11

Limitations of Algorithm Power

11.1 Introduction

So far now we have studied many algorithms. we have seen how these algorithms play an important role in solving a variety of problems. But it is not possible that every problem could be solved by some algorithm. That means there are certain problems that could not be solved using any algorithm. In other words, we can say that power of algorithm is limited to some extent. It is seen that power of algorithm is limited because of following reasons -

1. There are some problems which can not be solved with algorithms.
2. There are some problems that can be solved by algorithm but such solving is not within polynomial time.
3. There are some problems that can be solved in polynomial time but there are lower bounds for efficiency of these algorithms.

In this chapter we will discuss about limitations of algorithm power. First of all we will understand of concept of **lower bound** and various methods for establishing lower bounds. Then we will get introduced with the concept of **decision trees** by which the performance of different algorithms can be studied. Finally, we will end up with the discussion on **P, NP and NP-Complete** problems.

11.2 Lower-Bound Arguments

In this section we will first understand "**how to obtain lower bounds?**" Basically obtaining lower bounds means : *estimating minimum amount of work needed to solve the problem.* When we try to obtain lower bounds of the algorithm we basically look for limits of efficiency of any algorithm that may solve the problem. Such a problem may be known or unknown

For example : Following are the examples in which lower bounds of corresponding algorithm can be obtained -

1. Number of comparisions needed to sort an array of size n.
2. Number of comparisions that are required to search a desired element from a sorted array.
3. Number of additions needed to add two n x n matrices.

To obtain efficiency of particular algorithm there are two ways -

Step 1 : First establish the asymptotic efficiency class for the given problem.

Step 2 : Then check where the class of given problem fits in the hierarchy of efficiency classes. In other words, whether the problem lies in linear, quadric, logarithmic or exponential category of efficiency class, is to be checked.

For instance : Consider the problem of sorting algorithms, if we choose insertion sort method for sorting whose efficiency is quadratic (i.e. n^2), then we go for various sorting algorithms and find that there are even more faster algorithms (eg. quick sort, heap sort) with the efficiency of O (nlogn) are available. In comparision to O (nlogn) efficiency class insertion sort is proved to be slower algorithm.

Whenever we try to find an efficiency of an algorithm it is better to compare this algorithm with other algorithms that are used to solve same type of problems. For example if we want to decide the efficiency of insertion sort it is better to compare insertion sort with other sorting methods. However, if we compare insertion sort with efficiency to Tower of Hanoi problem them we can not determine the efficiency of insertion sort. This is basically because these two are different types of problems and hence their algorithms need not be compared.

There is another issue while determining the lower bounds of an algorithm. IT is desirable to know the best possible efficiency of any algorithm solving the given problem. By this basically we are looking for a chance of improvement of that algorithm. **Tight** lower bound means there exists an algorithm with the same efficiency as the lower bound. If there is a gap between fastest algorithm and best lower bound obtained then the algorithm of corresponding problem can be improved.

For example : In binary search algorithm the best lower bound is O(logn) and the fastest searching algorithm has an efficiency of O(logn). That means there exists **tightness.** And there is no need to improve binary search algorithm. On the other hand, Consider insertion sort method whose lower bound efficiency is $O(n^2)$, but still faster algorithms with O(nlogn) efficiency are available. This implies that insertion sort can be further improved in order to make it more efficient.

There are **Various methods** of establishing lower bound -

1. Trivial Lower Bounds
2. Information-Theoretic Arguments

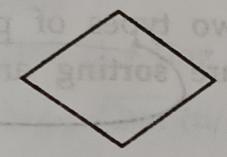
11.3 Decision Trees

All the sorting and searching algorithms are based on comparision method. The comparision is usually made on input items. A model is prepared to study the performance of such algorithms which is called **Decision Tree**. Hence any comparision-based sorting algorithm can be represented by a decision tree. In decision tree

- Internal nodes represent the comparisions made between input items.
- External nodes or leaf nodes represent the results of the algorithm.

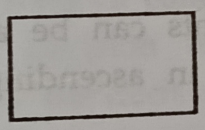
There may be a case that we get more number of leaf nodes than actual outputs (results.) This because, we can obtain results in decision trees through a different chain of comparisions.

For example : Consider a decision tree drawn for finding largest number among the three variables x, y and z. The simple graphical notations used in decision trees are



Condition

Transition



Outcome/ result

Fig. 11.1

Let us now draw a decision tree for the above mentioned problem -

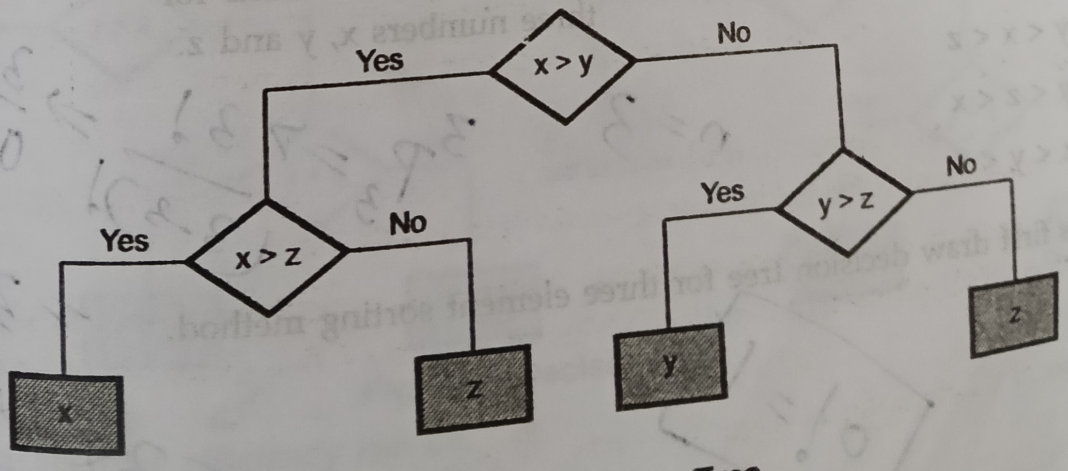


Fig. 11.2 Decision Tree

From above drawn decision tree following observations can be made -

Let, h be the height of a tree and

n be the total number of leaf nodes

Then

$$h \geq \lceil \log n \rceil$$

{Note $\lceil \log n \rceil$ indicates the rounded up value i.e. ceil function }

In other words we can also state that

$$2^h \geq n$$

Due to the inequality indicated by above equation the lower bound on height of the binary decision tree can be obtained. And we get such a lower bound using the information about the output of algorithm. Hence this bound is callled the **information theoretic lower bound.** There are two types of problems that can be usually solved using **decision trees** and those are **sorting** and **searching**. Let us discuss them in following subsections -

11.3.1 Decision Trees for Sorting Algorithms

we can derive the lower bounds from decision trees for sorting algorithm. Consider the example of binary insertion sort. In this method we will compare three elements and based on their values those elements can be arranged in ascending order. Now for arranging three different values in ascending order there can be various cases -

1. $x < y < z$
2. $x < z < y$
3. $z < x < y$
4. $y < x < z$
5. $y < z < x$
6. $z < y < x$

$\Rightarrow$

These different cases of three different numbers represent permutations for three numbers x, y and z.

$n = 3$

$${}^3P_3 = \frac{3!}{(3-3)!} \Rightarrow \frac{3!}{0!}$$

$$= 3!$$

$$= 3 \times 2 \times 1$$

$$= 6$$

$0! = 1$

Let us first draw decision tree for three element sorting method.

