Part I : Introduction to Analysis of Algorithm

2.1 Analysis of Algorithm

SPPU : Dec.-08,10,12, Aug.-15, May-09, Marks 8

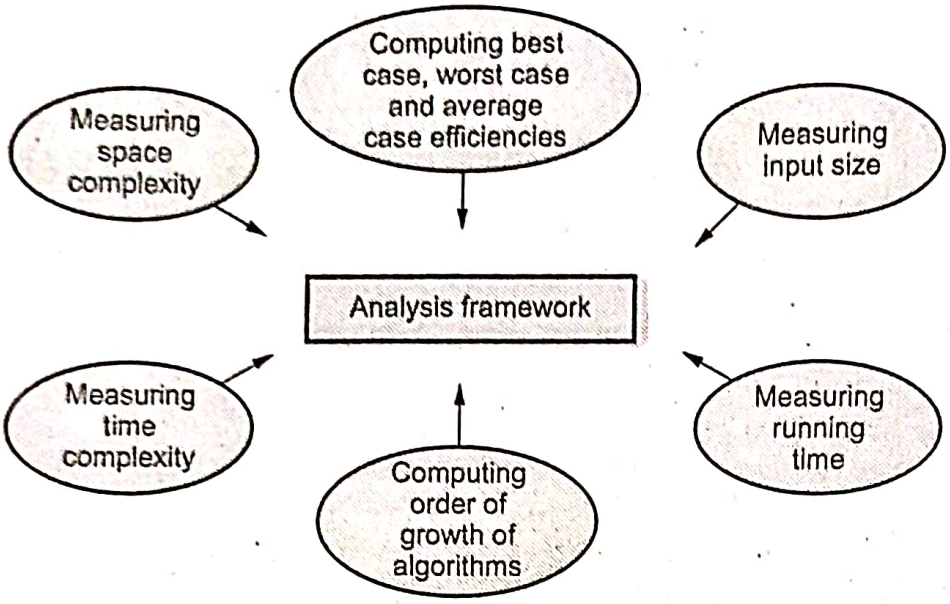


Fig. 2.1.1 Analysis framework

The efficiency of an algorithm can be decided by measuring the performance of an algorithm. We can measure the performance of an algorithm by computing two factors.

1. Amount of time required by an algorithm to execute.
2. Amount of storage required by an algorithm.

This is popularly known as **time complexity** and **space complexity** of an algorithm.

2.1.1 Space Complexity

The space complexity can be defined as amount of memory required by an algorithm to run.

To compute the space complexity we use two factors : **Constant and instance characteristics**. The space requirement S(p) can be given as

$$S(p) = C + Sp$$

where C is a constant i.e. fixed part and it denotes the space of inputs and outputs. This space is an amount of space taken by instruction, variables and identifiers. And Sp is a space dependent upon instance characteristics. This is a variable part whose space requirement depends on particular problem instance.

There are two types of components that contribute to the space complexity - Fixed part and variable part.

SPPU : May-09, Marks 4

The **fixed part** includes space for

- Instructions
- Variables
- Array size
- Space for constants

The **variable part** includes space for

- The variables whose size is dependent upon the particular problem instance being solved. The control statements (such as for, do, while, choice) are used to solve such instances.
- Recursion stack for handling recursive call.

Consider three examples of algorithms to compute the space complexity.

Example 1 : Compute the space required by following algorithm.

```
Algorithm Add (a,b,c)
//Problem Description : This algorithm computes the addition
//of three elements
//Input: a,b, and c are of floating type
//Output: The addition is returned
        return a+b+c
```

The space requirement for algorithm given in Example 1 is

$$S(p) = C \qquad \Theta (Sp) = 0$$

If we assume that a, b and c occupy one word size then total size comes to be 3.

Example 2 : Compute the space needed by the following algorithms justify your answer.

```
Algorithm Sum(a,n)
{
    s : = 0.0 ;
        For i : = 1 to n do
            s : = s + a[i] ;
            return s
}
```

SPPU : Dec.-08, Marks 8

In the given code we require space for

```
s : = 0      ←    O(1)
    For i : = 1 to n    ←    O(n)
        s : = s + a[i] ; ← O(n)
        returns ;        ← O(1)
```

Hence the space complexity of given algorithm can be denoted in terms of big-oh notation. It is **O(n).**

Example 3 : Compute the space required by following algorithm.

```
Algorithm Add (x,n)
//Problem Description: This is a recursive algorithm which
```

```
//computes addition of all the elements in an array x[]
//Input: x[i] is of floating type, total number of elements
//in an array
//Output: returns addition of n elements of an array
return Add (x,n-1)+x[n]
```

The space requirement is -

$$S(p) \geq 3(n + 1)$$

The internal stack used for recursion includes space for formal parameters, local variables and return address. The space required by each call to function Add requires atleast three words (space for n values + space for return address + pointer to x [ ]). The depth of recursion in n+1 (n times call to function and one return call). The recursion stack space will be $\geq 3(n+1)$.

2.1.2 Time Complexity

The time complexity of an algorithm is the **amount of computer time required** by an algorithm to run to completion.

It is **difficult to compute** the **time complexity** in terms of **physically clocked time**. For instance in multiuser system, executing time depends on many **factors** such as -

- System load
- Number of other programs running
- Instruction set used
- Speed of underlying hardware

The time complexity is therefore given in terms of frequency count.

Frequency count is a count denoting number of times of execution of statement.

Definition : The frequency count is a count that denotes how many times particular statement is executed.

For Example : Consider following code for counting the frequency count

```
void fun()
{
 int a;
 a=10; .....1
 printf("%d",a); .....1
}
```

The frequency count of above program is **2.**

Example 2.1.1 *Obtain the frequency count for the following code.*

Solution :

```
void fun()
{
 int a;
 a=0; .....1
 for(i=0;i<n;i++) .....n+1
  {
    a = a+i; .....n
  }
  printf("%d",a); .....1
}
```

The frequency count of above code is **2n + 3.**

The for loop in above given fragment of code is executed n times when the condition is true and one more time when the condition becomes false. Hence for the for loop the frequency count is n + 1. The statement inside the for loop will be executed only when the condition inside the for loop is true. Therefore this statement will be executed for n times. The last printf statement will be executed for once.

Example 2.1.2 *Obtain the frequency count for the following code.*

Solution :

```
void fun(int a[][],int b[][])
{
 int c[3][3];
 for(i=0;i<m;i++) .....m+1
 {
  for(j=0;j<n;j++) .....m(n+1)
  {
   c[i][j]=a[i][j]+b[i][j]; .....m.n
  }
 }
```

The frequency count = (m + 1) + m(n + 1) + mn = 2m + 2mn + 1 = **2m(1 + n) + 1**

Example 2.1.3 *Obtain the frequency count for the following code.*

```
for(i=1;i<=n;i++)
{
 for(j=1;j<=n;j++)
 {
   c[i][j]=0;
   for(k=1;k<=n;k++)
     c[i][j]=c[i][j]+a[i][k]*b[k][j];
 }
}
```

Solution :

Statement	Frequency count
for(i=1;i<=n;i++)	n + 1
for(j=1;j<=n;j++)	n.(n + 1)
c[i][j]=0;	n.(n)
for(k=1;k<=n;k++)	n.n(n + 1)
c[i][j]=c[i][j]+a[i][k]*b[k][j];	n.n.n
Total	$2n^3 + 3n^2 + 2n + 1$

After counting the frequency count, the constant terms can be neglected and only the order of magnitude is considered. The time complexity is denoted in terms of algorithmic notations. The Big oh notation is a most commonly used algorithmic notation. For the above frequency count all the constant terms are neglected and only the order of magnitude of the polynomial is considered. Hence the time complexity for the above code can be $O(n^3)$. The higher order is the polynomial is always considered.

2.1.3 Measuring an Input Size

It is observed that if the input size is longer, then usually algorithm runs for a longer time. Hence we can compute the efficiency of an algorithm as a function to which input size is passed as a parameter. Sometimes to implement an algorithm we require prior knowledge of input size. For example, while performing multiplication of two matrices we should know order of these matrices. Then only we can enter the elements of matrices. Sometimes the input size is taken as an approximate value. For example, in spell checking algorithms we can predict exact size of the input.

