UNIT - IV

6 Backtracking

Syllabus

General method, Recursive backtracking algorithm, Iterative backtracking method. n-Queen problem, Sum of subsets, Graph coloring, 0/1 Knapsack Problem.

Contents

6.1 General Method

SPPU : Dec.-06,10,11,12,13, May-08,11,15, 18, Marks 18

- In the backtracking method
  1. The desired solution is expressible as an n tuple $(x_1, x_2,...x_n)$ where x_i is chosen from some finite set S_i.
  2. The solution maximizes or minimizes or satisfies a criterion function $C(x_1, x_2,...x_n)$.
- The problem can be categorized into three categories. For instance - 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.

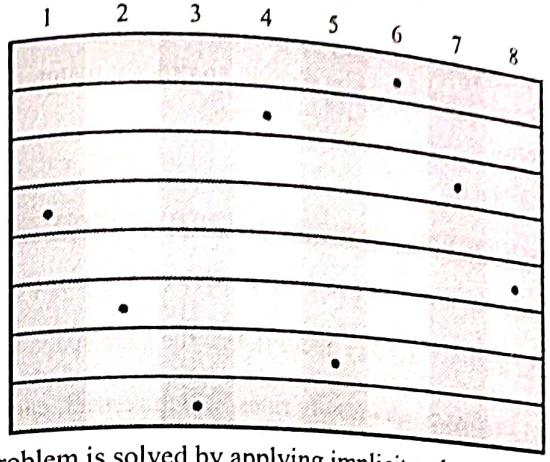
  Listing of all the feasible solution - Is the enumeration problem.
- The basic idea of backtracking is to build up a vector, one component at a time and to test whether the vector being formed has any chance of success.
- The major **advantage** of backtracking algorithm is that we can realize the fact 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_i 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 x_i 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 as follows. Consider a chessboard of order 8 × 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 to 8-queens problem can be obtained using backtracking method.

- 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 S_i must be = {1, 2, 3, 4, 5, 6, 7, 8} with $1 \le i \le 8$. Hence solution space consists of 8^8 8-tuples.

The **implicit constraint** will be -

1) No two x_i 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}.

Example 2 : Sum of subsets -

There are n positive numbers given in a set. The desire is to find all possible subsets of this set, the contents of which add onto a predefined value M.

In other words,

Let there be n elements given by the set $w = (w_1, w_2, w_3,..., w_n)$ then find out all the subsets from whose sum is M.

For example

Consider n = 6 and $(w_1, w_2, w_3, w_4, w_5, w_6) = (25, 8, 16, 32, 26, 52)$ and M = 59 then we will get desired sum of subset as (25, 8, 26).

We can also represent the sum of subset as (1, 1, 0, 0, 1, 0). If solution subset is represented by an n-tuple $(x_1, x_2,...x_n)$ such that x_i could be either 0 or 1. The $x_i = 1$ means the weight w_i is to be chosen and $x_i = 0$ means that weight w_i is not to be chosen.

The **explicit constraint** on sum of subset problem will be that any element $x_i \in \{i | i$ is an integer and $1 \le i \le x\}$. That means we must select the integer from given set of elements.

The **implicit constraints** on sum of subset problem will be -

1) No two elements will be the same. That means no element can be repeatedly selected.
2) The sum of the elements of subset = M (a predefined value).
3) The selection of the elements should be done in an orderly manner. That means (1, 3, 7) and (1, 7, 3) are treated as same.

Ex. 6.1.1 *Define following terms : State space, explicit constraints, implicit constraints, problem state, solution states, answer states, live node, E-node, dead node, bounding functions.*
SPPU : May-11, Marks 8, Dec.-11, Marks 16, Dec-12, Marks 18, May 18, End Sem., Marks 8

Sol. : The tree organization for 4-queen's problems is shown by given Fig. 6.1.1(on next page).

State space : All paths from root to other nodes define the state space of the problem.

Explicit constraints : Explicit constraints are rules, which restrict, each vector element to be chosen from given set.

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

Problem states : Each node in the state space tree is called problem state.

Solution states : The solution states are those problem states s for which the path from root to s defines a tuple in the solution space. In some trees the leaves define solution states.

Answer states : These are the leaf nodes which correspond to an element in the set of solutions. These are the states which satisfy the implicit constraints.

Live node : A node which is generated and whose children have not yet been generated is called live node.

E-node : The live node whose children are currently being expanded is called E-node.

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

Bounding functions : Bounding functions will be used to kill live nodes without generating all their children. A care should be taken while doing so, because atleast one answer node should be produced or even all the answer nodes be generated if problem needs to find all possible solutions.

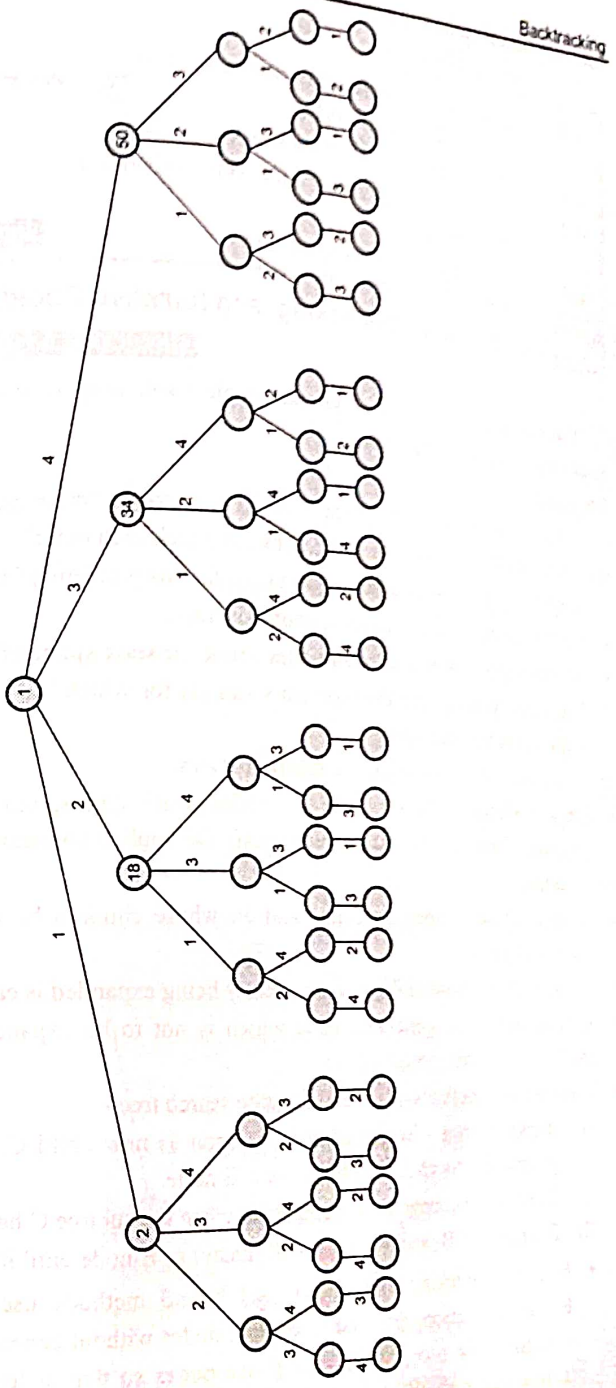


Fig. 6.1.1 State space tree for 4-queen's problems

Review Questions

Q.1 What are the constraints that must be satisfied while solving any problem using backtracking ? Explain briefly. **SPPU : Dec.-06, Marks 4, Dec.-10, Marks 6**

Q.2 Enlist the characteristics of backtracking strategy. **SPPU : May-08, Marks 2**

Q.3 Explain the following terms : Live nodes, expanding nodes, bounding function and solution space. **SPPU : Dec.-13, Marks 8**

Q.4 Write short note on state space tree. **SPPU : May-15, End Sem, Marks 2**

6.2 Recursive Backtracking and Iterative Backtracking Algorithm

SPPU : Dec.-06,07, 18, May-07,15, 18, Marks 8

Backtracking algorithms determine problem solutions by systematically searching for the solutions using **tree structure.**

For example -

Consider a 4-queen's problem. It could be stated as "there are 4 queens that can be placed on 4 × 4 chessboard. Then no two queens can attack each other".

Following Fig. 6.2.1 shows tree organization for this problem. [See Fig. 6.2.1 on next page]

- Each node in the tree is called a **problem state.**
- All paths from the root to other nodes define the **state space of** the problem.
- The solution states are the problem states (s) for which the path from root to s defines a **tuple** in the solution space.

In some trees the leaves define the **solution states.**

- **Answer states :** These are the leaf nodes which correspond to an element in the set of solutions, these are the states which satisfy the implicit constraints.

For example

- A node which is been generated and all whose children have not yet been generated is called **live node.**
- The live node whose children are currently being expanded is called **E-node.**
- A **dead node** is a generated node which is not to be expanded further or all of whose children have been generated.
- There are two methods of generating state search tree -
  - i) **Backtracking -** In this method, as soon as new child C of the current E-node N is generated, this child will be the new E-node.
    The N will become the E-node again when the subtree C has been fully explored.
  - ii) **Branch and Bound -** E-node will remain as E-node until it is dead.
- Both bracktracking and branch and bound methods use bounding functions. The bounding functions are used to kill live nodes without generating all their children. The care has to be taken while killing live nodes so that at least one answer node or all answer nodes are obtained.

