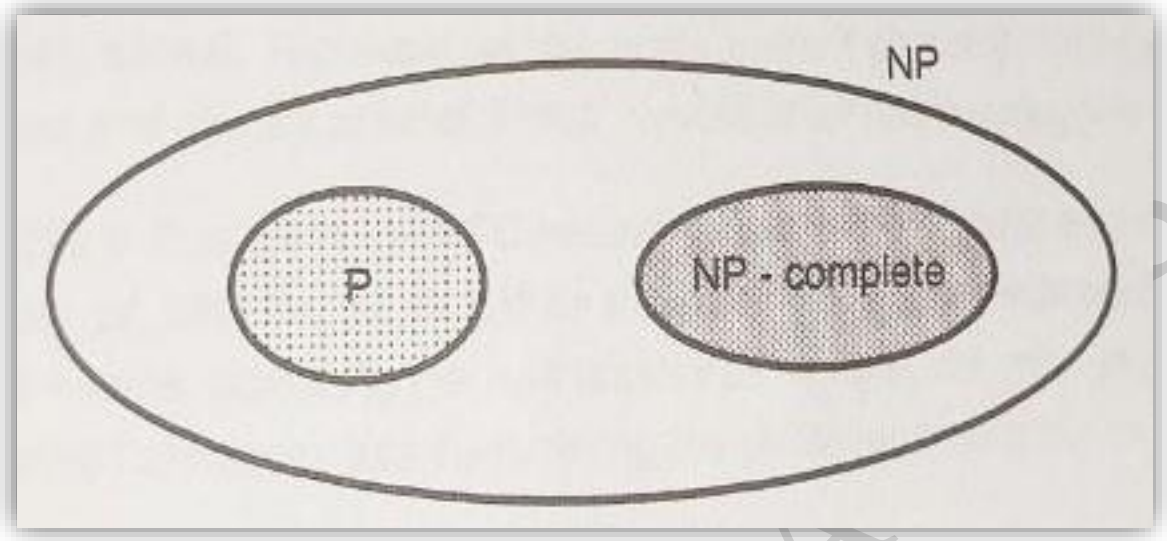
problem then it is possible that A α B.

For instance the knapsack decision problem can be knapsack optimization problem.

There are some NP-hard problems that are not NP-complete. For example halting problem. The halting problem states that: "Is it possible to determine whether an algorithm will ever halt or enter in a loop on certain input?"

Two problems P and Q are said to be polynomial equivalent if and only if

P α Q and Q α P.



Thanking You

Visit Our Official Website

http://searchcreators.org/