2.1.4 Measuring Running Time

SPPU : Dec.-10, Marks 8

We have already discussed that the time complexity is measured in terms of a unit called **frequency count.**

The time which is measured for analyzing an algorithm is generally running time.

From an algorithm :

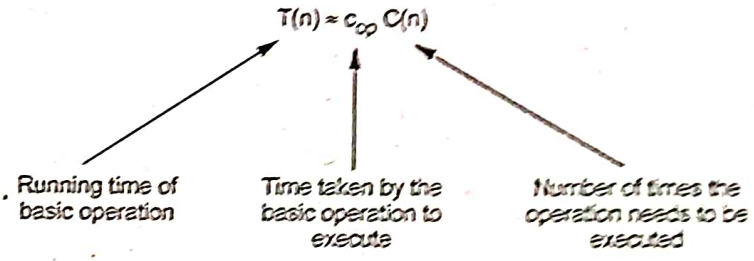
- We first identify the important operation (core logic) of an algorithm. This operation is called the **basic operation.**
- It is not difficult to identify basic operation from an algorithm. Generally the operation which is more time consuming is a basic operation in the algorithm. Normally such basic operation is located in **inner loop.** For example in sorting algorithm the operation which is comparing the elements and then placing them at

appropriate locations is a basic operation. The concept of basic operations can be well understood with the help of following example.

Problem statement	Input size	Basic operation
Searching a key element from the list of n elements.	List of n elements.	Comparison of key with every element of list.
Performing matrix multiplication.	The two matrices with order $n \times n$.	Actual multiplication of the elements in the matrices.
Computing GCD of two numbers.	Two numbers.	Division.

Table 2.1.1 Basic operations from input

- Then we compute total number of time taken by this basic operation. We can compute the running time of basic operation by given formula.



- Using this formula the computing time can be obtained approximately.

Review Questions

1. *What are the basic components that contribute to the space complexity ?*
   SPPU : May-09, Marks 4
2. *Explain the different ways of measuring the running time of an algorithm.*
   SPPU : Dec.-12, Marks 6
3. *Explain time and space complexity with suitable examples.*
   SPPU : Aug.-15, (In Sem.) Marks 4

2.2 Asymptotic Notations

SPPU : Dec.-06,08,09,10,12,14, May-08,09,12,13,14,17, Aug.-15, Marks 8

To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity.

Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "average time".

Various notations such as Ω, Θ and O used are called asymptotic notions.

2.2.1 Big oh Notation

The **Big oh** notation is denoted by 'O'. It is a method of representing the **upper bound of** algorithm's running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.

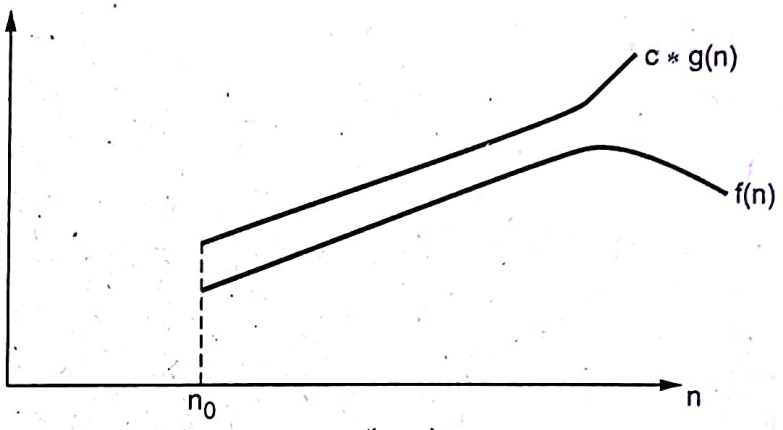
Definition

Let f(n) and g(n) be two non-negative functions.

Let n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that $c > 0$. We can write

$$f(n) \leq c*g(n)$$

then f(n) is big oh of g(n). It is also denoted as **f(n) ∈ O (g(n))**. In other words f(n) is less than g(n) if g(n) is multiple of some constant c.



f(n) ∈ O(g(n))

Fig. 2.2.1 Big oh representation

Example : Consider function f(n) = 2n + 2 and $g(n) = n^2$. Then we have to find some constant c, so that f(n) ≤ c * g(n). As f(n) = 2n + 2 and $g(n) = n^2$ then we find c for n = 1 then

$$f(n) = 2n+ 2$$
$$= 2(1) + 2$$
$$f(n) = 4$$

and $$g(n) = n^2$$
$$= (1)^2$$
$$g(n) = 1$$

i.e. $$f(n) > g(n)$$

If n = 2 then,

$$f(n) = 2(2) + 2$$
$$= 6$$
$$g(n) = (2)^2$$
$$g(n) = 4$$

i.e. $f(n) > g(n)$

If n = 3 then,

$$f(n) = 2(3) + 2$$

$$= 8$$

$$g(n) = (3)^2$$

$$g(n) = 9$$

i.e. $f(n) < g(n)$ is true.

Hence we can conclude that for $n > 2$, we obtain

$$f(n) < g(n)$$

Thus always upper bound of existing time is obtained by big oh notation.

2.2.2 Omega Notation

Omega notation is denoted by 'Ω'. This notation is used to represent the **lower bound** of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm.

Definition

A function f(n) is said to be in Ω (g(n)) if f(n) is bounded below by some positive constant multiple of g(n) such that

$$f(n) \geq c * g(n) \qquad \text{For all } n \geq n_0$$

It is denoted as $f(n) \in \Omega(g(n))$. Following graph illustrates the curve for Ω notation.

Example :

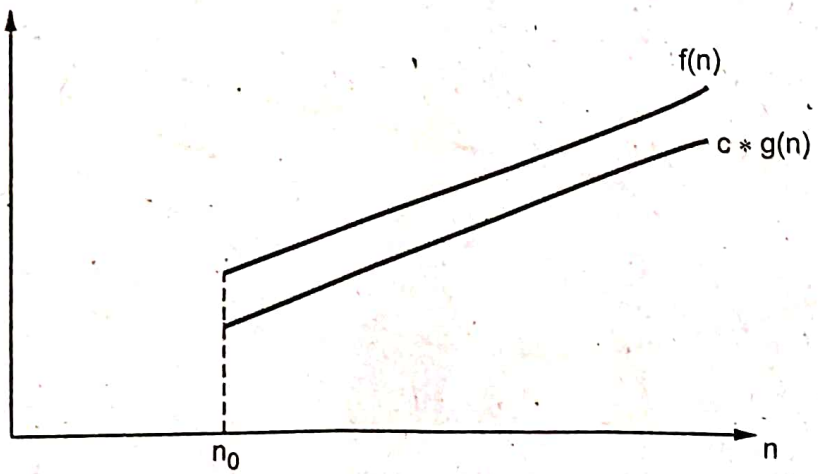


Fig. 2.2.2 Omega representation

Consider $f(n) = 2n^2 + 5$ and $g(n) = 7n$

Then if $n = 0$

$f(n) = 2(0)^2 + 5$

$= 5$

$g(n) = 7(0)$

$= 0$ i.e. $f(n) > g(n)$

But if $n = 1$

$f(n) = 2(1)^2 + 5$

$= 7$

$g(n) = 7(1)$

7 i.e. $f(n) = g(n)$

If n = 3 then,

$f(n) = 2(3)^2 + 5$

$= 18 + 5$

$= 23$

$g(n) = 7(3)$

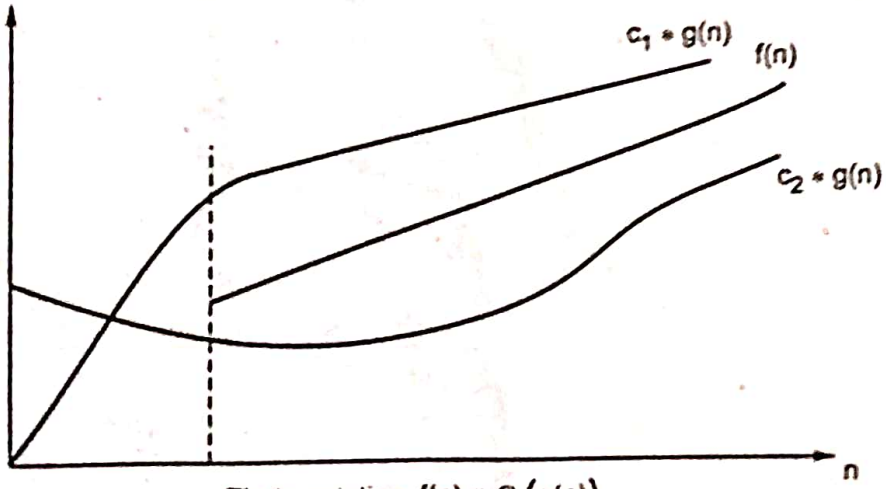
$= 21$

i.e. $f(n) > g(n)$

Thus for $n > 3$ we get $f(n) > c * g(n)$.