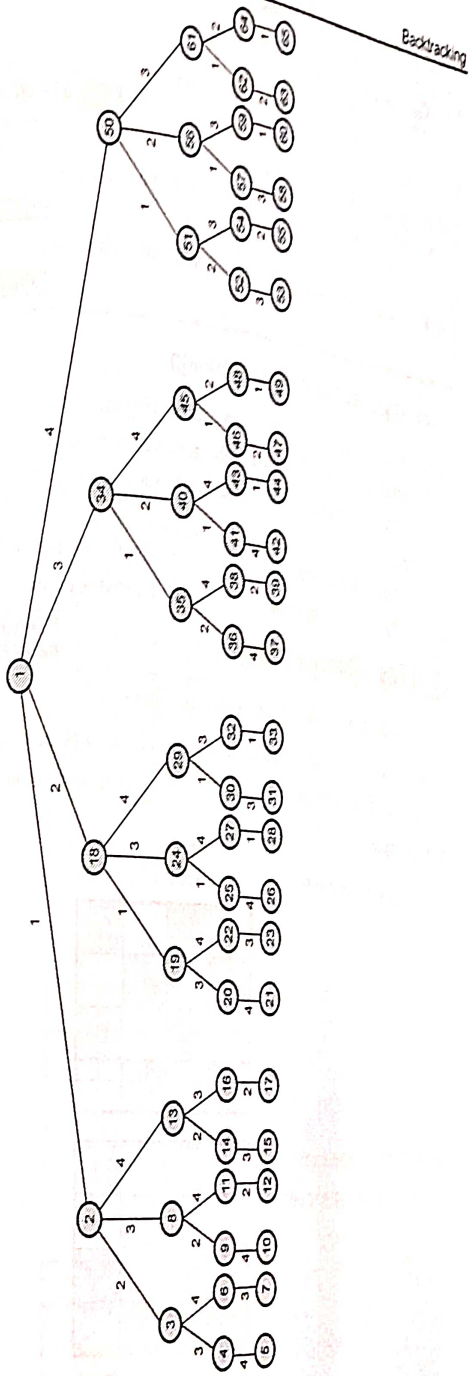


Fig. 6.2.1 State-space tree for 4-queen's problem

Review Questions

Q.1 Write a recursive algorithm which shows a recursive formulation of the backtracking technique. **SPPU : Dec.-06, Dec.-18, End Sem, Marks 8**

Q.2 Write a schema for an iterative backtracking method. **SPPU : May-07, Dec.-07, Marks 8**

Q.3 What is backtracking ? Write general iterative algorithm for backtracking. **SPPU : May-15, End Sem, Marks 8**

Q.4 Write a recursive and Iterative algorithm of backtracking method. **SPPU : May-18, End Sem, Marks 8**

6.3 Applications of Backtracking

Various applications that are based on backtracking method are -

1. 8 Queen's problem : This is a problem based on chess games. By this problem, it is stated that arrange the queens on chessboard in such a way that no two queens can attack each other.
2. Sub of subset problem.
3. Graph coloring problem.
4. Finding Hamiltonian cycle.
5. Knapsack problem.

6.4 The n-Queen Problem

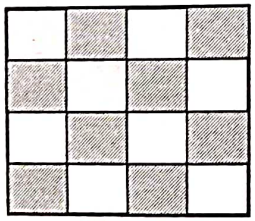
SPPU : May-11,12,14, 19, Marks 10

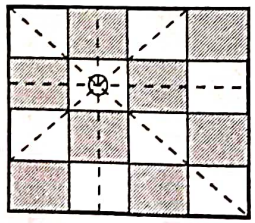
The n-queen's problem can be stated as follows.

Consider a n × n chessboard on which we have to place n 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

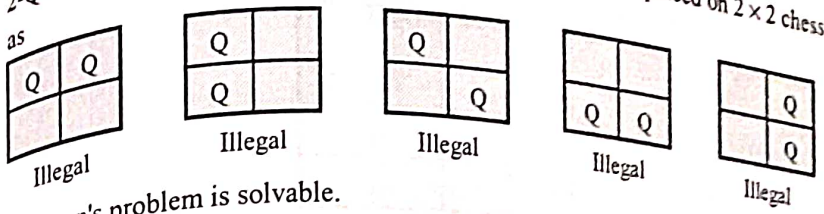
Consider 4 × 4 board





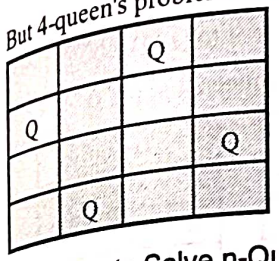
The next queen - if is placed on the paths marked by dotted lines then they can attack each other

- 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.



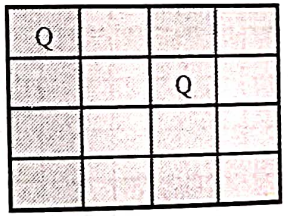
⇐ Note that no two queens can attack each other.

6.4.1 How to Solve n-Queen's Problem ?

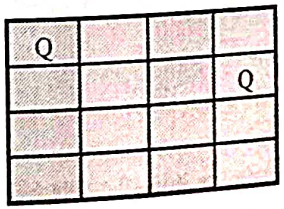
Let us take 4-queens and 4 × 4 chessboard.

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

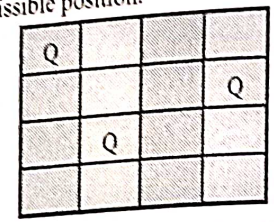
- Then place queen 2 after trying unsuccessful place - 1(1, 2), (2, 1), (2, 2) at (2, 3) i.e. 2nd row and 3rd column.



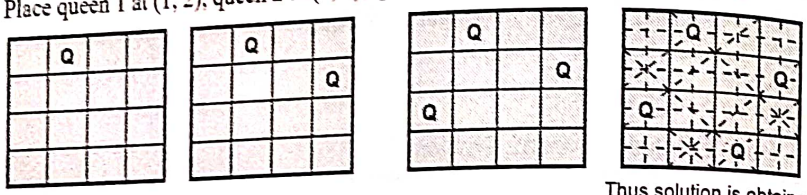
- This is the dead end because a 3rd queen cannot be placed in next column, as there is no acceptable position for queen 3. Hence algorithm **backtracks** and places the 2nd queen at (2, 4) position.



- The place 3rd queen at (3, 2) but it is again another dead end as next queen (4th 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).

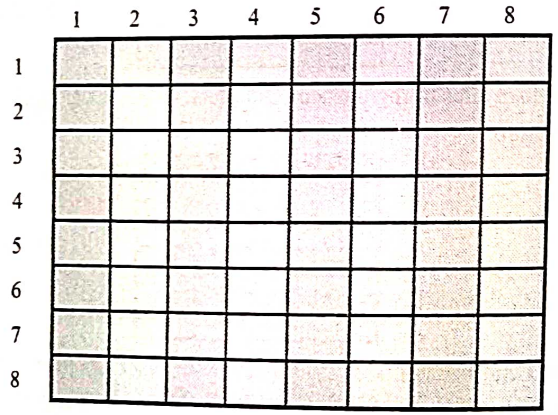


Thus solution is obtained.
(2, 4, 1, 3) in rowwise manner.

The state space tree of 4-queen's problem is shown in Fig. 6.4.1 [See Fig. 6.4.1 on next page].

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

- Initially the chessboard is empty.