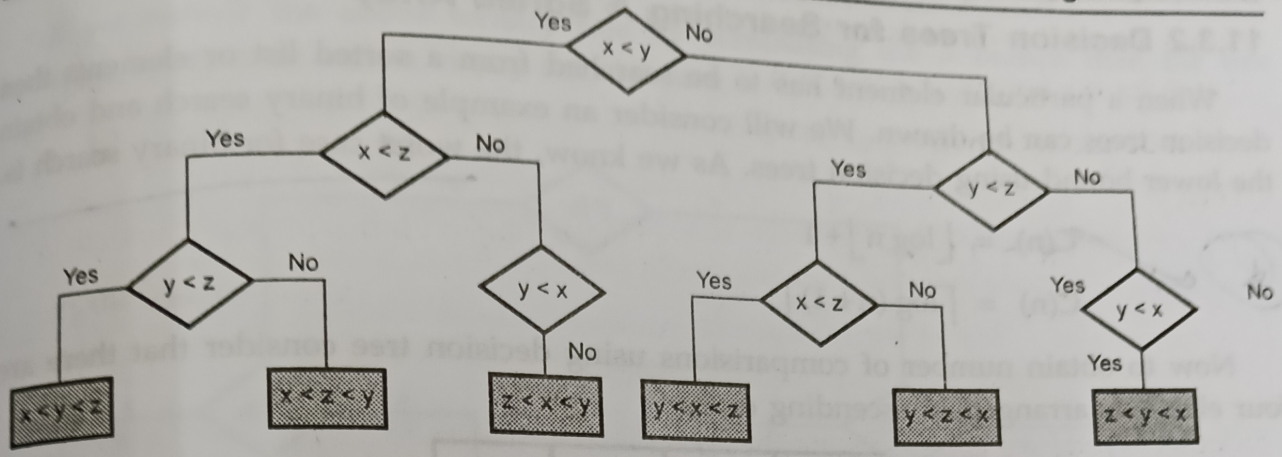


Fig. 11.3 Decision tree for 3 element insertion sort

From this tree the worst case number of comparisions can be obtained as -

$$C(n) \geq \lceil \log n! \rceil$$

Using stirling's formula

$$\lceil \log n! \rceil = (\log 1) + (\log 2) + \ldots + (\log n)$$

$$\therefore \quad \lceil \log n! \rceil = \log_2 \sqrt{2\pi n}\,(n/e)^n$$

$$= n \log_2 n - n \log_2 e + \frac{\log_2 n}{2} + \frac{\log 2\pi}{2}$$

$$\approx n \log_2 n$$

That means at least $n \log_2 n$ comparisions are necessary to sort an arbitrary list of n elements. Using the decision tree only, we can compute average number of comparisions Let us draw the decision tree for insertion sort of 3 elements

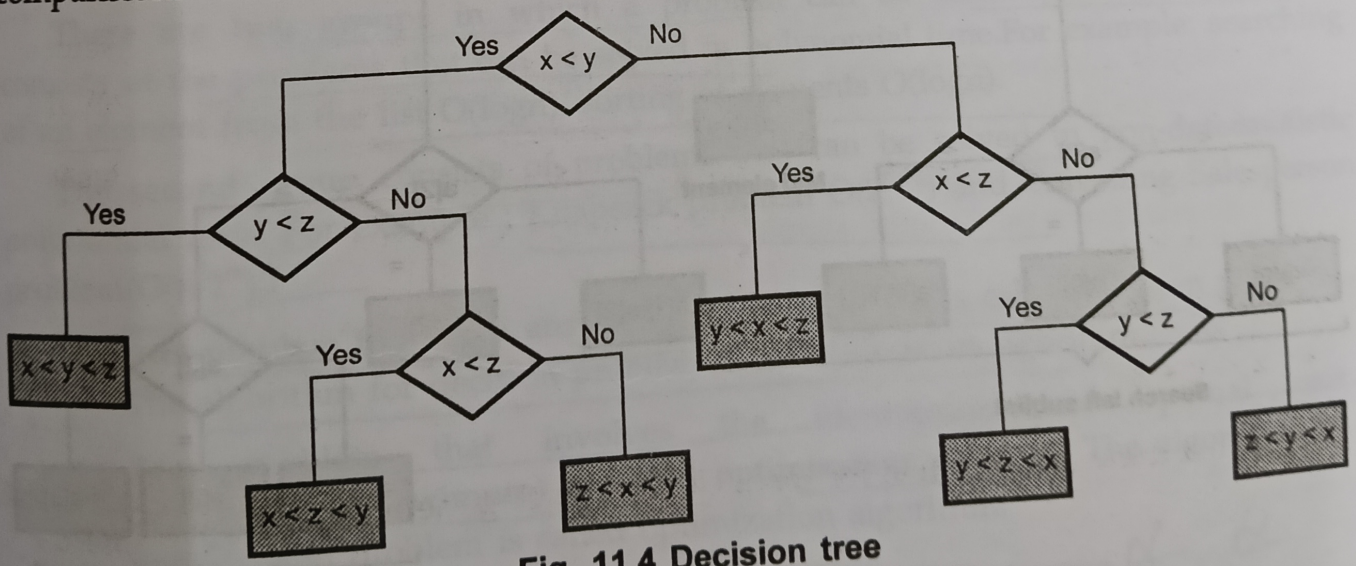


Fig. 11.4 Decision tree

The average case efficiency can be

$$C(n) \geq \log_2 n!$$

11.3.2 Decision Trees for Searching A Sorted Array

When a particular element has to be searched from a sorted list or elements then decision trees can be drawn. We will consider an example of binary search and obtain the lower bound using decision trees. As we know, the worst case for binary search is,

$$C(n) = \lfloor \log n \rfloor + 1$$

$$C(n) = \lceil \log (n+1) \rceil$$

Now to obtain number of comparisions using decision tree consider that there are four elements arranged in ascending order

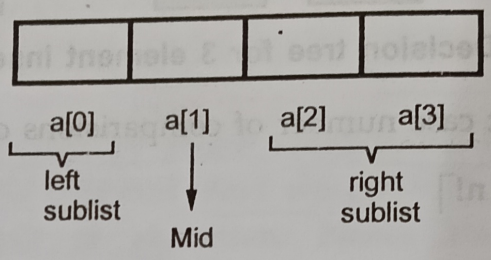


Fig. 11.5

If we draw a ternary decision tree then for n number of elements there will approximately (2n + 1) leaf nodes. The height ternary tree with n nodes will be $\log_3 n$. Thus we will obtain lower bound on number of worst case comparisions

$C(n) \geq \lceil \log_3 (2n+1) \rceil$ total number of leaf nodes.

The ternary decision tree would be

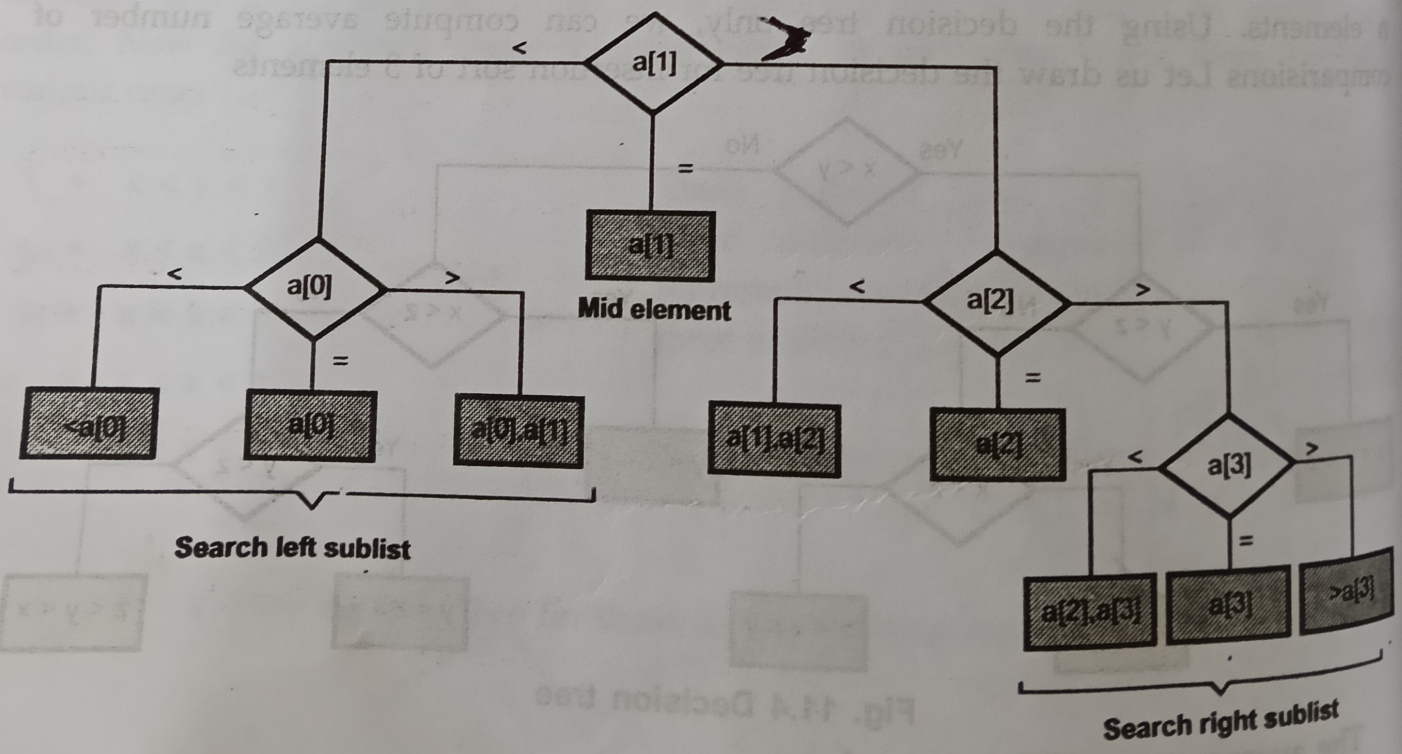


Fig. 11.6

If we redraw the above ternary tree by eliminating the = branch then the tree becomes binary ones such a binary decision tree is -