It can be represented as,

$2n^2 + 5 \in \Omega(n)$



Theta notation $f(n) \in \Theta(g(n))$

Fig. 2.2.3 Theta representation

2.2.3 Θ Notation

The theta notation is denoted by Θ. By this method the running time is between upper bound and lower bound.

Definition

Let f(n) and g(n) be two non negative functions. There are two positive constants namely c_1 and c_2 such that,

$$c_2 g(n) \leq f(n) \leq c_1 g(n)$$

Then we can say that,

$$f(n) \in \Theta(g(n))$$

Example :

If f(n) = 2n + 8 and g(n) = 5n.

where $n \geq 2$

Similarly $f(n) = 2n + 8$

$g(n) = 7n$

i.e. $5n < 2n + 8 < 7n$ For $n \geq 2$

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$.

The theta notation is more precise with both big oh and omega notation.

Some examples of asymptotic order

1) $\log_2 n$ is f(n) then

$\log_2 n \in O(n)$ ∵ $\log_2 n \leq O(n)$, the order of growth of $\log_2 n$ is slower than n.

$\log_2 n \in O(n^2)$ ∵ $\log_2 n \leq O(n^2)$, the order of growth of $\log_2 n$ is slower than n^2 as well.

But

$\log_2 n \notin \Omega(n)$ ∵ $\log_2 n \leq \Omega(n)$ and if a certain function F(n) is belonging to Ω(n) it should satisfy the condition $f(n) \geq c * g(n)$

Similarly $\log_2 n \notin \Omega(n^2)$ or $\Omega(n^3)$

2) Let $f(n) = n(n-1)/2$

Then

$n(n-1)/2 \notin O(n)$ ∵ $f(n) > O(n)$ we get $f(n) = n(n-1)/2 = \frac{n^2-1}{2}$

i.e. maximum order is n^2 which is $> O(n)$.

Hence $f(n) \notin O(n)$

But $n(n-1)/2 \in O(n^2)$ As $f(n) \leq O(n^2)$

and $n(n-1)/2 \in O(n^3)$

Similarly,

$n(n-1)/2 \in \Omega(n)$ ∵ $f(n) \geq \Omega(n)$

$n(n-1)/2 \in \Omega(n^2)$ ∵ $f(n) \geq \Omega(n^2)$

$n(n-1)/2 \notin \Omega(n^3)$ ∵ $f(n) > \Omega(n^3)$

2.2.4 Properties of Order of Growth

1. If $f_1(n)$ is order of $g_1(n)$ and $f_2(n)$ is order of $g_2(n)$, then $f_1(n)+f_2(n) \in O(\max(g_1(n), g_2(n))$.
2. Polynomials of degree $m \in \Theta(n^m)$.

   That means maximum degree is considered from the polynomial.

   For example : $a_1n^3 + a_2n^2 + a_3n + c$ has the order of growth $\Theta(n^3)$.
3. $O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$.
4. Exponential functions a^n have different orders of growth for different values of a.

Key Points

i) $O(g(n))$ is a class of functions $f(n)$ that grows less fast than $g(n)$, that means $f(n)$ possess the time complexity which is always slower than the time complexities that $g(n)$ have.

ii) $\Theta(g(n))$ is a class of functions $f(n)$ that grows at same rate as $g(n)$.

iii) $\Omega(g(n))$ is a class of functions $f(n)$ that grows faster than or atleast as fast as $g(n)$. That means $f(n)$ is greater than $\Omega(g(n))$.

Example 2.2.1 *State whether the following functions are CORRECT or INCORRECT and justify your answer.*

i) $3n+2 = O(n)$ ii) $100n+6 = O(n)$ iii) $10n^2+4n+2 = O(n^2)$

SPPU : Dec.-06, Marks 6, Dec.-09, 10, Marks 2

Solution : The function f(n) is said to be in O(g(n)) i.e.

$$f(n) = O(g(n) \text{ or } f(n) \in O(g(n)),$$

if it is bounded above by some constant multiple of g(n) for all large n.

In order words if,

$$f(n) \leq c \cdot g(n) \quad \text{For } n \geq n_0$$

Consider

i) $f(n) = 3n + 2$

$g(n) = n$

This assertion can be CORRECT if $c = 5$ and $n \geq 1$

i.e. $f(n) = 3n_0 + 2$

$= 3(1) + 2$ When n = 1

$= 5$

$c \cdot g(n) = 5 \times 1$

$= 5$

$\therefore$ $f(n) \leq c \cdot g(n)$ is true.

If $n = 2$ and $c = 5$ then

$f(n) = 3n + 2 = 3(2) + 2 = 8$

$g(n) = 2 \quad \therefore \quad c \cdot g(n) = 5(2) = 10$

Again $f(n) < g(n)$ is true

Thus we can find that $f(n) \leq g(n)$ remains true for $n \geq 1$ when $c = 5$.

But $n = 0$ then

$f(n) = 3n + 2 = 3(0) + 2 = 2$

$g(n) = 0 \; c \cdot g(n) = 5 \cdot (0) = 0$

Then $f(n) > c \cdot g(n)$. Hence $3n + 2 = O(n)$ is correct for $n \geq 1$ and $c = 5$

$3n + 2 = O(n)$ is INCORRECT with $n = 0$ and any constant value.

ii) $f(n) = 100\,n + 6$

$g(n) = n$

$f(n) = O(n)$ is CORRECT if we consider $c = 106$ and $n \geq 1$.

Case 1 : If $n_0 = 1$ then

$$f(n) = 100n + 6$$
$$= 100(1) + 6$$
$$= 106$$
$$g(n) = 1$$
$$\therefore \quad c \cdot g(n) = 106 \cdot (1)$$
$$= 106$$

Thus $f(n) \leq c \cdot g(n)$ is true.

Case 2 : If $n_0 = 2$

$$f(n) = 100n + 6$$
$$= 100(2) + 6$$
$$= 206$$
$$g(n) = 2$$
$$\therefore \quad c \cdot g(n) = 106 \cdot (2)$$
$$= 212$$

Again $f(n) < c \cdot g(n)$

Thus if $f(n) \leq c \cdot g(n)$ for $n \geq 1$ then $100n + 6 = O(n)$ is CORRECT.

But for $n = 0$

$$f(n) = 100n + 6$$
$$= 100(0) + 6$$
$$= 6$$
$$g(n) = 0$$
$$c \cdot g(n) = c \cdot (0) = 0$$

Thus $f(n) > c.g(n)$

Hence $100n + 6 = 0(n)$ is INCORRECT for $n = 0$.

iii) $f(n) = 100n^2 + 4n + 2$

$g(n) = n^2$

Consider $c = 106$. The values for n can be

Case 1 : If $n = 1$ then

$f(n) = 100n^2 + 4n + 2$

$= 100(1)^2 + 4(1) + 2$

$= 100 + 4 + 2$

$= 106$

$g(n) = n^2$

$= (1)^2$

$= 1$

$c \cdot g(n) = 106(1)$

$= 106$

$f(n) = c \cdot g(n)$ is true.

Case 2 : If $n = 2$ then

$f(n) = 100n^2 + 4n + 2$

$= 100(2)^2 + 4(2) + 2$

$= 400 + 8 + 2$

$= 410$

$c \cdot g(n) = 106\ (2)^2$

$= 106(4)$

$= 424.$

$f(n) < c \cdot g(n)$ is true.

From the above cases, $f(n) \leq c \cdot g(n)$ remains true for $n \geq 1$.

Hence $100n^2 + 4n + 2 = 0(n^2)$ is CORRECT for $n \geq 1$ and $c = 106$.