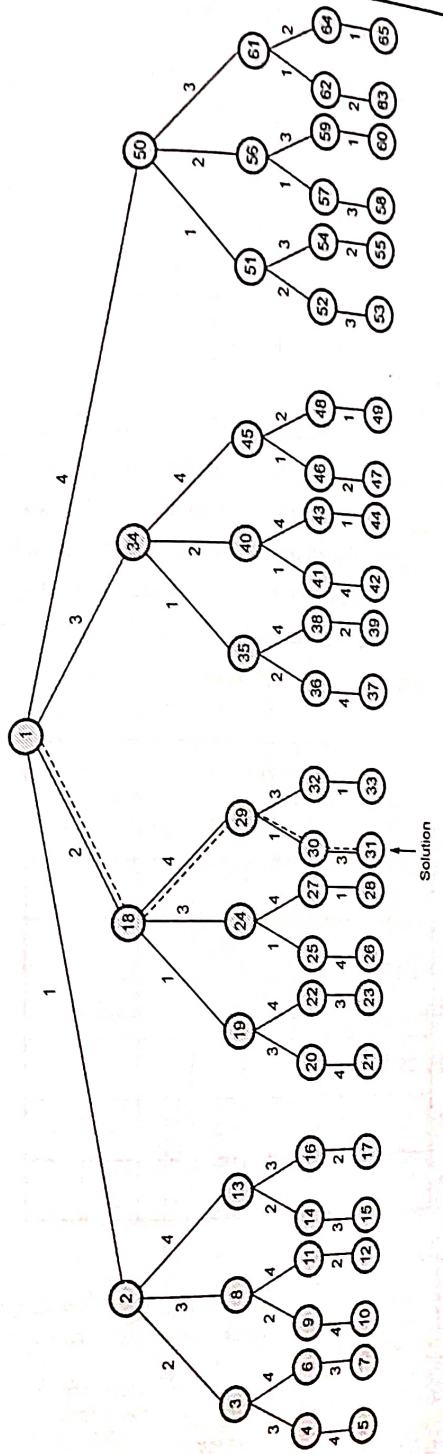
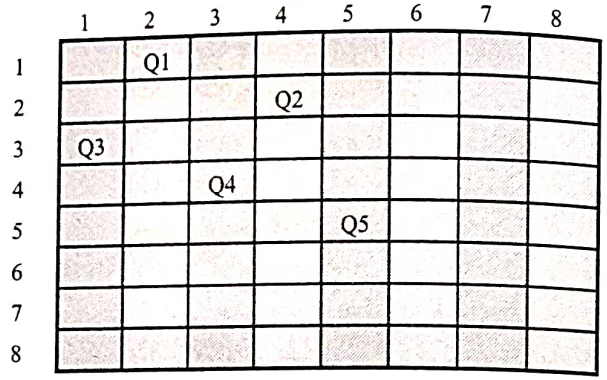
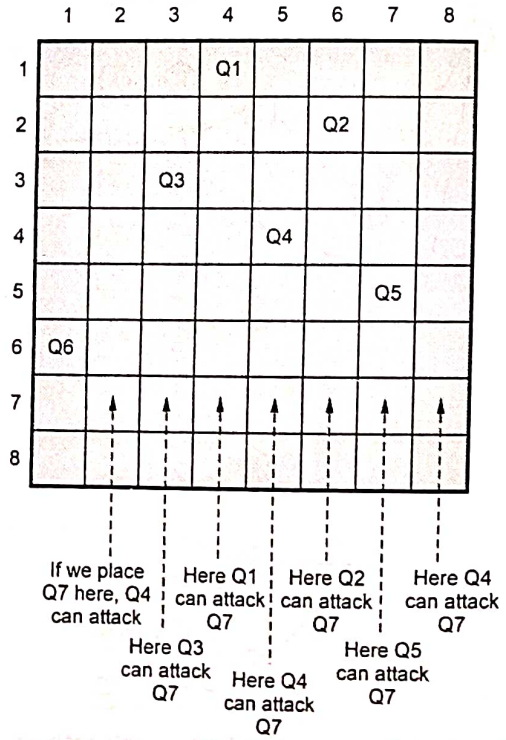


Fig. 6.4.1 State space tree for 4-queen's problem

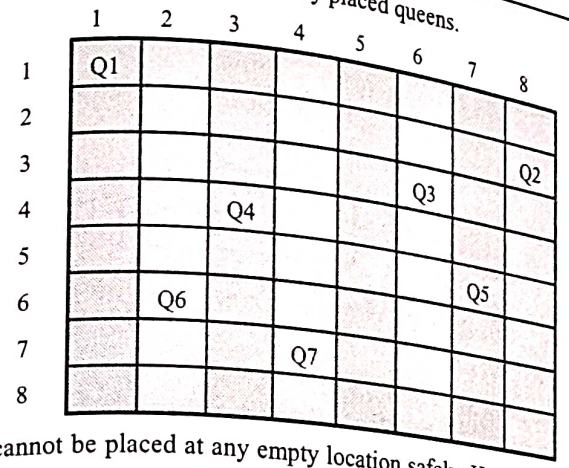
Now we will start placing the queens on the chessboard.



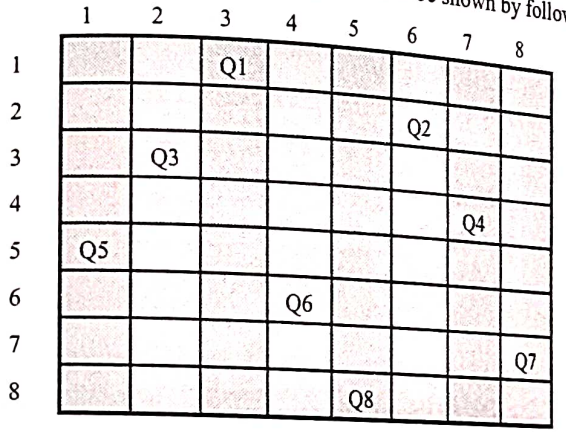
Thus we have placed 5 queens in such a way that no two queens can attack each other. Now if we want to place Q6 at location (6, 6) then queen Q5 can attack, if Q6 is placed at (6, 7) then Q1 can attack, if Q6 is placed at (6, 8) then Q2 can attack it. Similarly at (6, 5) the Q5 will attack it. At (6, 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.



6.4.2 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
```

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

```
            if(row=n)then//dead end
          print_board(n)
          //printing the board configuration
               else//try next queen with next position
                    Queen(row+1,n)
               }
          }
```

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

C Functions

```
/* By this function we try the next free slot
and check for proper positioning of queen
*/
void Queen(int row,int n)
{
          int column;
          for(column=1;column<=n;column++)
          {
               if(place(row,column))
               {
                board[row] = column; //no conflict so place queen
                if(row==n)   //dead end
```