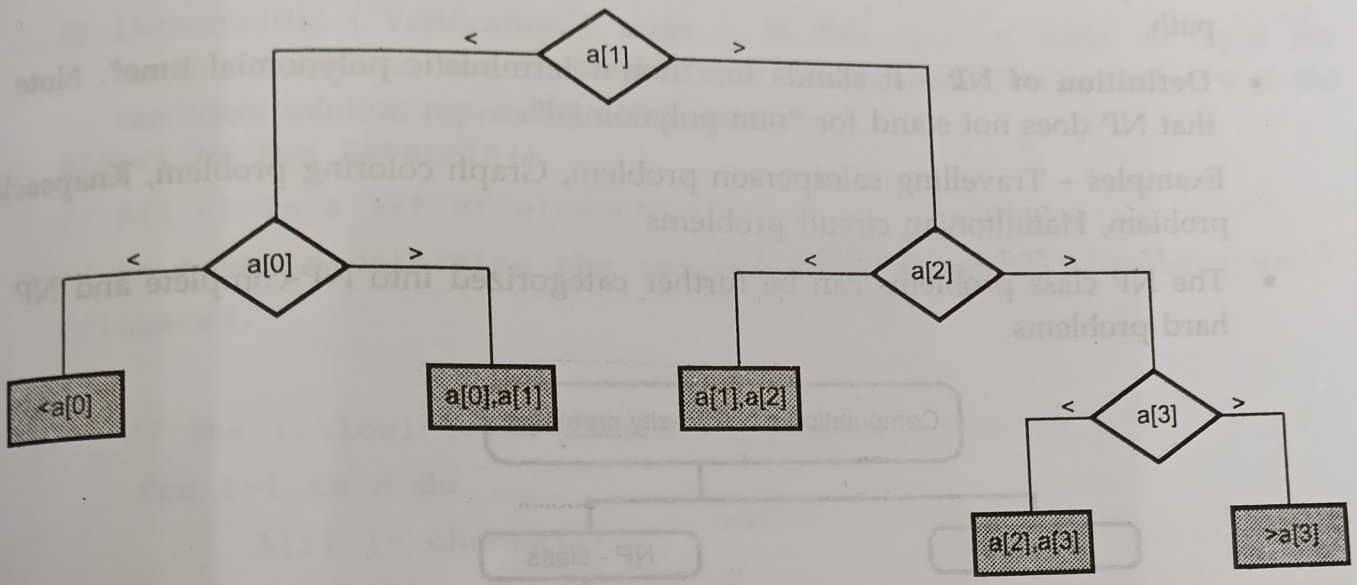


Fig. 11.7 Binary decision tree

Now there are 5 leaves for n = 4 nodes. That means n + 1 leaf nodes. Hence lower bound on the number of worst case comparisins would be :

$$C(n) \geq \lceil \log (n+1) \rceil$$

11.4 P, NP and NP-Complete Problems

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 O(logn).

The second group consists of problems that can be solved in non-deterministic polynomial time. For example : Knapsack problem $O(2^{n/2})$ and Travelling Salesperson problem($O(n^2 2^n)$).

- 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).

  Examples – Searching of key element, Sorting of elements,All pair shortest path.

- **Definition of NP** - 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 NP class problems can be further categorized into NP-complete and NP hard problems.

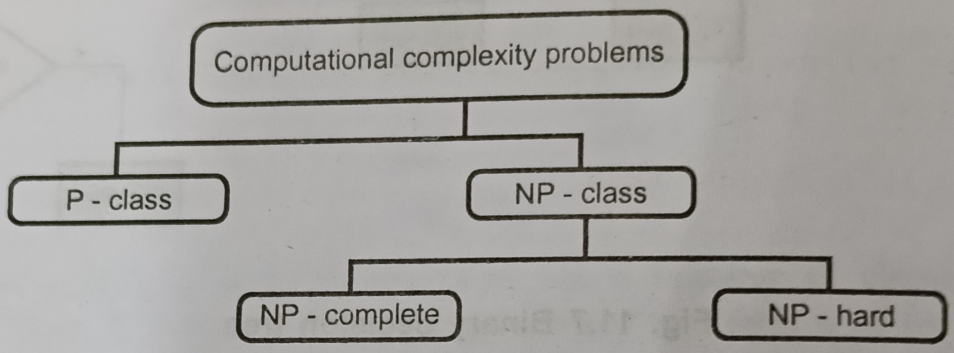


Fig. 11.8

- A problem D is called NP-complete if -

i) It belongs to class NP

ii) 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 can not be NP-complete.
- The NP class problems are the decision problems that can be solved by non deterministic polynomial algorithms.

4.1 Non Deterministic Algorithm

- 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 –

i) Non deterministic (Guessing) stage - generate an arbitrary string that can be thought of as a candidate solution.

ii) Deterministic ("Verification") stage – In this stage it takes as input the candidate solution and the instance to the problem and returns *yes* if 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();
}
```

In the above given non deterministic algorithm there are three functions used –

1) choose – arbitrarily chooses one of the element from given input set.
2) fail – indicates the unsuccessful completion.
3) success – indicates successful completion.

The algorithm is of non deterministic complexity O(1), when A is not ordered then the deterministic search algorithm has a complexity Ω (n).

```
    Profit:=Profit+x[i]*p[i];
}
//verification stage
// selected items exceed the capacity or less profit is earned