But for $n = 0$, $100n^2 + 4n + 2 \neq 0(n^2)$

Example 2.2.2 *Prove that : $f(n) = a_m n^m + \ldots + a_1 n + a_0$, then*

$f(n) = O(n^m)$

SPPU : Dec.-06, 08, Marks 6, May-08, 13, Marks 8, Aug.-15 (In Sem.), Marks 4

Solution : Let,

$$f(n) = a_m n^m + a_{m-1} n^{m-1} + a_{m-2} n^{m-2} + \ldots + a_0 \quad \ldots(1)$$

If we treate a_m^i as a constant then the equation (1) becomes

$$f(n) = A \sum_{i=0}^{m} a^m \quad \ldots(2)$$

where A is $a_0^0 + a_1^1 + a_2^2 + \ldots$

If equation (2) is

$$f(n) = A \left[\sum_{i=0}^{n} 1 \right] = A [1 + 1 + 1 + \ldots + 1]$$

$$= A(n^1) = A \cdot O(n^1)$$

If equation (2) is

$$f(n) = A \sum_{i=0}^{n} n = A [1 + 2 + 3 + \ldots n]$$

$$= A(n^2) = A \cdot O(n^2)$$

If equation (2) is

$$f(n) = A \sum_{i=0}^{n} n^2 = A\, O(n^3)$$

Thus $$f(n) = A \sum_{i=0}^{m} n = A\, O(n^m)$$

Thus neglecting the constant term we will have,

$$f(n) = O(n^m)$$

Example 2.2.3 *Interpret the following equations :*

i) $2n^2 + \Theta(n) = \Theta(n^2)$

ii) $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

SPPU : May-09, Marks 4

Solution : **i)** $2n^2 + \theta(n) = \Theta(n^2)$

$f(n) = \Theta(g(n))$ is true when

$c_2 g(n) \leq f(n) \leq c_1 g(n)$ for all $n \geq n_0$ and where c_1 and c_2 are some positive constants.

Let, L.H.S $= 2^n + \Theta(n)$ and R.H.S $= \Theta(n^2)$

We can rewrite it as

$$2^n = \Theta(n^2) - \Theta(n)$$

$$2^n = \Theta(n^2 - n)$$

$\therefore$ $f(n) = 2^n$

$g(n) = n^2 - n$

Now if we have $c_2 = 1$ and $c_1 = 2$ then for $n > = 2$

We get $2^n = \Theta(n^2 - n)$ to be CORRECT. It is illustrated as follows -

$n = 2$, $c_2 = 1$, $c_1 = 2$

$$c_2 \cdot g(n) \leq f(n) \leq c_1 g(n)$$

i.e. $$1(n^2 - n) \leq 2^n \leq 2 \cdot (n^2 - n)$$

$$1(2^2 - 2) \leq 2^1 \leq 2(2^2 - 2)$$

$2 \leq 2 \leq 4$ is true.

$n = 3$, $c_2 = 1$, $c_1 = 2$

$$c_2(n^2 - n) \leq 2^n \leq c_1(n^2 - n)$$

$$1(3^2 - 3) \leq 2^3 \leq 2(3^2 - 3)$$

$6 \leq 8 \leq 12$ is true

Thus for $n \geq 2$ and $c_1 = 2$ and $c_2 = 1$ the $2^n + \Theta(n) = \Theta(n^2)$ is true.

ii) $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

If we simplify above equation, we get

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

$\therefore$ $3n + 1 = \Theta(n)$

Here $f(n) = 3n + 1$

$g(n) = n$

If we have $c_2 = 1$ and $c_1 = 4$ then for $n \geq 1$ We get $3n + 1 = \Theta(n)$ to be CORRECT. It is illustracted as follows -

If $n = 1$, $c_1 = 4$ $c_2 = 1$,

Let, $c_2\, g(n) \leq f(n) \leq c_1 g(n)$

i.e. $c_2(n) \leq (3n+1) \leq c_1(n)$

$1(1) \leq (3\times1+1) \leq 4(1)$

$1 \leq 4 \leq 4$ is true

If $n = 2$, $c_1 = 4$ $c_2 = 1$,

Let, $c_2 \cdot g(n) \leq f(n) \leq c_1 g(n)$

$1(2) \leq (3\times2+1) \leq 4(2)$

$2 \leq 7 \leq 8$ is true

Thus for $n \geq 1$ having $c_2 = 1$ and $c_1 = 4$ as constant values the given equation $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ is true.

2.2.5 Little Oh and Little Omega Notation

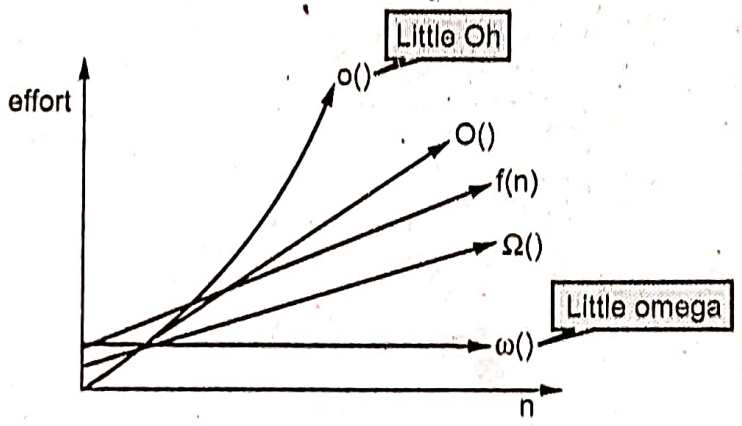


Fig. 2.2.4

$o(f) = O(f)\text{-}\Theta(f)$

$\omega(f) = \Omega(f)\text{-}\Theta(f)$

$$g \in o(f) \text{ if } \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

$$g \in \omega(f) \text{ if } \lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty$$

In other words

$o(f) \equiv \{g | \forall c \in N, \exists n0 \in N \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n0\}$

$o(f) \equiv \{g | \forall c \in N, \exists n0 \in N \text{ such that } 0 \leq cf(n) \leq g(n) \text{ for all } n \geq n0\}$

Relationship between all the asymptotic notations and small oh

"g(n) is O(f(n))" basically means that, f(n) describes the upper bound for g(n)

"g(n) is Ω (f(n))" basically means that f(n) describes the lower bound for g(n)

"g(n) is Θ (f(n))" basically means that f(n) describes the exact bound for g(n)

"g(n) is o(f(n))" basically means that g(n) is the upper bound for g(n) but that g(n) can never be equal to g(n)

Another way of saying

"g(n) is O(f(n))" basically means growth rate of g(n)<=growth rate of f(n)

"g(n) is Ω(f(n))" basically means growth rate of g(n)>=growth rate of f(n)

"g(n) is Θ(f(n))" basically means growth rate of g(n)==growth rate of f(n)

"g(n) is o(f(n))" basically means growth rate of g(n)< growth rate of f(n)

Review Questions

1. *Define the asymptotic notations :*
   i) Ω ii) ω iii) Θ
   SPPU : May-09, Marks 6
2. *Define asymptotic notations. Explain their significance in analyzing algorithms.*
   SPPU : May-12, 14, Marks 5
3. *Define the following :*
   i) Big ''oh'' ii) Theta iii) Omega.
   SPPU : Dec.-12, Marks 6
4. *Explain with example the notations Big-O and Omega.*
   SPPU : Dec.-14, Marks 4
5. *Explain all three asymptotic notations with 2 examples of each.*
   SPPU : Aug.-15, Marks 6, Dec.-19, Marks 4
6. *Explain Big Oh(O), Omega(Ω) and Theta(Θ) notation in detail along with suitable examples.*
   SPPU : May-17, Marks 6
7. *Explain asymptotic notations with example.*
   SPPU : May-18, Marks 8
8. *Define asymptotic notation. What is their significance in anlyzing algorithms ? Explain Big Oh, Omega and Theta notations.*
   SPPU : May-19, Marks 8

2.3 Best Case, Worst Case, Average Case

SPPU : Dec.-12, 15, Oct.-16, May-09, Marks 6

If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called **best case** time complexity.

For example : While searching a particular element by using sequential search we get the desired element at first place itself then it is called best case time complexity.

If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called **worst case** time complexity.

For example : While searching an element by using linear searching method if desired element is placed at the end of the list then we get worst time complexity.

The time complexity that we get for certain set of inputs is as a average same. Then for corresponding input such a time complexity is called **average case** time complexity.