```
                print_board(n);
          //printing the board configuration
          else  //try next queen with next position
               Queen(row+1,n);
          }
     }
}
int place(int row,int column)
{
     int i;
     for(i=1;i<=row – 1;i++)
     { //checking for column and diagonal conflicts
          if(board[i] == column)
               return 0;
          else
               if(abs(board[i] – column) == abs(i – row))
                    return 0;
     }
     //no conflicts hence Queen can be placed
     return 1;
}
```

Ex. 6.4.1 *Solve 8-queen's problem for a feasible sequence (6, 4, 7, 1).*

Sol.: As the feasible sequence is given, we will place the queens accordingly and then try out the other remaining places.

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

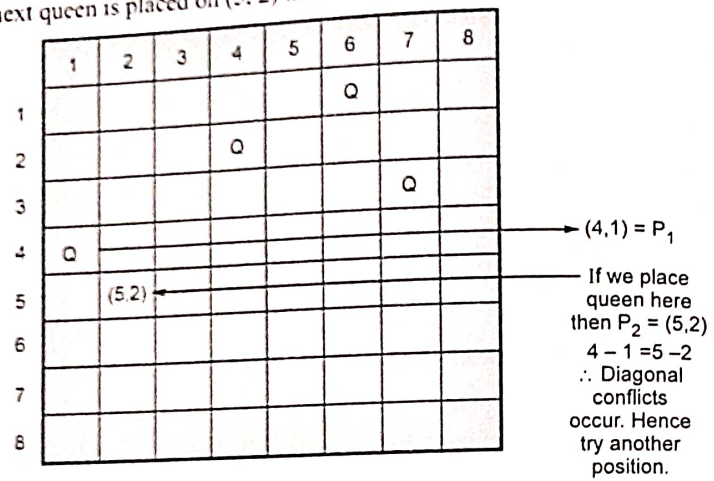
The diagonal conflicts can be checked by following formula -

Let, $P_1 = (i, j)$ and $P_2 = (k, l)$ are two positions. Then P_1 and P_2 are the positions that are on the same diagonal, if

$$i + j = k + l \quad \text{or}$$

$$i - j = k - l$$

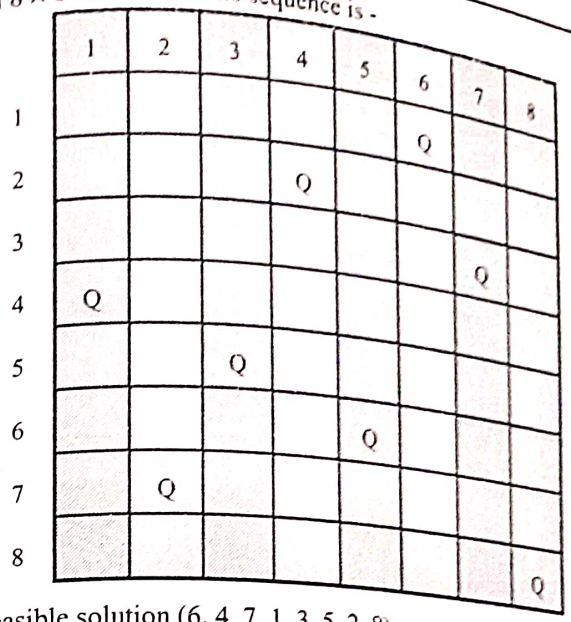
Now if next queen is placed on (5, 2) then



It can be summarized below.

Queen Positions								Action
1	2	3	4	5	6	7	8	
6	4	7	1					Start
6	4	7	1	2				As 4 – 1 = 5 – 2 conflict occurs.
6	4	7	1	3				5 – 3 ≠ 4 –1 or 5 + 3 ≠ 4 + 1 ∴ Feasible
6	4	7	1	3	2			As 5 + 3 = 6 + 2. It is not feasible.
6	4	7	1	3	5			Feasible
6	4	7	1	3	5	2		Feasible
6	4	7	1	3	5	2	8	List ends and we have got feasible sequence.

The 8-queens on 8×8 board with this sequence is -



8-queens with feasible solution (6, 4, 7, 1, 3, 5, 2, 8)

Ex. 6.4.2 *Draw the tree organization of the 4-queen's solution space. Number the nodes using depth first search.*

Sol.: The state space tree for 4-queen's problem consists of 4! leaf nodes. That means there are 24 leaf nodes. The solution space is defined by a path from root to leaf node.

The solution can be obtained for 4 queen's problem. For instance from 1 to leaf 31 represents one possible solution.

Ex. 6.4.3 *For a feasible sequence (7, 5, 3, 1) solve 8 queen's problem using backtracking.*

SPPU : May-14, Marks 10

Sol.: While placing the queen at next position we have to check whether the current position chosen is on the same diagonal of previous queen's position.

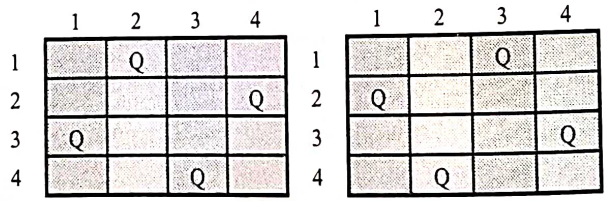
If $P_1 = (i, j)$ and $P_2 = (k, l)$ are two positions then P_1 and P_2 lie on the same diagonal if $i + j = k + l$ or $i - j = k - l$. Let us put the given feasible sequence and try out remaining positions.

(1), 2, (3), 4, (5), 6, (7), 8 → Already occupied position.

i values i.e. row values	j → 1	2	3	4	5	6	7	8	
	7	5	3	1					
	7	5	3	1	2				The remaining vacant position is 2. But 4 – 1 = 5 – 2. Not feasible.
	7	5	3	1	4				No conflict.
	7	5	3	1	4	6			(3, 3) = (6, 6) Conflict occurs.
	7	5	3	1	4	8			List ends and we can not place further queen's.
	7	5	3	1	4				Hence backtrack.
	7	5	3	1	6				The next free slot 6 is tried.
	7	5	3	1	6	2			Not feasible because (7, 1) = (2, 6) 1 + 7 = 6 + 2. That is on same diagonal.
	7	5	3	1	6	4			(1) 2 (3) (4) (5) (6) (7) 8 Occupied are circled.
	7	5	3	1	6	4	2		
	7	5	3	1	6	4	2	8	Conflict because (3, 3) = (8, 8)
	7	5	3	1	6	4	2		Hence backtrack.
	7	5	3	1	6	4	8		(6, 5) = (8, 7). Not feasible.
	7	5	3	1	6	4			Backtrack.
	7	5	3	1	6	8			
	7	5	3	1	6	8	2		
	7	5	3	1	6	8	2	4	List ends with solution of 8 queen's.

Ex. 6.4.4 *Obtain any two solutions to 4-queen's problem. Establish the relationship between them.*

Sol. : The solutions to 4-queen's problem are as given below.



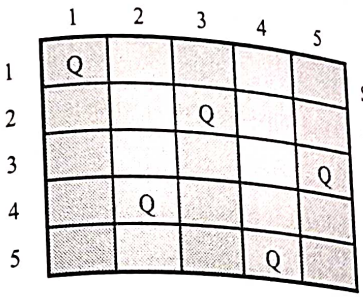
Solution 1 (2, 4, 1, 3)

Solution 2 (3, 1, 4, 2)

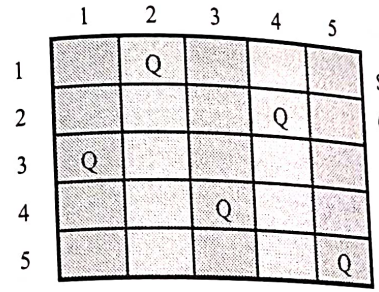
If these two solutions are observed then we can say that second solution can be simply obtained by reversing the first solution.

Ex. 6.4.5 *Generate at least 3 solutions for 5-queen's problem.* **SPPU : May-12, Marks 8**

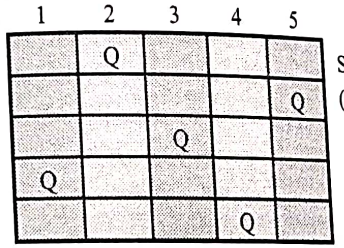
Sol. : The solutions are as given below.



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



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



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

Review Questions

Q.1 Analyze the 8-queen problem using backtracking strategy of problem solving. **SPPU : May-11, Marks 8**

Q.2 What is backtracking technique ? Find one solution for 4 - Queen's problem. Show all the steps and explain why you need to backtrack.

Q.3 State the principle of backtracking and write backtracking algorithm for N-Queen problem. **SPPU : May-19, End Sem, Marks 8**

6.5 Sum of Subsets

SPPU : May-10,11,15,17, 18, 19, Dec.-08,09,11,12,15, 18, Marks 12

Problem Statement

Let, S = $\{S_1, \ldots S_n\}$ be a set of n positive integers, then we have to find a subset whose sum is equal to given positive integer d.

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

$$S_1 \leq S_2 \leq \ldots \leq S_n$$

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 subset as 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 backtracking is possible then stop without solution. This problem can be well understood with some example.

Ex. 6.5.1 *Consider a set S = {5, 10, 12, 13, 15, 18} and d = 30. Solve it for obtaining sum of subset.* **SPPU - May-17, 18, End Sem, Dec. 18, End Sem, Marks 8**

Sol. :

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}

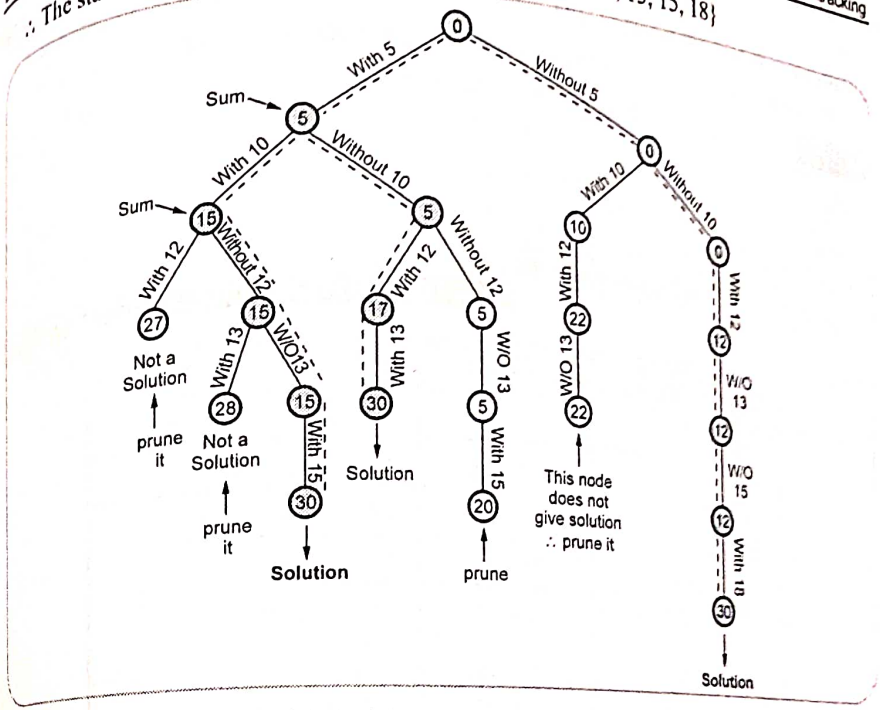


Fig. 6.5.1 State space tree for sum of subset

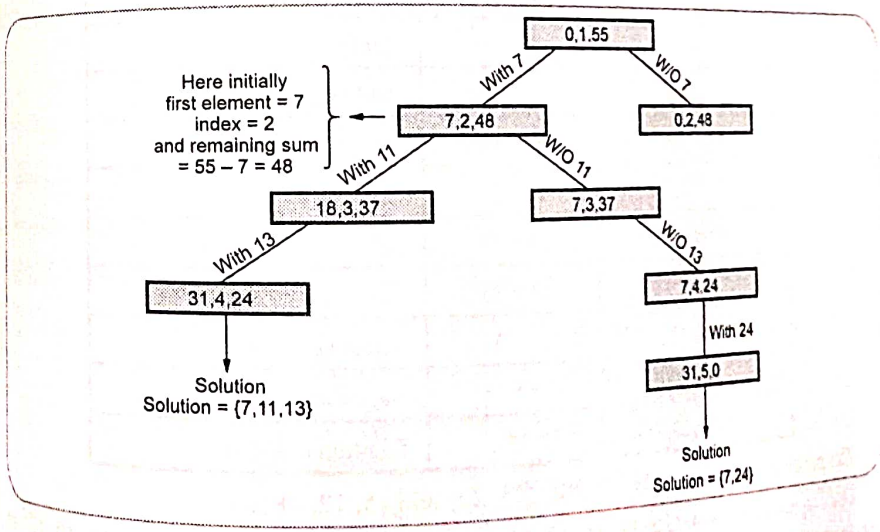


Fig. 6.5.2

Ex. 6.5.2 *Let m = 31 and W = {7, 11, 13, 24}. Draw a portion of state space tree for solving sum-of-subset problem for the above given algorithm.*

Sol. : Initially we pass (0, 1, 55) to sum-of-subset function. The sum = 0, index = 1 and remaining_sum = 55 initially {7 + 11 + 13 + 24}.

The recursive execution of the algorithm will be, as shown in Fig. 6.5.2.

Ex. 6.5.3 *Analyze sum of subset algorithm on data :*

M = 35 and

i) W = {5, 7, 10, 12, 15, 18, 20}

ii) W = {20, 18, 15, 12, 10, 7, 5}

iii) W = {15, 7, 20, 5, 18, 10, 12}

Are there any discernible differences in the computing time ?

SPPU : May-11, Marks 10, May 19, End Sem, Marks 5

Sol. :

Initially Subset = { }	Sum = 0	
5	5	Then add next element
5, 7	12 < 35	Add next element
5, 7, 10	22 < 35	Add next element
5, 7, 10, 12	34 < 35	Add next element
5, 7, 10, 12, 15	49 > 35	Sum exceeds ∴ Backtrack
5, 7, 10, 15	37 > 35	Backtrack
5, 7, 12	24 < 35	Select next element
5, 7, 12, 15	39 > 35	Backtrack
5, 10	15 < 35	Add next element
5, 10, 12	27< 35	Select next element
5, 10, 12, 15	42 > 35	Backtrack
5, 10, 15	30 < 35	Select next element
5, 10, 15, 18	48 > 35	Backtrack
5, 10, 18	33 < 35	Select next element
5, 10, 18, 20	53 > 35	Backtrack
5, 10, 20	35	Solution is found

The possible solutions are {5, 10, 20}, {15, 20} and {5, 12, 18}.

The sequence (ii) W = {20, 18, 15, 12, 10, 7, 5} and (iii) W = {15, 7, 20, 5, 18, 10, 12} are not in non-decreasing order. The (ii) sequence is in decreasing order and (iii) is totally unsorted.

For sum of subset problem, For efficient findings of solution the weights must be arranged in non-decreasing order. to show how this order is beneficial consider a scenario we have selected the elements {5, 7, 10, 12} which sums up 34. Now if we select the next elements {15, 20} then this sum will exceed the limit because 49 > 35.

So we will simply backtrack. It is not even necessary to examine the remaining elements of the list. If at all, the list is arranged according to decreasing order of the weights then we have to analyse all the elements of the given list. Similarly if the list is unsorted then each and every element needs to be analysed for required sum. Thus the sequences (ii) and (iii) makes the computing time worst.

Ex. 6.5.4 *What is backtracking method for algorithmic design ? Solve the sum of subset problem using backtracking algorithmic strategy for the following data N = 4 (w_1, w_2, w_3, w_4) = {11, 13, 24, 7} and M = 31.*

SPPU : Dec.-11, 12, Marks 12

Sol. :

Initially subset = { }	sum = 0	
11	13 < 31	Add next element
11, 13	24 < 31	Add next element
11, 13, 24	48 > 31	Backtrack
11, 13	24 < 31	Add 7
11, 13, 7	31	Solution is found i.e. {11, 13, 7}

The state space tree as shown in Fig. 6.5.3 -

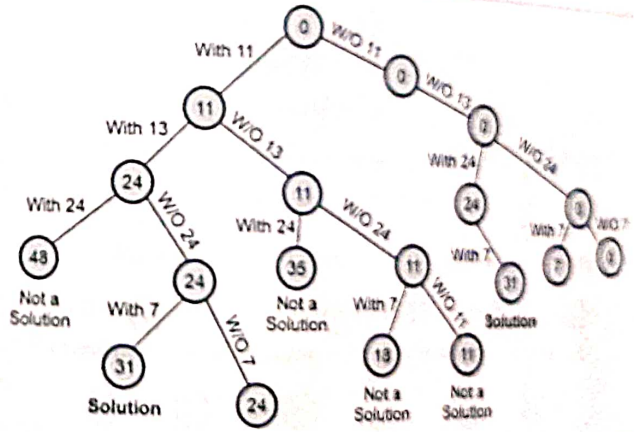


Fig. 6.5.3

Thus subsets {11, 13, 7} and {24, 7} are the solutions.

Ex. 6.5.5 *Draw a pruned state space tree for a given sum of subset problem. S = {3, 4, 5, 6} and d = 13*

Sol. : Fig. 6.5.4 state space tree the total sum is given inside each node. Pruned nodes has cross at its bottom.

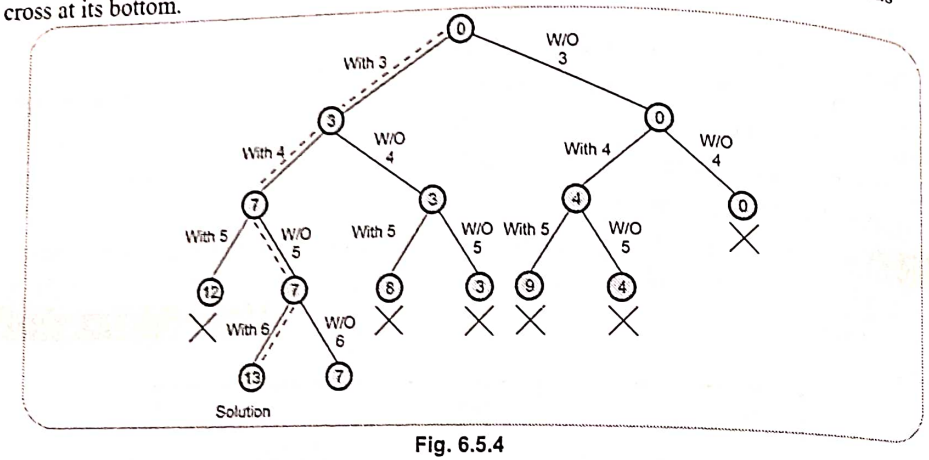


Fig. 6.5.4

Hence the solution is **{3, 4, 6}.**

Ex. 6.5.6 *Differentiate between backtracking and branch and bound. Draw state space tree for given sum of subset problem : Set of elements = { 3, 5, 6, 7 } and d = 15*

SPPU : Dec.-15, End Sem, Marks 8

Sol : Difference : (Refer section 7.1.1)

Following is a state space tree in which total sum is represented inside each node. Pruned node has a cross at its bottom. (Fig. 6.5.5)

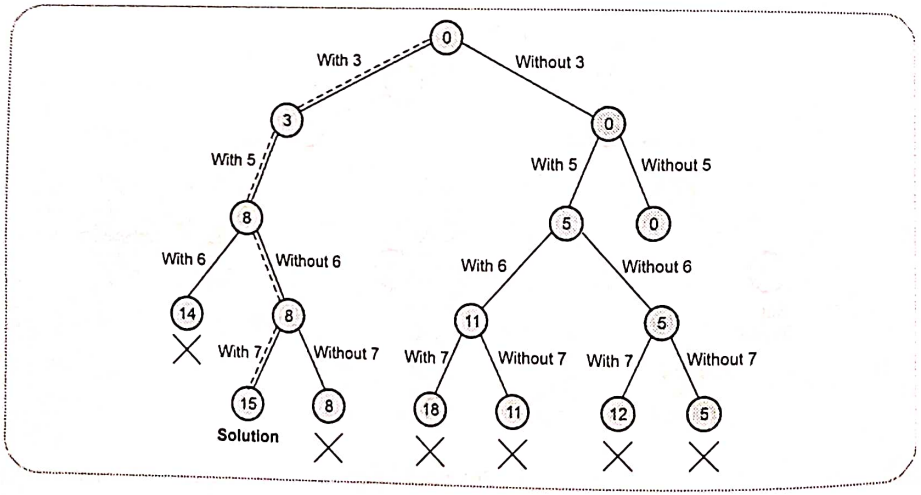


Fig. 6.5.5 State space tree

The solution is = **{3, 5, 7}**

Strategy to Solve Sum of Subsets Problem

1. Choose each number and sum it up. Compare it with the given sum d.
2. If the sum of chosen numbers is less than d then add the next subsequent element.
3. If the sum of chosen numbers is greater than d then remove one element and try some another number in the set.
4. Thus all the permutations of the numbers for summing up should be attempted.
5. When we get the sum=d then declare the chosen set of numbers as the solution.

Review Question

Q.1 Write a recursive backtracking algorithm for sum of subsets of problem.

SPPU : Dec.-08, 09, May-10, May-15, End Sem, Marks 8

6.6 Graph Coloring

SPPU : May-08,10,11,17, Dec.-12,13, Marks 12

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 given in Fig. 6.6.1.

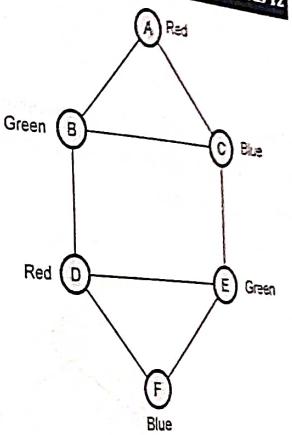
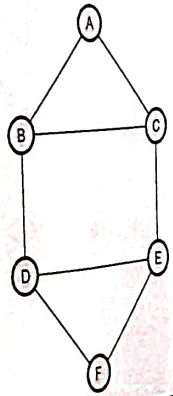


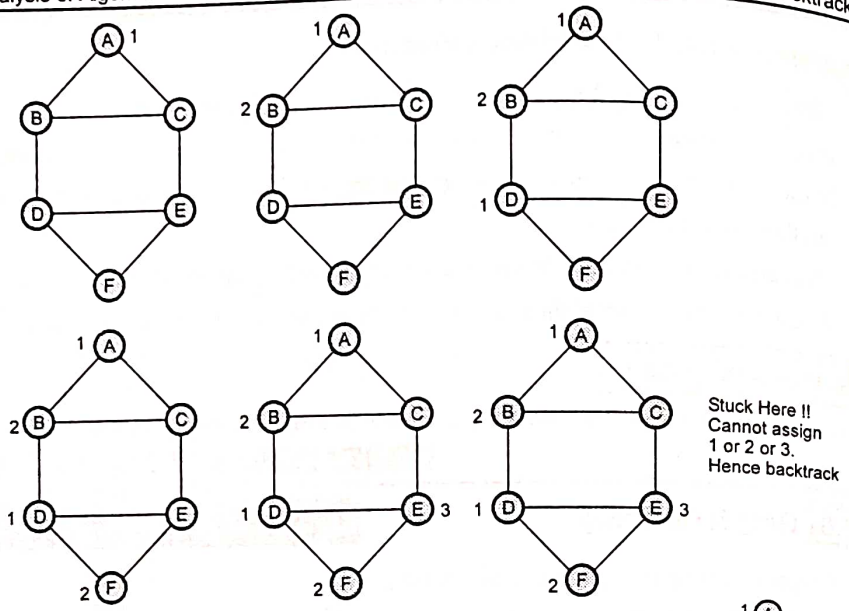
Fig. 6.6.1 Graph and its coloring

As given in Fig. 6.6.1 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 -

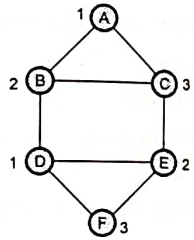


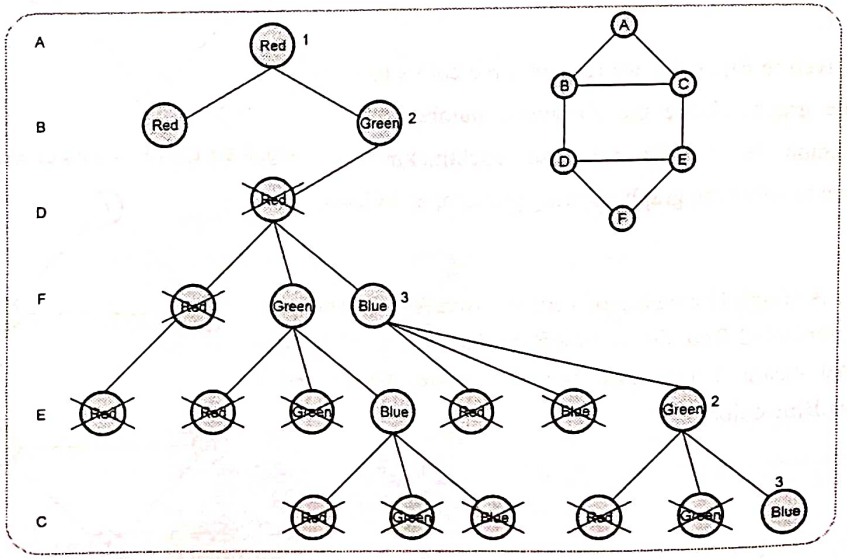
Step 1 : A Graph G consists of vertices from A to F. There are three colors used Red, Green and Blue. We will number them out. That means 1 indicates Red, 2 indicates Green and 3 indicates Blue color.

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.

6.6.1 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);        Takes O(nm) time

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);
}
```