if((Wt>m) or (Profit<pt)) then
    fail();
else
    success();//maximum profit earned
}
```

The time required by a non deterministic algorithm with some input set is minimum number of steps needed to reach to a successful completion if there are multiple choices from input set leading to successful completion. In short, a non deterministic algorithm is of complexity O(f(n)) where n is the total size of input.

11.4.2 NP-Hard and NP-Complete 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

  $$P \subseteq NP.$$

- 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 –

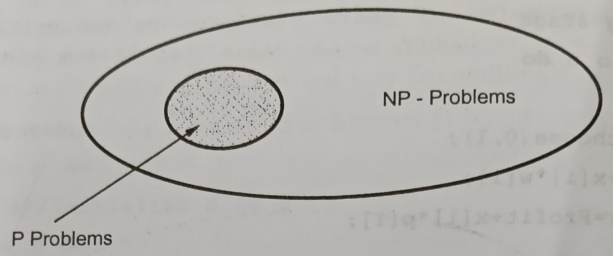


Fig. 11.8 P and NP problems assuming P ≠ NP

- 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 ∝ 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 ∝ B and B ∝ C then A ∝ 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 of NP-complete problem is the intersection of the NP and NP-hard classes.

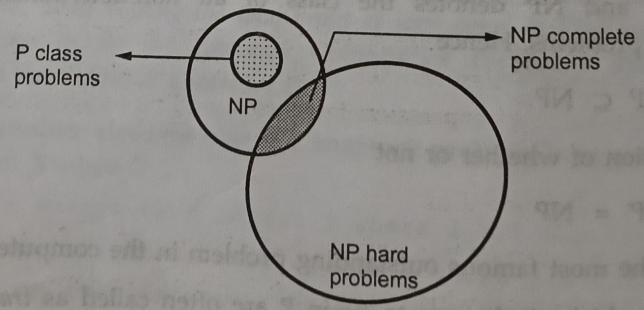


Fig. 11.9 Relationship between P, NP, NP-complete and NP-hard problems

- 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 $\propto$ 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 polynomially equivalent if and only if $P \propto Q$ and $Q \propto P$.

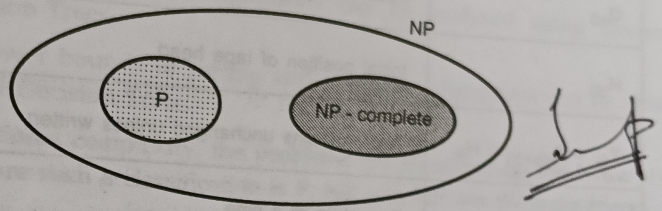


Fig. 11.10

11.4.3 Cook's Theorem

Before discussing the Cook's theorem it is important to know the satisfiability (SAT) problem.

11.4.3.1 Proof of Cook's Theorem

Scientist Stephen Cook in 1971 stated that Boolean Satisfiability (SAT) problem is NP complete.

Proof

Any instance of boolean satisfiability problem is a boolean expression in which boolean variables are combined using boolean operators. An expression is satisfiable if its value results to be true on some assignments of boolean variables.

The boolean satisfiability problem is in NP. This is because a non-deterministic algorithm can guess an assignment of truth values of variables. This algorithm can also determine the value of expression for corresponding assignement and can accept if entire expression is true .

The algorithm is composed of -

- Input tape wherein tape is divided in finite number of cells.
- The read/write head which reads each symbol from tape.
- Each cell contain only one symbol at a time.

12 Coping with the Limitations of Algorithm Power

12.1 Introduction

There are two important algorithmic strategies which are : backtracking and branch and bound. These two methods in which solution is obtained by constructing **state space tree**. We will first discuss these two algorithmic strategies.

In earlier chapter we have discussed the limitations of algorithmic power. Now in this chapter we will see how to cope up with limitations of algorithmic power, by using approximation algorithm.

12.2 Backtracking - General Method

As we know, in the backtracking method in which the desired solution is expressed as a n-tuple $(x_1, x_2 \ldots x_n)$ which is chosen from finite set S.

The solution obtained i.e. $(x_1, x_2 \ldots x_n)$ can either minimizes or maximizes or satisfies the criteria function. Let us denote criteria function by C.

In backtracking the problem can be categorized into three categories -

- Decision problem
- Optimization problem
- Enumeration problem

In decision problem we find "whether there is any feasible solution ?"

In optimization problem we find "whether there exist any best solution ?"

In enumeration problem we find - all the possible feasible solutions.

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.

Backtracking algorithm determines the solution by systematically searching the solution space.

The **solution space** means set of all possible feasible solutions, for a given problem. In backtracking some bounding function is required when the solution is search. The search made in backtracking is generally of depth first search manner. The solutions obtained in backtracking need to satisfy some constraints. The constraints are of two types -

- Explicit constraints
- Implicit constraints

1. **Explicit constraints :** These are set of rules which restrict each vector element to be chosen from given set.
2. **Implicit constraints :** These are set of rules that determine which of the tuples in solution space satisfy the criterion function.

Exhaustive search :

It is basic and trivial search routine in which all possible solutions to a given problem are produced. In this search, it is checked if the problem is solved or not. And the search is continued until a correct solution is generated.

The exhaustive search algorithm produces the entire solution space for the problem.

State space tree :

In backtracking technique while solving a given problem a tree is constructed based on choices made. Such a tree with all possible solutions is called **state-space tree.**

Promising and non-promising nodes :

A node in a state space tree is said to be promising if it corresponds to a partially constructed solution from which a complete solution can be obtained. The nodes which are not promising for solution in a state space tree are called non-promising nodes.

Let us have a general algorithm for backtracking.

```
Algorithm Backtrack(a,T,B)
//Problem Description: This is recursive backtracking algorithm
//Input: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 a problem state.
// Bi+1 is a bounding function such that if Bi+1(a1,a2,...ai+1) is
```