Consider the following algorithm

```
Algorithm Seq_search(X[0 ... n-1 ],key)
// Problem Description: This algorithm is for searching the
//key element from an array X[0...n-1] sequentially.
//Input: An array X[0...n-1] and search key
//Output: Returns the index of X where key value is present
for i ← 0to n-1 do
 if(X[i]=key)then
  return i
```

Best case time complexity

Best case time complexity is a time complexity when an algorithm runs for short time. In above searching algorithm the element **key** is searched from the list of n elements. If the **key** element is present at first location in the list(X[0...n–1]) then algorithm run for a very short time and thereby we will get the best case time complexity. We can denote the best case time complexity as

$$C_{best} = 1$$

Worst case time complexity

Worst case time complexity is a time complexity when algorithm runs for a longest time. In above searching algorithm the element **key** is searched from the list of n elements. If the **key** element is present at n^{th} location then clearly the algorithm will run for longest time and thereby we will get the worst case time complexity. We can denote the worst case time complexity as

$$C_{worst} = n$$

The algorithm guarantees that for any instance of input which is of size n, the running time will not exceed $C_{worst}(n)$. Hence the worst case time complexity gives important information about the efficiency of algorithm.

Average case time complexity

This type of complexity gives information about the behaviour of an algorithm on specific or random input. Let us understand some terminologies that are required for computing average case time complexity.

Let the algorithm is for sequential search and P be a probability of getting successful search.

n is the total number of elements in the list.

The first match of the element will occur at i^{th} location. Hence probability of occurring first match is P/n for every i^{th} element.

The probability of getting unsuccessful search is (1 – P).

Now, we can find average case time complexity $C_{avg}(n)$ as -

$C_{avg}(n)$ = Probability of successful search (for elements 1 to n in the list)

+ Probability of unsuccessful search

$$C_{avg}(n) = \left[1 \cdot \frac{P}{n} + 2 \cdot \frac{P}{n} + \ldots + i \cdot \frac{P}{n}\right] + n \cdot (1 - P)$$

There may be n elements at which chances of 'not getting element' are possible.

$$= \frac{P}{n} [1 + 2 + \ldots + i \ldots n] + n (1 - P)$$

$$= \frac{P}{n} \frac{n (n+1)}{2} + n (1 - P)$$

$$C_{avg}(n) = \frac{P (n+1)}{2} + n (1 - P)$$

Thus we can obtain the general formula for computing average case time complexity.

Suppose if P = 0 that means there is no successful search i.e. we have scanned the entire list of n elements and still we do not found the desired element in the list then in such a situation,

$$C_{avg}(n) = 0(n + 1)/2 + n (1 - 0)$$

$$C_{avg}(n) = n$$

Thus the average case running time complexity becomes equal to n.

Suppose if P = 1 i.e. we get a successful search then

$$C_{avg}(n) = 1 (n + 1)/2 + n (1 - 1)$$

$$C_{avg}(n) = (n + 1)/2$$

That means the algorithm scans about half of the elements from the list.

For calculating average case time complexity we have to consider probability of getting success of the operation. And any operation in the algorithm is heavily dependent on input elements. Thus computing average case time complexity is difficult than computing worst case and best case time complexities.

Example 2.3.1 *Calculate the worst case time complexity of* $f(n) = 6n(n^3 - n) + 9n$ *using the running time complexity.*

Solution : While finding the worst case analysis we calculate **upper bound** on running time of an algorithm. The upper bound is calculated using big oh notation.

Let $f(n) = 6n(n^3 - n) + 9n$

i.e. $f(n) = 6n^4 - 6n^2 + 9n$

We will find $f(n) \leq c * g(n)$.

For f(n), Let n = 1.

$6(1)^4 - 6(1)^2 + 9(1)$

$= 6 - 6 + 9$

$= 9$

$c \geq 0$ and $g(n) = 9n^4$

$f(n) \leq c * g(n)$.

$6n^4 - 6n^2 + 9n \leq 9n^4$

Time complexity is $O(n^4)$.

Consider set of the data, if we are having quick sort and merge sort you select for the following instance of the data to sort it in

SPPU : Oct.-16 (In Sem), Marks 6

Solution :

- The quicksort's behaviour changes on worst case and average case. If the set of data is already sorted in ascending order or descending order then this causes worst case behaviour of quicksort. This is because - the pivot element which we choose in this case is always a maximum element or minimum element. Due to which there will be an increase in number of comparisons.
- However, if data are in ascending or descending order then merge sort becomes faster.
- For set 1 and set 2 the merge sort algorithm will be chosen which results in O(n log n) performance.
- For set 3 the quicksort algorithm will be chosen which results in O(n log n) performance.

Review Questions

1. *What are the basic components that contribute to the space complexity ?*
   SPPU : May-09, Marks 4
2. *Explain the different ways of measuring the running time of an algorithm.*
   SPPU : Dec.-12, Marks 6
3. *Explain the best case, worst case and average case complexity of an algorithm with one example.*
   SPPU : Dec.-18, Marks 8

2.4 Growth Rate

Measuring the performance of an algorithm in relation with the input size n is called order of growth. For example, the order of growth for varying input size of n is as given below.

n	log n	n log n	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296

Table 2.4.1 Order of growth

We will plot the graph for these values. (Refer Fig. 2.4.1 on next page)

From the above drawn graph it is clear that the logarithmic function is the slowest growing function. And the exponential function 2^n is fastest and grows rapidly with varying input size n. The exponential function gives huge values even for small input n. For instance : For the value of n=16 we get $2^{16} = 65536$.