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);
}
```

6.6.2 Analysis

The Gr_color takes computing time of $\sum_{i=0}^{n-1} m^i$. Hence total computing time for this algorithm is $O(nm^n)$.

If we trace the above algorithm then we can assign distinct colors to adjacent vertices. For example : Consider, a graph given below can be solved using three colors.

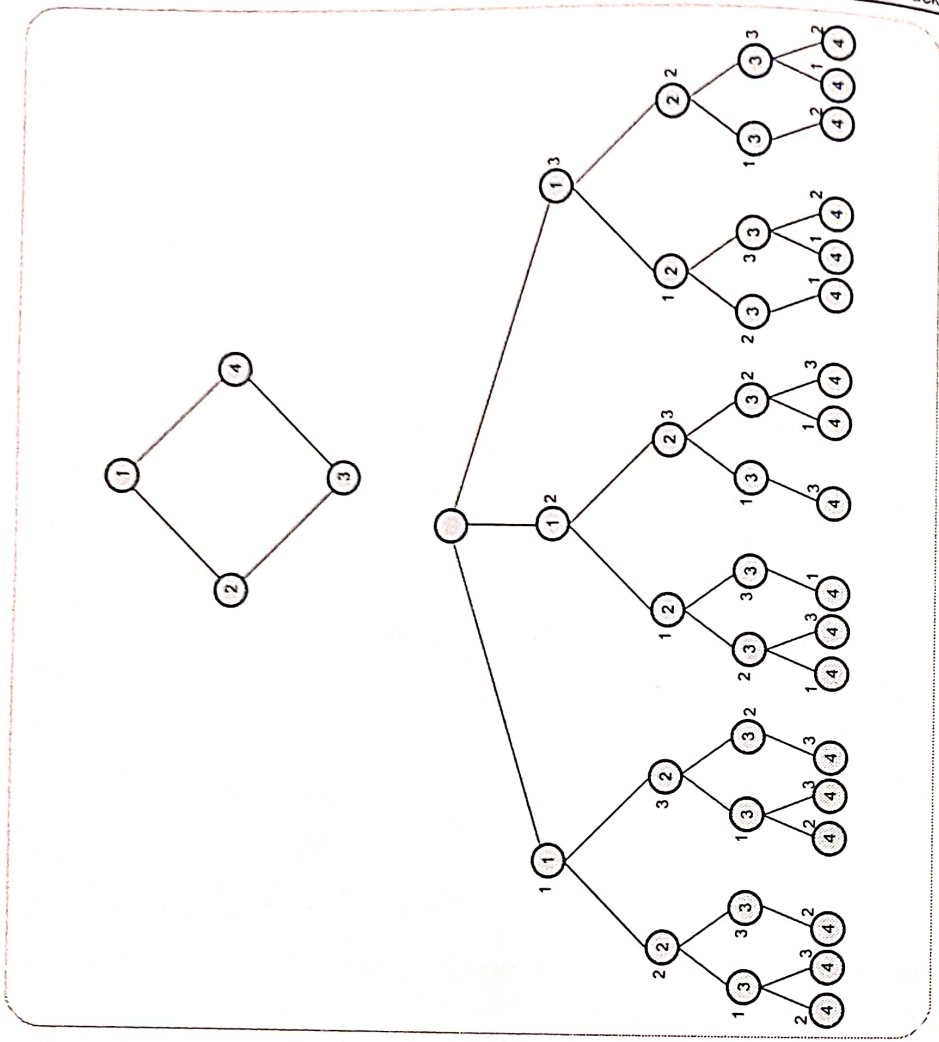
Consider,

Red = 1, Green = 2 and Blue = 3.

The number inside the node indicates vertex number and the number outside the node indicates color index.

C Function