```
false for a path
//Output: (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)
{
   if(Bk(a1,a2,...ak) = true) then
   {
      if ((a1,a2,...ak) is a path to answer node then
         write(a[1],a[2],...a[k]);
      if (k<n) then
         Backtrack(k+1); //find the next set.
   }
}
```

Generation of feasible solutions

Printing the answer nodes as solution to given problem

Otherwise find another set of solution

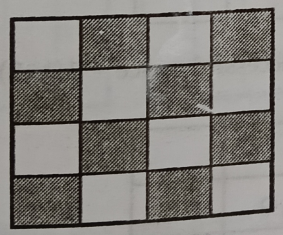
12.3 n-Queen's Problem

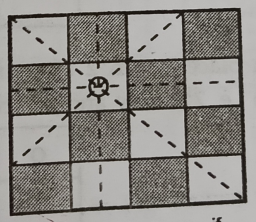
The n-queen's problem can be stated as follows.

Consider a $n \times n$ chessboard on which we have to place n queen's so that no two queen's attack each other by being in the same row or in the same column or on the same diagonal.

For example

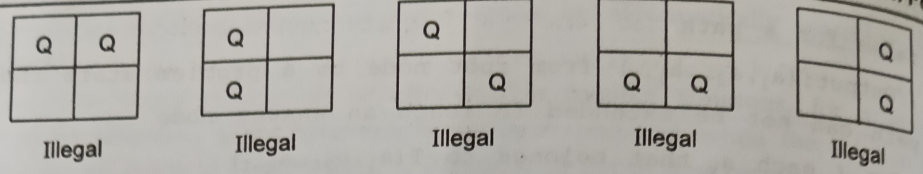
Consider 4×4 board



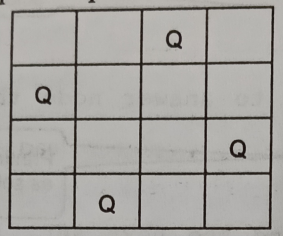


The next queen - if is placed on the paths marked by dotted lines then they can attack each other

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



- But 4 - queen's problem is solvable.

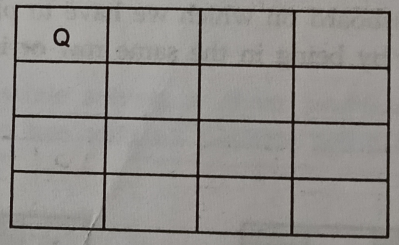


⇐ Note that no two queen's can attack each other.

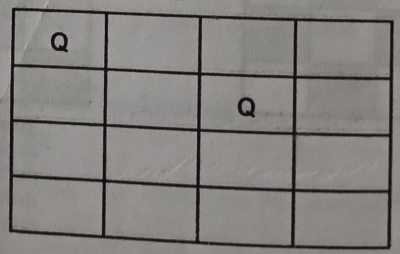
12.3.1 How to Solve n-Queen's Problem ?

Let us take 4 - queen's and 4×4 chessboard.

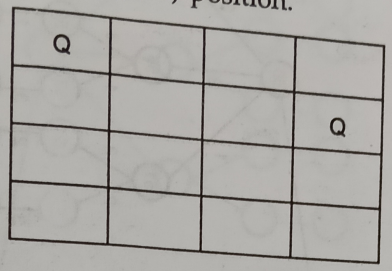
- Now we start with empty chessboard.
- Place queen 1 in the first possible position of its row. i.e. on 1^{st} row and 1^{st} column.



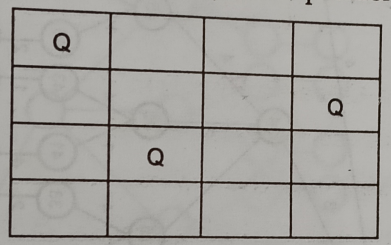
- Then place queen 2 after trying unsuccessful place - 1(1, 2), (2, 1), (2, 2) at (2, 3) i.e. 2^{nd} row and 3^{rd} column.



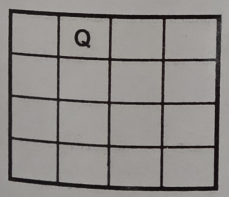
- This is the dead end because a 3rd queen cannot be placed in next column, as there is no acceptable position for queen 3. Hence algorithm backtracks and places the 2nd queen at (2, 4) position.

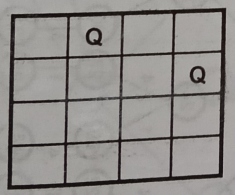


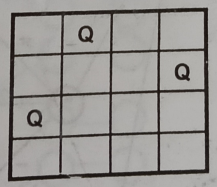
- The place 3rd queen at (3, 2) but it is again another dead end as next queen (4th queen) cannot be placed at permissible position.

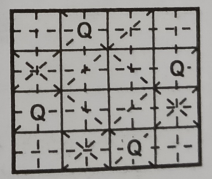


- Hence we need to backtrack all the way upto queen 1 and move it to (1, 2).
- Place queen 1 at (1, 2), queen 2 at (2, 4), queen 3 at (3, 1) and queen 4 at (4, 3).









Thus solution is obtained.

The state space tree of 4-queen's problem is shown in Fig. 12.1

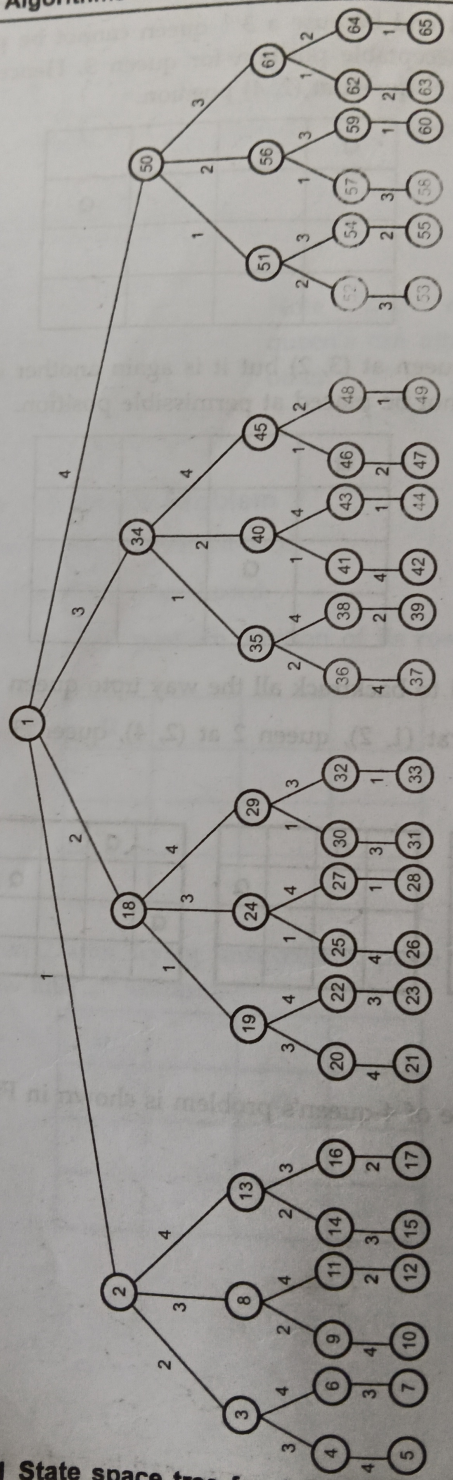


Fig. 12.1 State space tree for 4-queen's problem

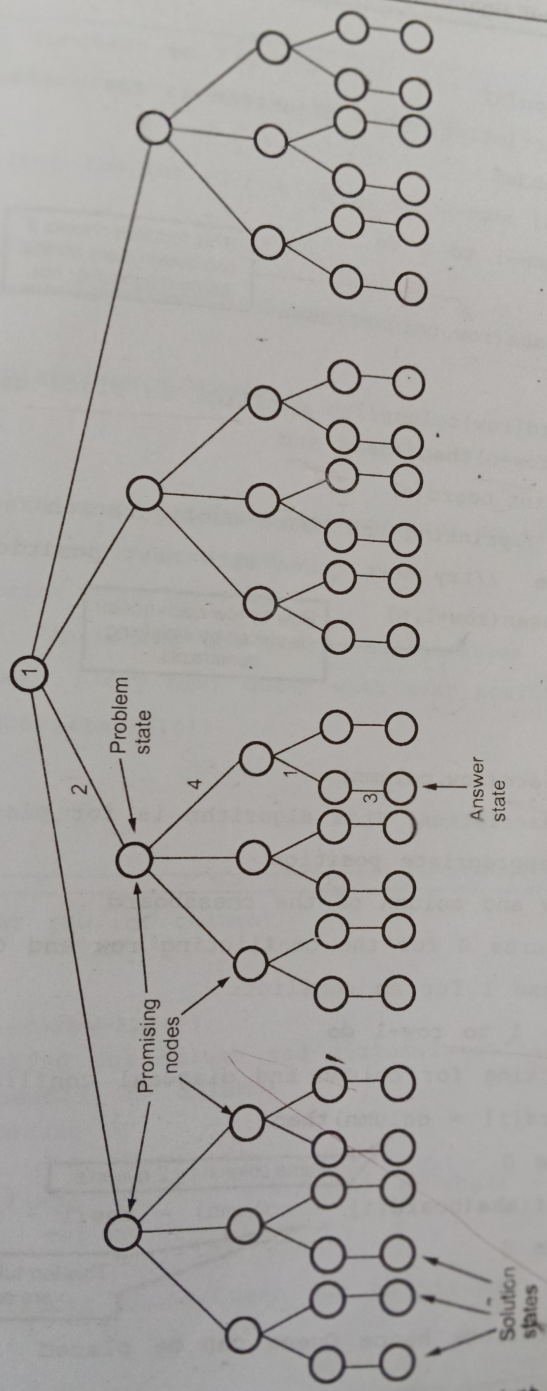


Fig. 12.2 Basic terms used in state space tree

12.3.2 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
         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 queen's 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 queen's are on same diagonal

Solution 2 :