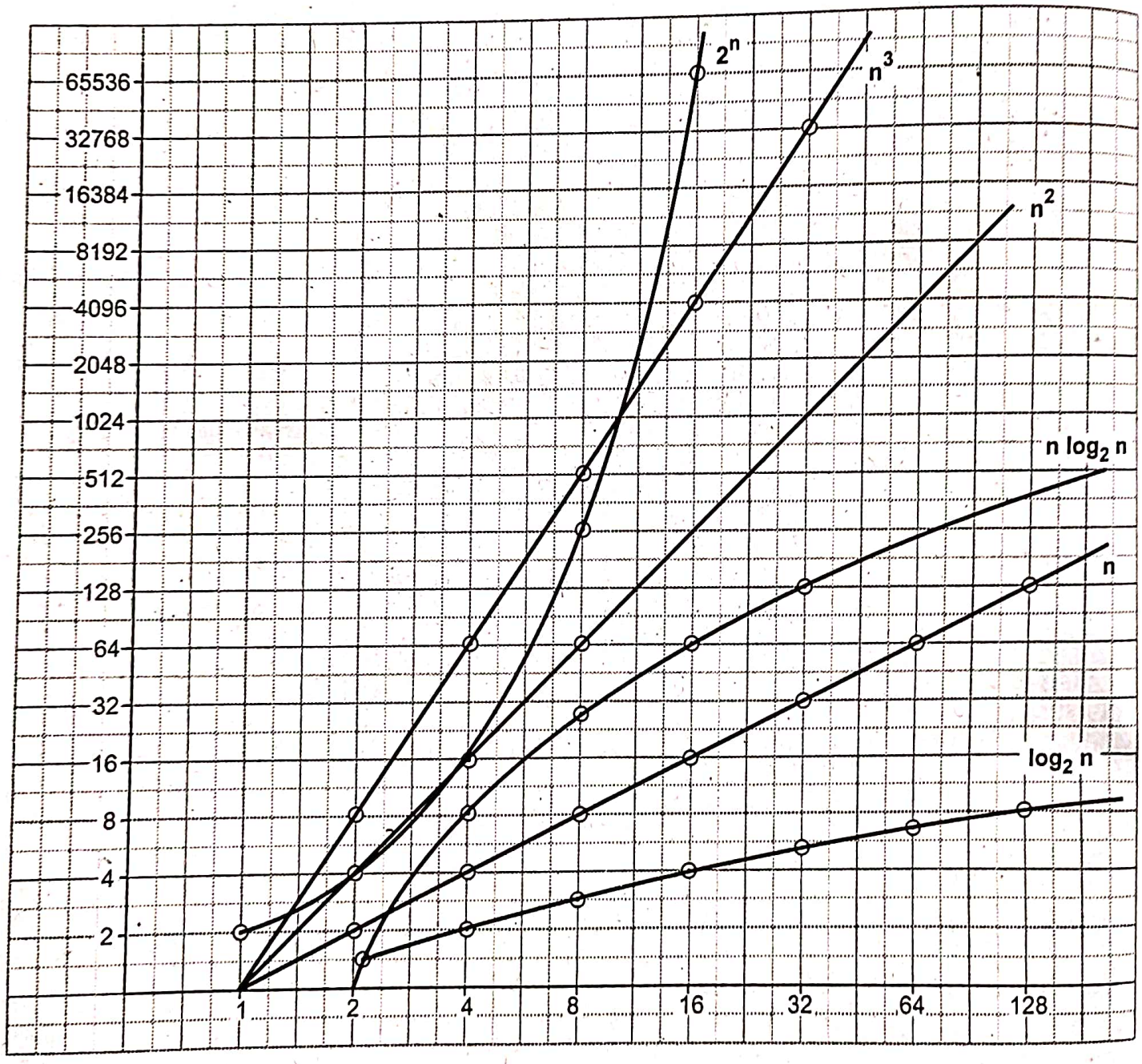


Fig. 2.4.1 Rate of growth of common computing time function

Part II : Complexity Theory

2.5 Polynomial and Non-polynomial Problems

- Polynomial time algorithms are those algorithms that take polynomial time.
- For example - Linear search O(n), binary search O(logn), insertion sort $O(n^2)$, matrix multiplication $O(n^3)$.
- Exponential time algorithms are those algorithms that take exponential time.
- For example - Knapsack problem $O(2^n)$, traveling salesperson problem $O(2^n)$, sum of subset problem (2^n), graph coloring problem (2^n).

2.6 P-class Problems and NP-class of Problems

2.6.1 Polynomial and Non-polynomial 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). The polynomial time is nothing but the time expressed in terms of polnomial.
  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".
- The NP class problems are those problems that can be **solved in non-deterministic polynomial time** but can be **verified in polynomial time.**

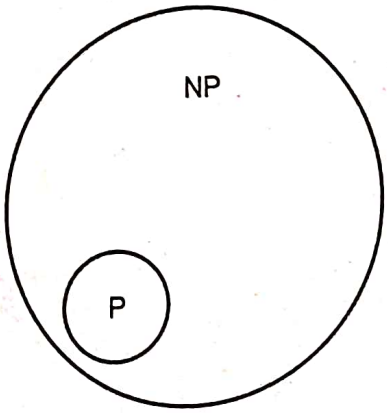


Fig. 2.6.1 The P and NP class

For example - Consider an example of sudo-ku game. In order to solve this entire puzzle, the algorithm would have to check each 3 x 3 matrix to see which numbers are missing, then each row, then each column and then make sure there are no repetitions of any digit from 0 - 9. This becomes more complex because the number of digits that are missing is inconsistent in each row, column and matrix. Solving this problem is not possible in polynomial time. But once we get the solution, then verifying this solution is possible within very less time. Thus verification of solution of this problem is possible in polynomial time.

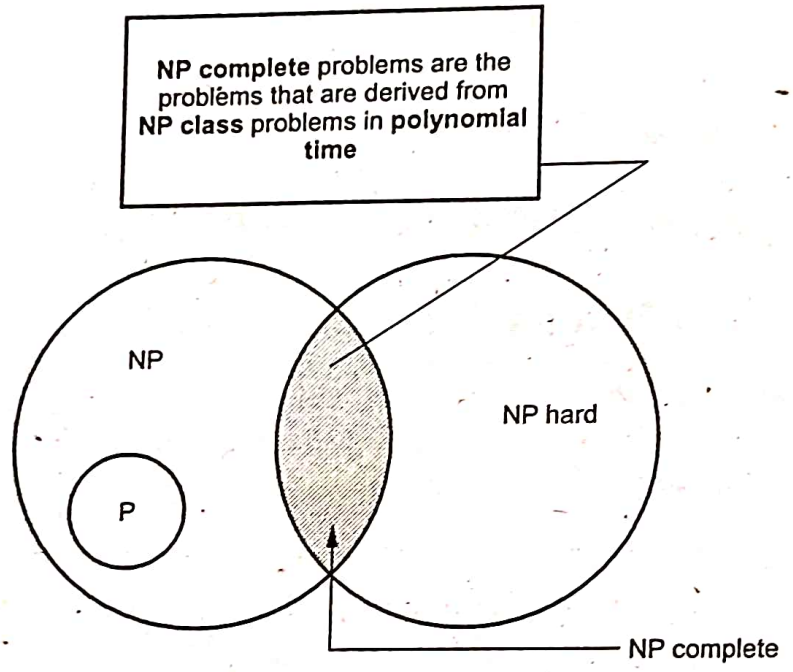


Fig. 2.6.2 P, NP, NP complete and NP hard class problems

- The NP class problems can be further categorized into NP-complete and NP hard problems.
- 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 cannot be NP-complete.

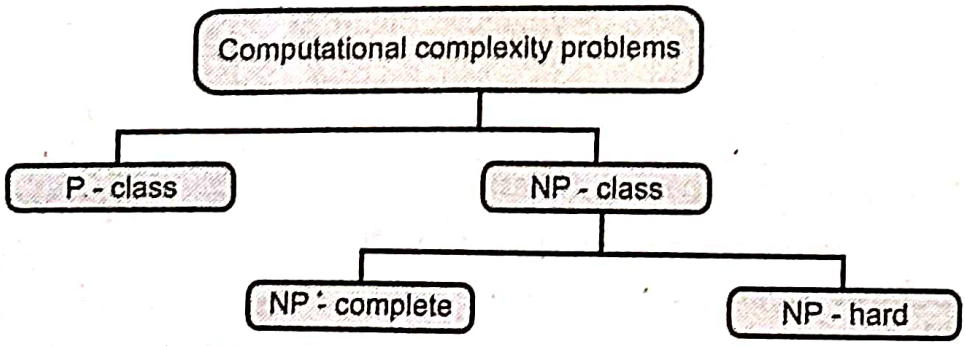


Fig. 2.6.3 Class of computational complexity

- The NP complete problems are intersection of NP class problems and NP hard problems.
- Normally the NP complete problems are decision problems(For example - is a graph Euler graph or not).
- The NP hard problems are optimization problems (For example - shortest path problem).
- **Computational complexity** - The computational problems is an infinite collection of instances with a solution for every instance.
  In computational complexity the solution to the problem can be "yes" or "no" type. Such type of problems are called **decision problem.** The computational problems can be **function problem**. The function problem is a computational problem where single output is expected for every input. The output of such type of problems is more complex than the decision problem.
  The computational problems also consists of a class of problems, whose output can be obtained in polynomial time.
- **Complexity classes** - The complexity classes is a set of problems of related complexity. It includes function problems, P classes, NP classes, optimization problem.
- **Intractability** - Problems that can be solved by taking a long time for their solutions is known as intractable problems.
  If NP is not same as P then NP complete problems are called intractable problems.
- The P class problems are considered as tractable problems.

2.6.1.1 Example of P Class Problem

Kruskal's algorithm : In Kruskal's algorithm the minimum weight is obtained. In this algorithm also the circuit should not be formed. Each time the edge of minimum weight has to be selected, from the graph. It is not necessary in this algorithm to have edges of minimum weights to be adjacent. Let us solve one example by Kruskal's algorithm.

Example :

Find the minimum spanning tree for the following figure using Kruskal's algorithm.

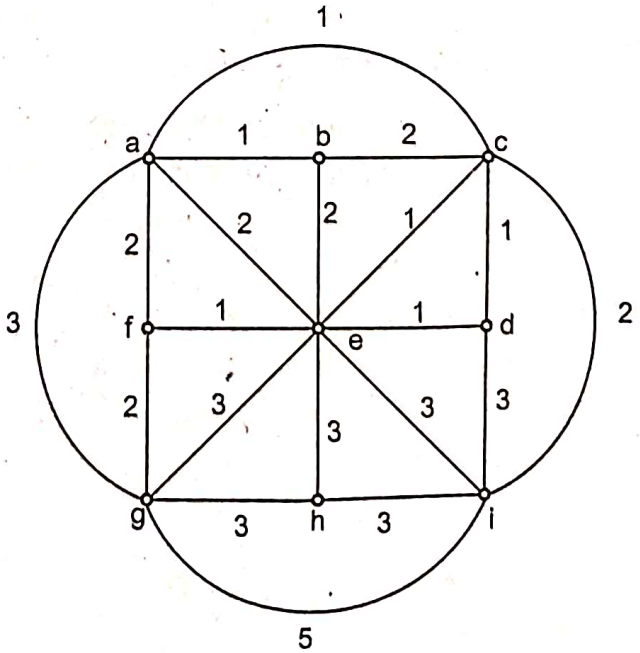


Fig. 2.6.4

In Kruskal's algorithm, we will start with some vertex and will cover all the vertices with minimum weight. The vertices need not be adjacent.