```
void Gen_Col_Value(int k,int n)
{
  int j;
  int a,b;
  while(1)
  {
  a=color_tab[k]+1;
  b=m+1;
  color_tab[k] = a%b;       // next highest color
  if(color_tab[k]==0) return;    // all colors have been used
  for(j=1;j<=n;j++)
  {
  // check if this color is distinct from adjacent colors
  if(G[k][j] && color_tab[k]==color_tab[j])
   break;
  }
  if(j==n+1) return;         // next new color found
 }
}
// such that adjacent vertices are assigned distinct integers
// k is the index of next vertex color.
void Gr_coloring(int k,int n)
{
   Gen_Col_Value(k,n);
   if(color_tab[k]==0) return;   // No new color available
if(k==n) return;     // at most m colors have been
   else Gr_coloring(k+1,n);  // used to color the n vertices
}
```



Strategy to solve Graph Coloring Problem

1. Choose some color say c and apply to some vertex say v.
2. Go on choosing different colors that are available from the list and apply to each successive vertex. After applying the color just check whether it is same with the adjacent color or not.
3. If the chosen color is not same to the adjacent colors then go to the next adjacent vertex otherwise try out some another color from the list or change the already assigned colors.
4. Repeat this procedure for coloring all the vertices.

Ex. 6.6.1 *Construct plannar graph for following map. Explain how to find m-coloring of this plannar graph by using m-coloring backtacking algorithm.* **SPPU : May-11, Marks 10, May-17, End Sem. Marks 8**

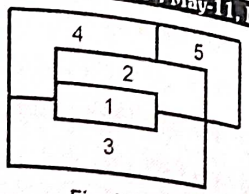


Fig. 6.6.2

Sol. : Refer section 6.6.1 for m-coloring of plannar.

Steps for graph coloring -

1. Number out each vertex (V0, V1, V2,...Vn-1)
2. Number out the available colors (C0,C1,C2,...Cn-1)
3. Start assigning Ci to each Vi. While assigning the colors note that two adjacent vertices should not be colored by the same color. And least number of colors should be used.
4. While assigning the appropriate color, just backtrack and change the color if two adjacent colors are getting the same.

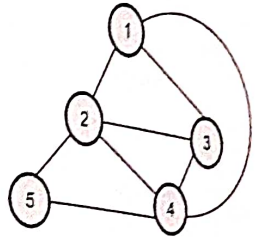


Fig. 6.6.3

Review Questions

Q.1 Write recursive backtracking schema for m-coloring of the graph. Determine the time complexity of the same. **SPPU : May-08, 10, Marks 12**

Q.2 Discuss and analyze the problem of graph coloring using backtracking. **SPPU : Dec.-12, Marks 6**

Q.3 Explain graph coloring problem. **SPPU : Dec.-13, Marks 8**

6.7 The 0/1 Knapsack Problem

SPPU : Dec.-11, May-12,14,15, 18, Marks 16

In this section we will discuss "How Knapsack problem be solved using backtracking". The Knapsack problem can be stated as follows.

Problem statement : "If we are given n objects and a Knapsack or a bag in which the object i with weight is to be placed. The Knapsack has a capacity m. Then the profit value that can be earned is p_i. Then **objective** is to obtain filling up of Knapsack with maximum profit earned, but it should not exceed the capacity m of 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 Knapsack is filled up with selected objects. Note that the Knapsack's capacity m 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.

Hence we must arrange the given data in non-increasing order i.e.

$$p_i / w_i \geq p_{i+1} / w_{i+1}$$

- 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.

$$ub + (1 - (c - m) / w_i) * p_i$$

The algorithm for computing an upper bound is as given below -