```
     1   2   3   4
1    -   -   Q   -
2    Q   -   -   -
3    -   -   -   Q
4    -   Q   -   -
Press any key to continue...
```

Example 12.1 : *Solve 8-queen's problem for a feasible sequence (6, 4, 7, 1).*

Solution : As the feasible sequence is given, we will place the queen's 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								

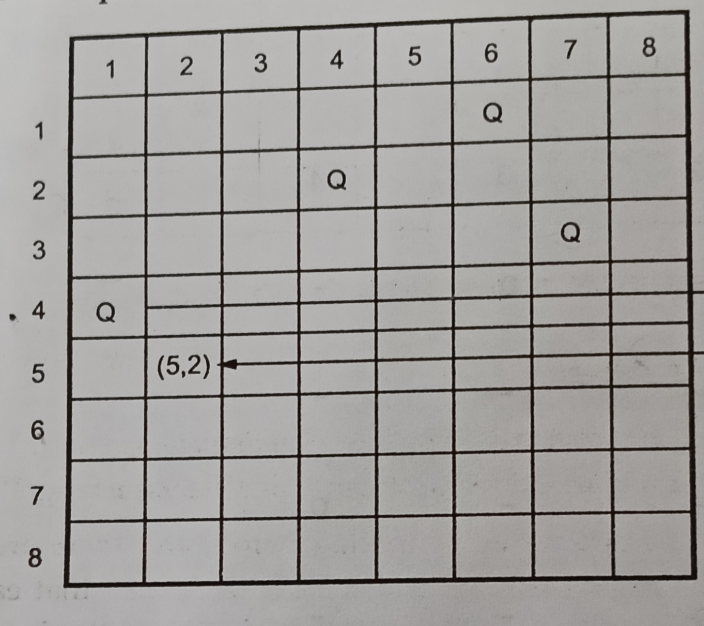
The diagonal conflicts can be checked by following formula-

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 summerised 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-queen's on 8 × 8 board with this sequence is -

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

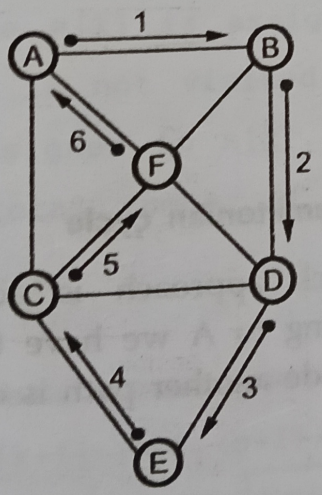
8-queen's with feasible solution (6, 4, 7, 1, 3, 5, 2, 8)

12.4 Hamiltonian Circuit Problem

In this section we will first understand *"What is Hamiltonian circuit ?"* Then we will discuss "How to solve Hamiltonian circuit problem using backtracking ?"

Problem statement : Given an undirected connected graph, and there are any two nodes A and B, then there exists a path from A-B and back to A such that each vertex gets visited exactly once. Such a path in a given graph is called **Hamiltonian circuit.**

For example : Consider a graph G -



The Hamiltonian circuit exists which is A - B - D - E - C - F - A

Fig. 12.3 Graph G showing Hamiltonian cycle

The problem of finding Hamiltonian cycle is solved using backtracking approach. A state space tree can be generated to obtain all possible state space trees.

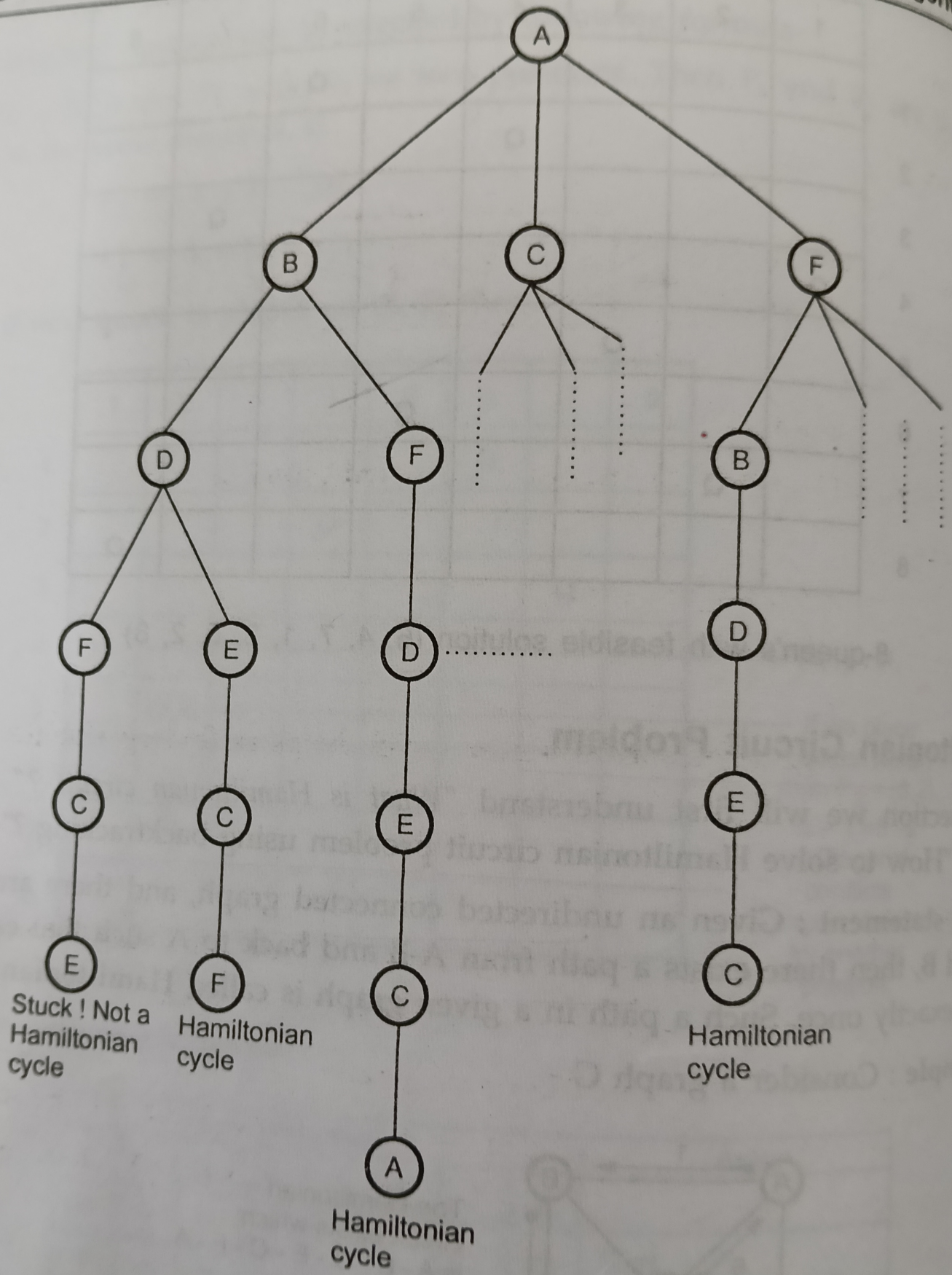


Fig. 12.4 Finding Hamiltonian cycle

In above Fig. 12.4, clearly a backtrack approach is adopted. For instan
-D-F-C-E; here we get stuck. For
ex. Hence we
it at least