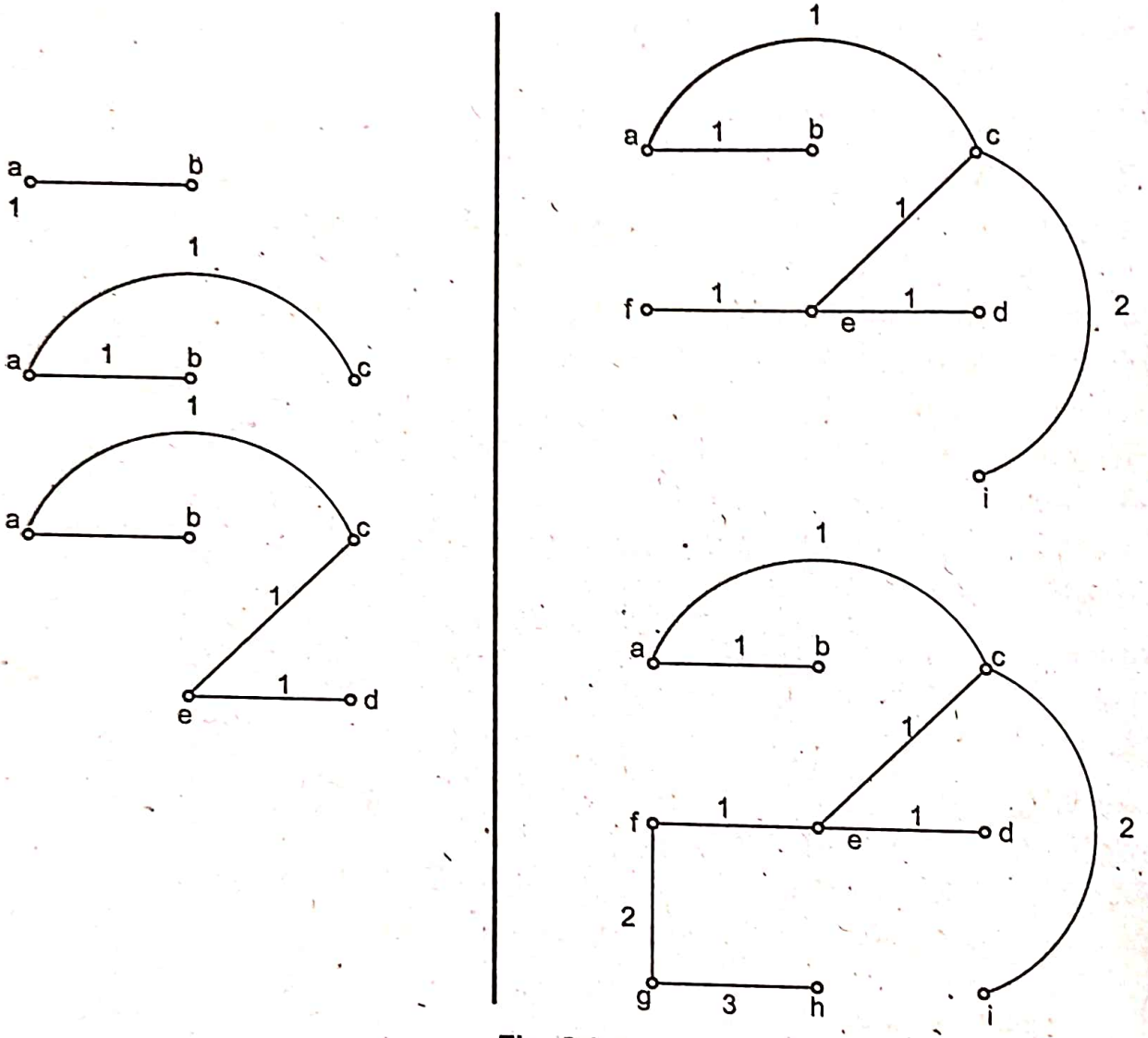


Fig. 2.6.5

2.6.1.2 Example of NP Class Problem

Travelling Salesman's Problem (TSP) : This problem can be stated as " Given a set of cities and cost to travel between each pair of cities, determine whether there is a path that visits every city once and returns to the first city. Such that the cost travelled is less".

For example : Refer Fig. 2.6.6.

The tour path will be **a-b-d-e-c-a** and total cost of tour will be **16.**

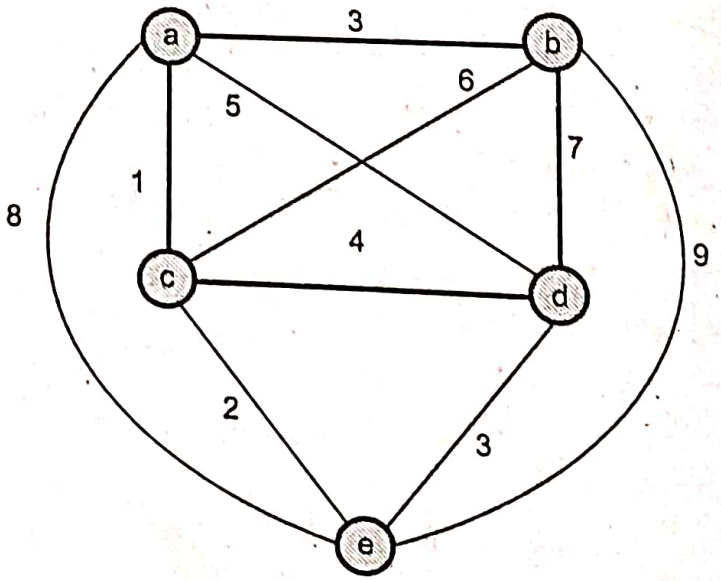


Fig. 2.6.6

This problem is NP problem as there may exist some path with shortest distance between the cities. If you get the solution by applying certain algorithm then Travelling Salesman problem is **NP Complete Problem**. If we get no solution at all by applying an algorithm then the travelling salesman problem belongs to **NP hard class**.

2.6.1.3 Difference between P and NP Class Problems

P class problems	NP class problems
An algorithm in which for given input the definite output gets generated is called Polynomial time algorithm(P class)	An algorithm is called non deterministically polynomial time algorithm (NP class) when for given input there are more than one paths that the algorithm can follow. Due to which one can not determine which path is to be followed after particular stage.
All the P class problems are basically deterministic.	All the NP class problems are basically non deterministic.
Every problem which is a P class is also in NP class.	Every problem which is in NP is not the P class problem.
P class problems can be solved efficiently.	NP class problems can not be solved efficiently as efficiently as P class problems.
Examples : Binary search, bubble sort	**Examples :** Knapsack problem, traveling salesperson problem.

Example 2.6.1 *List three problems that have polynomial time algorithms. Justify your answer.*

Solution : The problems that can be solved in polynomial time are called P - class problems. For example -

1. **Binary search -** In searching an element using binary search method, the list is simply divided at the mid and either left or right sublist is searched for key element. This process is carried out in O(logn).
2. **Evaluation of polynomial -** In a polynomial evaluation we make out the summation of each term of polynomial. The evaluated result is simply an integer. This process is carried out in O(n) time.
3. **Sorting a list -** The elements in a list can be arranged either in ascending or descending order. This procedure is carried out in O(nlogn) time.

This shows that all the above problems can be solved in polynomial time.

Example 2.6.2 *Show that both P and NP are closed under the operation union, intersection, concatenation and kleen closure (*).}* **SPPU : Dec.-12, Marks 10**

Solution : Assume L_1, $L_2 \in$ NP. That means there exists machines M_1 and M_2 that M_1 decides L_1 in nondeterministic time $O(n^p)$ and M_2 decides L_2 in nondeterministic time $O(n^q)$.