```
1   Algorithm Bound_Calculation (cp,cw,k)
2   //Problem description : This algorithm
3   //calculates the upper bound, for each item
4   //Input : cp is the current profit, cw is the
5   //current weight and k is the index of just
6   //removed item
7   //Output : The upper bound value can be returned
8   ub ← cp
9   c ← cw
10    for (i ← k+1 to n) do
11  {
12      c ← c+w[i]
13      if (c < m) then              //m is capacity of knapsack
14        ub ← ub + p[i]
15      else
16        {
17          ub ← ub + (1-(c-m)/w[i])*p[i]
18          return ub
19        }
20      return ub
21  }
```

This function is invoked by a Knapsack function **Bk (k, cp, cw)**. The algorithm for the same is as given below.

Algorithm Bk (k,cp,cw)

```
1   //Problem description : This algorithm is to obtain
2   //solution for knapsack problem. Using this algorithm
3   //a state space tree can be generated.
```



```
4   //Input : Initially k=1, cp=0 and cw=0. The
5   //k represents the index of currently referred
6   //item. The cp and cw represent the profit and
7   //weights of items, so far selected.
8   //Output : The set of selected items that are
9   //satisfying the objective of knapsack problem.
10
11  if (cw+w[i]<= m) then          <- Generates left child
12  {
13    temp[k] ← 1                  //temp array stores currently selected object
14    if (k<n) then
15  Bk(k+1,cp+p[k],cw+w[k])        //recursive call
16  if ((cp+p[k] > final_profit) AND (k==n)) then
17  {//final_profit is initially -1, it represents final profit
18  final_profit ← cp + p[k]
19    final_wt ← cw + w[k]
20    for (j ← 1 to k) then        <- Finally get the solution
21      x[j] ← temp[j]
22  }
23  if (Bound_calculation (cp,cw,k)  final_profit) then   <- Generates right child
24  {
25      temp[k] ← 0
26      if (k < n) then            <- Invoke this function in order to obtain upper bound
27          Bk(k+1, cp, cw)
28      if((cp > final_profit) AND (k=n)) then
29      {
30          final_profit ← cp
31          final_wt ← cw
32          for(j ← 1 to k)        <- Finally get the solution
33              x[j] ← temp [j]
34      }
35  }
```

Ex. 6.7.1 *Obtain the optimal solution to the Knapsack problem n = 3, m = 20 (p_1, p_2, p_3) = (25, 24, 15) and (w_1, w_2, w_3) = (18, 15, 10).* **SPPU : May-14, Marks 10**

Sol. : Given that n = 3 and m = Capacity of Knapsack = 20.

We have

i	p [i]	w [i]	p [i] / w [i]
1	25	18	1.3
2	24	15	1.6
3	15	10	1.5

As we want p[i] / w[i] ≥ p[i+1] / w [i+1], let us rearrange the items as

i	p [i]	w [i]	p [i] / w [i]
1	24	15	1.6
2	15	10	1.5
3	25	18	1.3

Step 1 : Here we will trace knapsack backtracking algorithm using above values.

Initially set **final_profit = – 1**

Bk (1, 0, 0)

∴ k = 1

cp = 0

cw = 0

Check if (cw + w [k] < = m)

i.e. if (0 + w [1] = 0 + 15 <= 20) → yes

∴ set temp [1] = 1

if (k < n) i.e. if (1 < 3) ? → yes

∴ Bk (k + 1, cp + p [k], cw + w [k]) is called

Hence Bk (2, 0 + 24, 0 + 15) will be called. Refer line 14 of **Algorithm Bk ( ).**

Step 2 :

k = 2

cp = 24

cw = 15

Check if (cw + w [k] <= m)

i.e. if (15 + w [2] i.e. 15 + 10 <= 20) → no

Hence we will calculate ub i.e. upper bound.

Step 3 : For calculating upper bound, initially set

ub = cp = 24

c = cw = 15 Refer line 8 and 9 of Algorithm Bound_calculation

for (i = k+1 to n) we will update upper bound value.

Consider i = k+1 = 3

c = c + w [i]

c = 15 + w [3]

c = 15 + 18 = 33

∴ As 33 > 20 i.e. exceeding the Knapsack capacity, we will compute ub as

ub = ub + (1 – (c = m) / w [i]) * p [i] ← See line 17 of Bound_calculation

ub = 24 (1 – (15 – 20) / w [3] * p[3]

∴ = 24 + (1 – 5 / 18) * 25

ub = 30.94 ≈ 31

As (ub > final_profit) i.e. 30.94 > – 1

Set temp [2] = 0

Now refer line 27 of **Bk** algorithm, a recursive call with k = 3 is made.

Step 4 :

k = 3

cp = 24 + p [2] = 24 + 15 = 39

cw = 15 + w [2] = 15 + 10 = 25

Check if (cw + w [k] <= m)

i.e. if (25 + w [3] = 25 + 18 < = 20) → no

Hence we will calculate upper bound.

Step 5 : For calculating upper bound, initially set

ub = cp = 39 } Refer line 8 and 9 of algorithm **Bound_calculation**

c = cw = 25 }

Set i = k + 1 i.e. 3 + 1 = 4 But i = 4 > n

Hence we will keep ub as it is

∴ ub = 39

As (39 > final_profit) i.e. 39 > – 1

Set temp [k] = temp [3] = 0

Now k = n = 3 and cp = 39 > final_profit (i.e. – 1)

final_profit = cp = 39 } Refer line 30 and 31
final_wt = cw = 25 } of algorithm **Bk**

Now copy all the contents of temp array to an array x [ ]. Refer line 32, 33 of **Algorithm Bk.**

Hence **x = {1, 0, 0}** i.e. item 1 with **weight = 15** and **profit = 24** is selected.

The state space tree can be drawn as

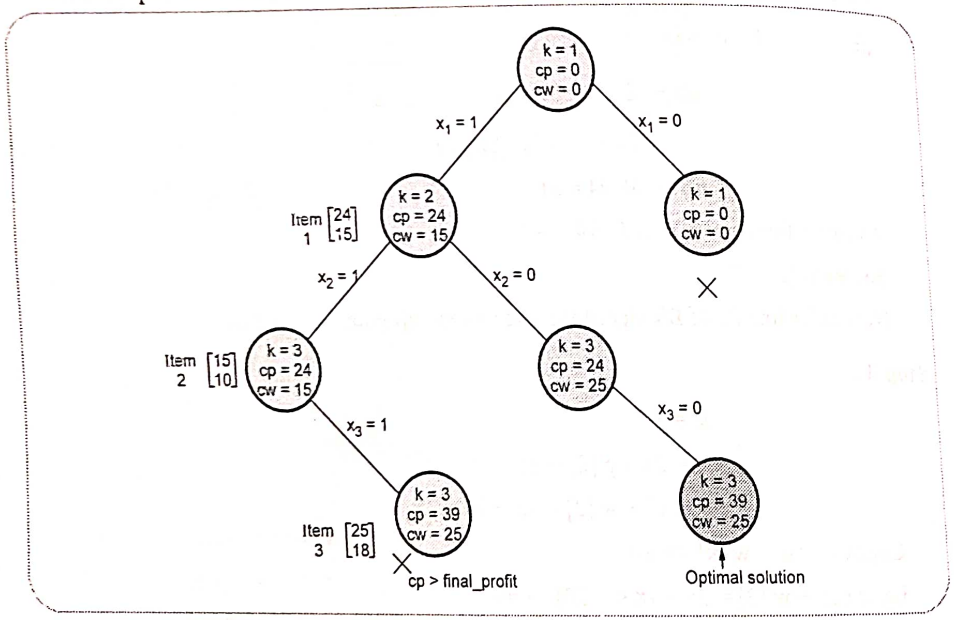


Fig. 6.7.1 State space tree for Knapsack

Here the left branch indicates inclusion of item and right branch indicates exclusion of items. Hence $x_1 = 1$ means 1st item selected and $x_2 = 0$, $x_3 = 0$ means 2nd and 3rd items are not selected. The state space tree can be created in DFS manner.

C Function