```
for (j←1 to k-1 ) do //for
    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
}
}
}
```

The next adjacent vertex is a distinct vertex from which the Hamiltonian path can be traced.

12.5 Subset Sum Problem

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 no backtracking is possible then stop without solution.

This problem can be well understood with some example.

Example 12.2 : *Consider a set S = {5, 10, 12, 13, 15, 18} 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}

Solve these Problems

1) $w = \{3, 5, 6, 7\}$ $M = 15$

2) $w = \{1, 2, 5, 6, 8\}$. $M = 9$.

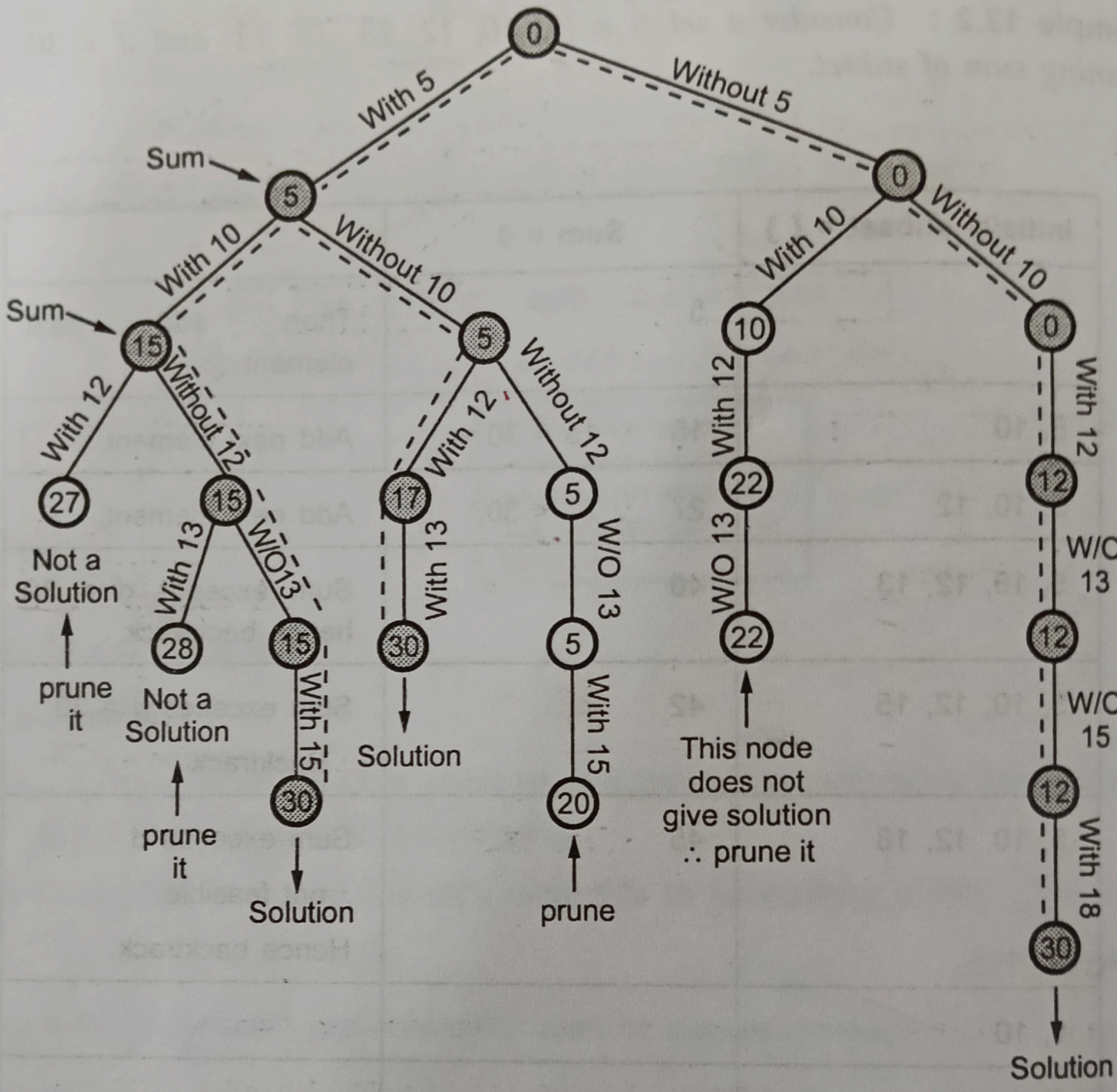


Fig. 12.5 State space tree for sum of subset

Example 12.3 : *Let m = 31 and W = {7, 11, 13, 24}. Draw a portion of stat tree for solving sum-of-subset problem. For the above given algorithm.*

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

The recursive execution of the algorithm will be,

}

12.6 Branch and Bound Technique

In this method a space tree of all possible solutions is generated. Then partitioning or branching is done at each node of the tree. We compute a bound value at each node. We then compare a node's bound value with the value of best solution seen so far. Each time we compare a node's bound value with the value of the best solution obtained uptill now. If the bound value of some node is not better than the best solution then that corresponding node is not expanded further. Such a node is called **non-promising node** and is terminated over there. It is also called that the branch is **pruned** over there. This is done so because such a branch will never lead to solution. This is the **principle idea** of branch and bound.

Thus computation of bounding value at each node, comparing it with best solution and then expanding the node further can lead to a final **solution node.**

Fig. 12.12 Best first branch and bound

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

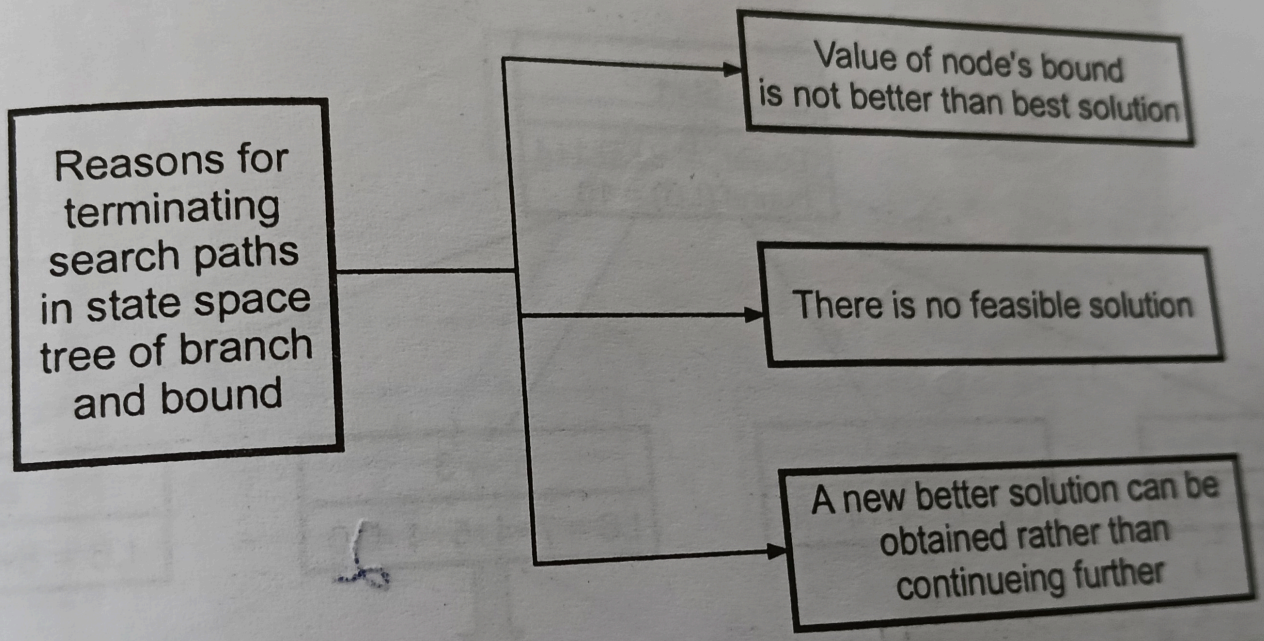


Fig. 12.7

12.7 Assignment Problem

jobs are to be assi

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 upperbound 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	$ 40	10
2	7	$ 42	6
3	5	$ 20	4
4	3	$ 12	4

W = Capacity of Knapsack = W = 10

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.

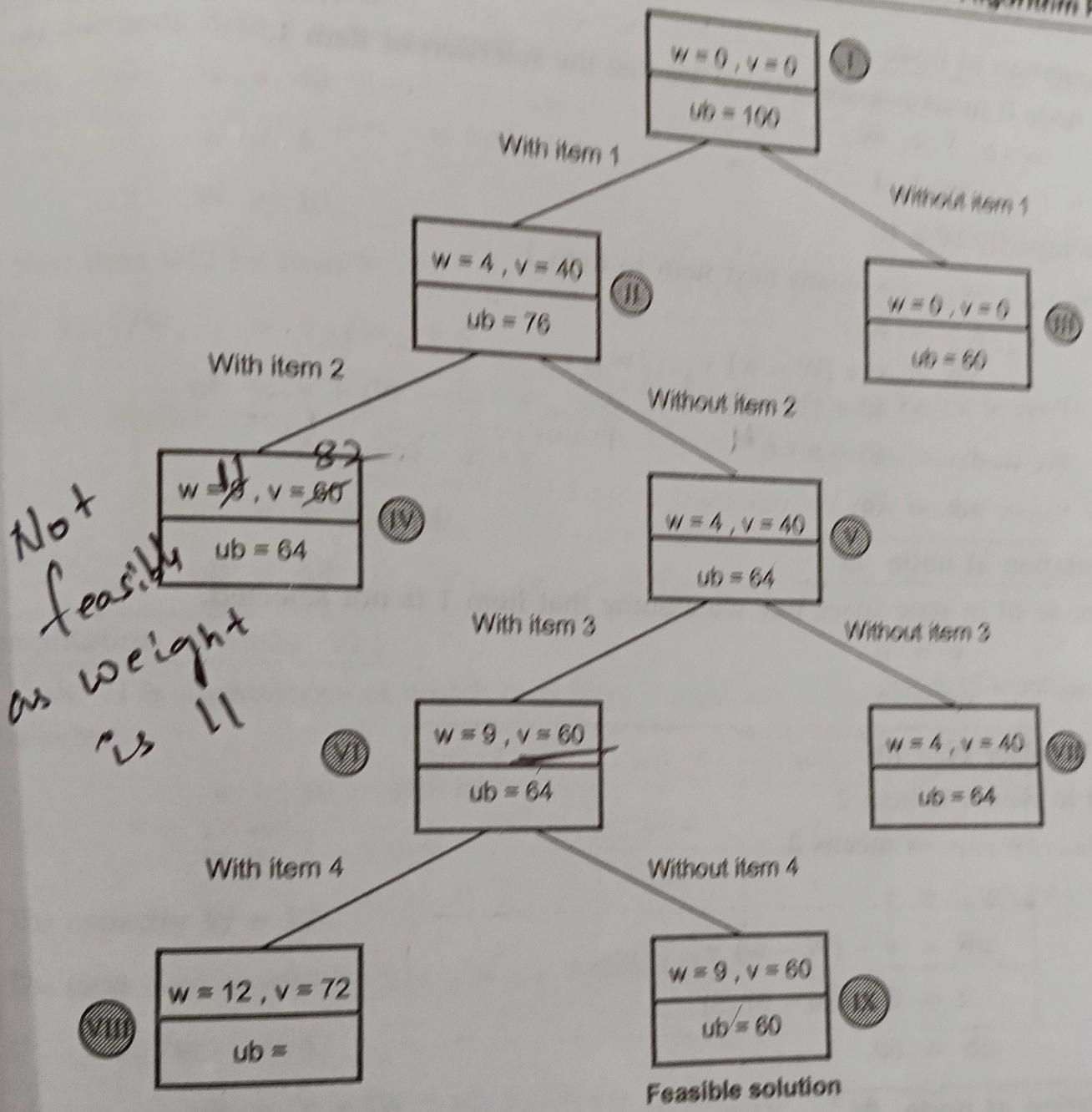


Fig. 12.13 State space tree for Knapsack problem

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. $v_2/w_2 = 6$

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

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

$= 40 + 6 * 6$ 4

$ub = 76$

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

$\therefore$ $v_{i+1}/w_{i+1} = v_3/w_3 = 20/5 = 4$

$\therefore$ $v = 40 + 20 = 60$ (40+42

$w = 4 + 5 = 9$ =82

The capacity $W = 10$ weight is 11 > 10 ↳ given weight so discard.

$\therefore$ $v_{i+1}/w_{i+1} = v_3/w_3 \rightarrow 4$

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

$= 60 + (10 - 9) * 4$

$= 60 + 4$

$= 64$

Computation at node V

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

$\therefore$ $v = 40$

$w = 4$

$W = 10$

Next item will be item 3

$\therefore$ $v_{i+1}/w_{i+1} = v_3/w_3 = 4$

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

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

$= 40 + 6 * 4$

$ub = 64$

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.

$\therefore$ $v = 40 + 20 = 60$

$w = 4 + 5 = 9$

The capacity $W = 10$.

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

$\therefore$ $v_4/w_4 = 4$

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

$= 60 + (10 - 9) * 4$

$= 60 + 4$

$ub = 64$

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.

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

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

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

$$= 40 + 24$$

$$ub = 64$$

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

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

$$v = 40 + 20 + 12 = 72$$

$$W = 10$$

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

$$ub = 72 + (10 - 12) * 0$$

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. .

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

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

$$v = 40 + 20 = 60$$

$$W = 10$$

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

$$= 60 + (10 - 9) * 0$$

$$ub = 60$$

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 {item 1, item 3} with maximum weight 9 and maximum profit gained = 60 $ as a solution.

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

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

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

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

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

$= 32/2$

$= 16$

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.

12.10 Approximation Algorithms for NP-Hard Problems

In this section we will discuss two important issues namely, **"What is NP-hard problem ?"** and **"How approximation algorithms are used for NP-hard problems ?"** Let us start our discussion with the understanding of NP-hard problems. In computational complexity theory there are different types of problems. Some problems are **decision problems** for which answer is yes or no, others are **search problems** and

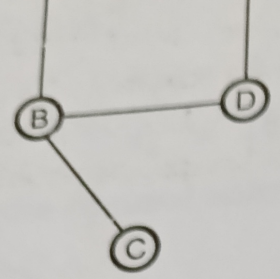


Fig. 12.24

Step 2 : Start from city A and have a DFS walk around the tree.