```
Bkar Bound_Calculation(int cp,int cw,int k)
{
 int ub,c,i;
 ub=cp;c=cw;
 for(i=k+1;i<=n;i++)
 {
     c=c+w[i];
     if(c<m)
         ub=ub+p[i];

     else
         return(ub+(1-(c-m)/w[i])*p[i]);
 }
 return ub;
}
void BK(int k,int cp,int cw)
{
   int new_k,new_cp,new_cw,j;
   //Generate left child
   if(cw+w[k]<=m)
   {
       temp[k]=1;
       if(k<n)
       {
           new_k=k+1;
           new_cp=cp+p[k];
           new_cw=cw+w[k];
           BK(new_k,new_cp,new_cw);
       }
       if((new_cp>final_profit)&&(k==n))
       {
        final_profit=new_cp;
        final_wt=new_cw;
        for(j=1;j<=k;j++)
            x[j]=temp[j];
```

```
        }
    }
    //Generate right child
    if(Bound_Calculation(cp,cw,k)>=final_profit)
    {
        temp[k]=0;
        if(k<n)
            BK(k+1,cp,cw);
        if((cp>final_profit)&&(k==n))
        {
            final_profit=cp;
            final_wt=cw;
            for(j=1;j<=n;j++)
            x[j]=temp[j];
        }
    }
}
```

Ex. 6.7.2 *Consider the following instance for knapsack problem using backtracking n = 8, P = (11, 21, 31, 33, 43, 53, 55, 65) W = (1, 11, 21, 23, 33, 43, 45, 55) M = 110.*

SPPU : Dec.-11, Marks 6, May-12, Marks 16

Sol. : We will arrange all the items in $\frac{P_i}{W_i} > \frac{P_{i+1}}{W_{i+3}} > \frac{P_{i+2}}{W_{i+3}} \ldots$

The instance will be

Item	P_i	W_i	P_i / W_i
1	11	1	11
2	21	11	1.90
3	31	21	1.47
4	33	23	1.43
5	43	33	1.30
6	53	43	1.23
7	55	45	1.22
8	65	55	1.18

Let $M = 110$

Initially total-profit $= -1$

$cp = 0$, $cw = 0$ We use formula as

$cp = cp + p[i]$, $cw = cw + w[i]$ where i represents item number.

$cp = cp + 11 = 0 + 11 = 11$

$cw = cw + 1 = 0 + 1 = 1 < m$

$p[i] = 21$

$i = 1$, $w[i] = 11$ ∴ select item1

$cp = 11 + 21 = 32$

∴ $cw = 1 + 11 = 12 < 110$

$p[3] = 31$

$i = 2$ $w[3] = 21$ ∴ select item 2

$cp = 32 + 31 = 63$

∴ $cw = 12 + 21 = 33 < 110$

$p[4] = 33$

$i = 3$ $w[4] = 23$ ∴ select item 3

$cp = 63 + 33 = 96$

$cw = 33 + 23 = 56 < 110$

$p[5] = 43$

$i = 4$ $w[5] = 33$ ∴ select item 4

∴ $cp = 96 + 43 = 139$

$cw = 33 + 56 = 89 < 110$

$i = 5$ $p[6] = 53$ ∴ select item 5

$w[6] = 43$

∴ $cp = 139 + 53 = 192$

$cw = 99 + 43 = 142 > 110$ ∴ not to select item 6

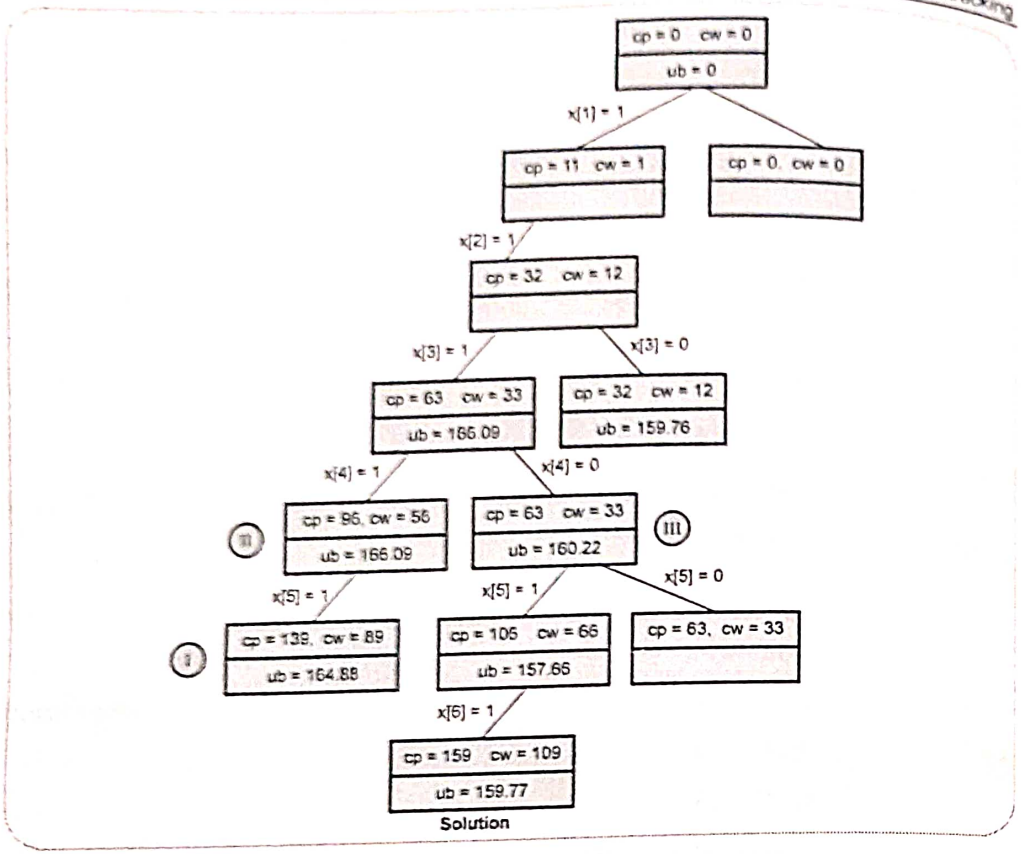


Fig. 6.7.2

Computation at node I

$$ub = cp + \left(\frac{m - cw}{W_{i+1}}\right) * P_{i+1}$$

$$ub = 11 + 21 + 31 + 33 + 43 + \left(\frac{110 - 89}{43}\right) * 53$$

$$ub = 164.88$$

Computation at node II

$$ub = 11 + 21 + 33 + \left(\frac{110 - 56}{33}\right) * 43$$

$$= 166.09$$

Computation at node III

$$ub = 11 + 21 + 31 + 43 + 53 + \left(\frac{110 - 109}{45}\right) * 55$$

$$ub = 160.22$$

Thus upper bound will be,

$$11 + 21 + 31 + 33 + 43 + \left(\frac{110 - 89}{43}\right) * 53 = 164.88$$ on selecting item 6, 7, 8 the total weight will exceed the capacity hence we will back track at item 4.

That means item 1, item 2, item 3, item 4 are selected. Thus upper bound will be calculated and a space tree is constructed, as shown in Fig. 6.8.2.

The solution giving maximum profit will be by selection of item 1, item 2, item 3, item 5 and item 6.

Review Question

Q.1 Write an algorithm for 0/1 knapsack problem using backtracking method. **SPPU : May-15, 18, End Sem. Marks 8**

6.8 Multiple Choice Questions

Q.1 A backtracking algorithm one problem instance might generate only ____.

a O(n) nodes b (n) nodes

c n! nodes d 2^n nodes

Q.2 In n-queen's problem when n=4 then how many leaf nodes exist ?

a 16 b 64 c 24 d 32

Q.3 _____ are those solutions state s for which the path from root to s defines the tuple that is a member of the set of solutions of the problem

a Problem state b Answer state

c Live nodes d None of the above

Q.4 Following is a solution for 5 queen's problem

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

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

Q.5 Following data structure is used for generating the solution using backtracking

a Binary Tree b Heap c State Space Tree d B-Tree

Q. 6 The live node whose children are currently being expanded is called _____.

a dead node b live node

c E node d current node

Q. 7 The number of colors used for graph coloring equal to _____.

a total number of vertices b total number of edges

c half the number of total vertices d degree of graph plus one

Q. 8 The graph coloring problem can be efficiently solved using _____.

a dynamic programming b greedy algorithm

c backtracking d divide and conquer

Q. 9 A _____ is a generated node all of whose children have been generated.

a dead node b live node

c E node d None of the above

Q. 10 Backtracking needs _____ for implementation.

a stack b queue c priority Queue d heap

Answer Keys for Multiple Choice Questions :

Q.1	a	Q.2	c	Q.3	b	Q.4	d	Q.5	c
Q.6	c	Q.7	d	Q.8	c	Q.9	a	Q.10	a

❑❑❑