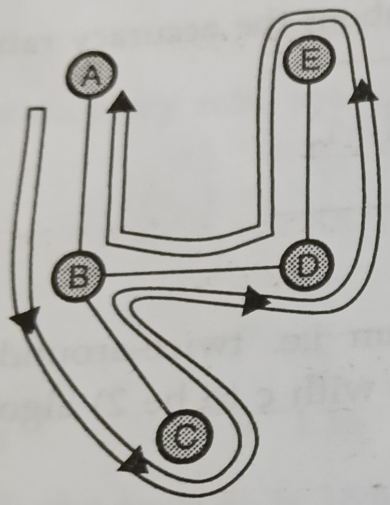


Fig. 12.25

Step 3 : Record the visited nodes A - B - C - B - D - E - D - B - A. Eliminate duplicates then A-B-C-D-E-A . This basically gives Hamiltonian circuit.

But the tour obtained is not the optimal tour.

12.10.2 Approximation Algorithms for Knapsack Problem

In this section we will discuss another NP-hard problem i.e. knapsack problem.

The knapsack problem can be defined as follows : Given that there are n items with weights $w_1, w_2 \ldots w_n$ and $v_1, v_2, \ldots v_n$ as profits associated with each item. *Then find the most valuable subset of items that fits into the knapsack. There are two variations of knapsack problem -*

- Discrete knapsack problem - Select the item entirely or don't select at all.
- Continuous knapsack problem - The fractional item can be allowed to select.

Now there exists two approximation algorithms for knapsack problem.

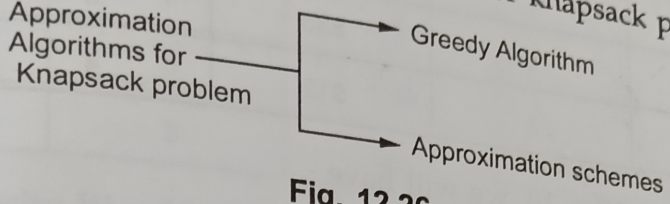


Fig. 12.26

Let us discuss these in detail -

1. Greedy algorithm for knapsack problem

Using greedy approach for knapsack, we can consider either weight or profit. That means either we select the items with least weights or we select items yeilding heavier profit items. But there is no guarantee that the knapsack's capacity will be used efficiently. Hence we will consider both **value to weight** ratio. Let us discuss Greedy approach for solving both discrete knapsack and continuous knapsack problem.

Greedy algorithm for the discrete knapsack problem

1. Compute value/weight ratio v_i / w_i for all items.
2. Sort the items in non increasing order of the ratios v_i / w_i.
3. Repeat until no item is left in sorted list using following steps
   a) If current item fits, use it.
   b) Otherwise skip this item, and proceed to next item.

➠ **Example 12.5 :** *Solve the knapsack problem for the following instance using Greedy approach. The item can be completely selected or skiped completely.*

Item	Weight	Value
1	7	$49
2	3	$12
3	4	$42
4	5	$30

4 40
7 42
5 25
3 12

Consider W = 10.

Solution : This is also called 0-1 knapsack. Either we can completely select an item or skip it. First of all we will compute value-to-weight ratio and arrange them in non increasing order of the ratio.

Item	Weight	Value	Value/Weight	
3	4	$42	10.5	→
1	7	$49	7	
4	5	$30	6	→
2	3	$12	4	

To fulfil the capacity W = 10, we will have -

- add item of weight 4
- skip item of weight 7
- add item of weight 5
- skip item of weight 3

} This will produce the total weight of 9 with profit of $72

This is the solution for given instance of knapsack problem.

But the Greedy algorithm does not give optimal solution always rather there is no upper bound on the accuracy of approximate solution.

Greedy algorithm for continous knapsack problem

1. Compute value/weight ratio v_i / w_i for all items.
2. Sort the items in **non increasing order** of their ratios - v_i / w_i.
3. Repeat until no item is left in sorted list using following steps -
   a) if current item fits, use it.
   b) Otherwise take its largest fraction to fill the knapsack to its full capacity.

Example 12.6 : *Solve the knapsack problem for following instance using Greedy method.*

Item	Weight	Value
1	7	$49
2	3	$12
3	4	$42
4	5	$30

The capacity of knapsack is W = 10.

Solution : This is fractional knapsack problem. The items can be selected fractionally. First of all we will obtain value to weight ratio and arrange the items in non increasing order.

Item	Weight	Value	Value to Weight	
3	4	$42	10.5	→
1	7	$49	7	→
4	5	$30	6	
2	3	$12	4	

To fulfill the capacity W = 10 we will have

- add item of weight 4.
- select item of weight 7 and take its fractional 6/7 weight.

∴ Weights of selected items

$\therefore 4+7\times\frac{6}{7} = 10$ which fits into the knapsack.

Hence, the profit obtained will be

$$42+49\times\frac{6}{7}$$

$$= 42 + 42$$

$$= \$84$$

This is an optimal solution to given instance of knapsack.

This algorithm always give optimal solution to continuous knapsack problem.

2. Approximation scheme

In this scheme certain approximation is obtained to certain accuracy level. This scheme is suggested by Prof. S. Sahni and is particularly used for 0-1 knapsack or discrete version of knapsack problem. The accuracy level can be defined as

$$\frac{f(s_a^k)}{f(s^*)} \leq \left(1+\frac{1}{k}\right) \text{ of n items}$$

where k is an integer such that less than or equal to k items can be selected.

Example 12.7 : *For the given instance of knapsack problem obtain optimal solution for approximation scheme k = 2.*

Item	Weight	Value	Value to weight
1	4	$42	10.5
2	7	$49	7
3	5	$30	6
4	1	$4	4

Capacity W = 10

Solution : As capacity of knapsack W = 10 we will select items fulfilling the capacity and giving maximum profit. The approximation scheme k = 2 means subsets of less than equal to k items i.e. subset with 0, 1 or 2 items can be considered and remaining items satisfying the capacity constraints are added accordingly. From such a list we then select an optimal solution.

Subset to be considered	Possible added item	Value of profit in $
{0}	1, 3, 4	42+30+4 = 76
{1}	3, 4	Subset + added item {1} + {3, 4} = 76
{2}	4	{2} + {4} = 49+4 = 53
{3}	1, 4	{3} + {1, 4} = 30 + {42+4} = 76
{4}	1, 3	76
{1, 2}	Not feasible	As weight = 11 i.e. 11 > W
{1, 3}	4	76
{1, 4}	3	76
{2, 3}	Not feasible	As weight = 12 i.e. 12 > W
{2, 4}	Cannot add any item	53
{3, 4}	1	76

Thus we obtain optimal solution as {1, 3, 4}. But this scheme is more of theoretical use than the practical use.