1. Union

- Let, M be a machine that gets w as input.
- Run M_1 for input w. If M_1 accepted then accept.
- Else run M_2 for input w. If M_2 accepted then accept.
- Otherwise reject.

Thus the longest branch in any computation tree on input w of length n is $O(n^{\max\{p,q\}})$. Thus M is polynomial time nondeterministic decider for $L_1 \cup L_2$.

2. Intersection

- Let, M be a machine that gets w as input.
- Run M_1 on w. If M_1 rejected then reject.
- Else run M_2 for input w. If M_2 rejected then reject.
- Otherwise accept.

Thus the longest branch in any computation tree on input w of length n is $O(n^{\max\{p,q\}})$. Thus M is polynomial time nondeterministic decider for $L_1 \cap L_2$.

3. Concatenation

- Let, M be a machine that gets input w.
- Split input w nondeterministically as w_1, w_2 such that $w = w_1 w_2$.
- Run M_1 for w_1. If M_1 rejected then reject.
- Else run M_2 for w_2. If M_2 rejected then reject.
- Else accept.

Thus the longest branch in any computation tree for input w of length n is $O(n^{\max\{p,q\}})$. M is polynomial time nondeterministic decider for $L_1 \circ L_2$.

4. Kleen closure

- If $w = \varepsilon$ then accept.
- Nondeterministically select a number m such that $1 \leq m \leq |w|$.
- Now split w into m pieces such that $w = w_1 w_2 \ldots w_m$.
- For all i such that $1 \leq i \leq m$, run M_1 for w_i. If M_1 rejected then reject.

The first two steps take O(n) time.

The fourth step takes $O(n^q)$. Thus we get M for w which is polynomial time nondeterministic decider for L_1^*.

Review Question

1. *Explain polynomial and non-polynomial problems. Explain its computations complexity.*

SPPU : May-18, Marks 8

2.7 Deterministic and Non-deterministic Algorithms

SPPU : Dec.-06,10,15,16, May-07,08,10,11,14,17, Marks 10

- 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).

Example 2.7.1 *Give the non-deterministic algorithm for sorting elements in non decreasing order.* **SPPU : May-11, Marks 8**

Solution :

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

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.

2.7.1 Nondeterministic Algorithm for 0/1 Knapsack Problem

The *0/1 Knapsack Decision Problem* is to determine, for a given profit P, whether it is possible to load the knapsack so as to keep the total weight no greater than m, while making the total profit at least equal to Pt. This problem has the same parameters as the 0-1 Knapsack Optimization Problem plus the additional parameter Pt.

The non deterministic algorithm for 0/1 knapsack problem is as given below -

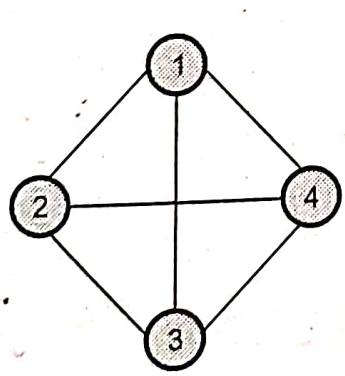
```
Algorithm Non_DKnaps()
//p[1:n] is a profit each object i where 1 ≤ i ≤ n
//w[1:n] is the weight of each object i where 1 ≤ i ≤ n
//Profit and Wt represent the current total profit and weight
// m is the capacity of knapsack
{
//initially no item is selected
```

```
 Wt:=0;
Profit:=0;

// guessing stage
      for i:=1 to n do
      {
            x[i]:=choose(0,1);
            Wt:=Wt+x[i]*w[i];
            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
}
```

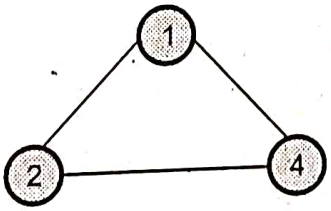
2.7.2 Clique Problem

Clique is nothing but a maximal complete subgraph of a graph G. The graph G is a set of (V, E) where V is a set of vertices and E is a set of edges. The size of Clique is number of vertices present in it.

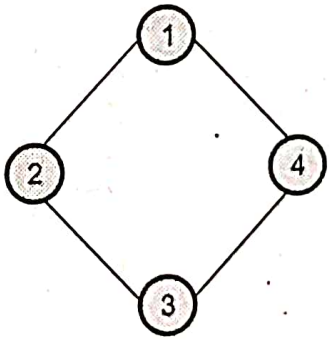
For example :



Graph G



Clique size = 3



Clique size = 4

Note that Clique of size = 3 and 4 are complete subgraph of graph G.

Max Clique Problem - It is an optimization problem in which the largest size Clique is to be obtained.

In the decision problem of Clique, we have to determine whether G has a Clique of size at least k for given k.

Let, DClique (G, k) is a deterministic decision algorithm for determining whether graph G has Clique or not of size atleast k. Then we have to apply DClique for $k = n, n-1, n-2, \ldots$ until the output is 1. The max Clique can be obtained in time $\leq n \cdot F(n)$ where F(n) is the time complexity of DClique. If Clique decision problem is solved in **polynomial time** then **max Clique** problem can be solved in **polynomial time.**

Non Deterministic algorithm for Clique

```
Algorithm NonDClique(G,n,k)
{
    S:=0
    for i:=1 to k do
    {
        t:=Choice(1,n);
        if t is from set S
            Failure();
        Add t to set S
    }
//Now S contains k distinct vertex indices
for all pairs(i,j) i and j are from set S and i!=j do
    if (i,j) is not an edge of graph G then
        Failure();
Success();
}
```

Review Questions

1. *Consider the following search algorithm :*
   j = any value between 1 to n
   If (a[j] = x) then
   print "Success";
   else
   print "Fails"
   Is this algorithm non-deterministic ? Justify your answer.
   SPPU : Dec.-06, Marks 6; Dec.-10, Marks 8
2. *What is the basic difference between deterministic and non-deterministic algorithms ?*
   SPPU : May-07, Marks 2, May-08, Marks 4
3. *Write a non-deterministic Knapsack algorithm.* **SPPU : May-07, 10, Marks 8**
4. *Comment on - "The time required by non-deterministic algorithm performing on any given input is the minimum number of steps needed to reach a successful completion if there exists a sequence of choices leading to such a completion".* **SPPU : May-07, Marks 2**
5. *Write non-deterministic algorithm for searching an item in an array. What is its complexity ?*
   SPPU : May-11, Marks 8

6. *What is a deterministic and non-deterministic algorithm ? Write a non-deterministic algorithm for searching element.*
SPPU : May-14, Marks 8
7. *Write non deterministic algorithm for Clique decision problem.*
SPPU : Dec.-15 (End Sem.), Marks 8
8. *What is 0-1 Knapsack problem ? Explain the algorithm as deterministic and non-deterministic versions.*
SPPU : Dec.-16, (End Sem), Marks 10
9. *Explain non-determininstic clique problem along iwth alogorithm.*
SPPU : May-17, (End Sem), Marks 8
10. *Give and explain non-deterministic sorting algorithm.*
SPPU : May-17, (End Sem), Marks 8
11. *What are deterministic and non-deterministic algorithms ? Explain with example.*
SPPU : Dec.-19, Marks 8

2.8 Polynomial Problem Reduction

To prove whether particular problem is NP complete or not we use polynomial time reducibility. That means if

$A \xrightarrow{Poly} B$ and $B \xrightarrow{Poly} C$ then $A \xrightarrow{Poly} C$.

The **reduction** is an important task in NP completeness proofs. This can be illustrated by Fig. 2.8.1. (See Fig. 2.8.1 on next page).

Various types of reductions are

Local replacement - In this reduction A→B by dividing input to A in the form of components and then these components can be converted to components of B.

Component design - In this reduction A→B by building special component for input B that enforce properties required by A.

- If L_1 and L_2 are two problems, then problem L_1 reduces to L_2 which can be denoted as $L_1 \propto L_2$ if and only if there is a way to solve L_1 by deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time. It means that - if we have polynomial time algorithm for L_2, then we can solve L_1 in polynomial time.
- We can also state that if

  $L_1 \propto L_2$ and $L_2 \propto L_3$

  then $L_1 \propto L_3$
- Two problems P and Q are said polynomially equivalent if and only if $P \propto Q$ and $Q \propto P$.

If problem P_1 is NP-complete and there is polynomial time reduction of P_1 to P_2 then P_2 is NP-complete.

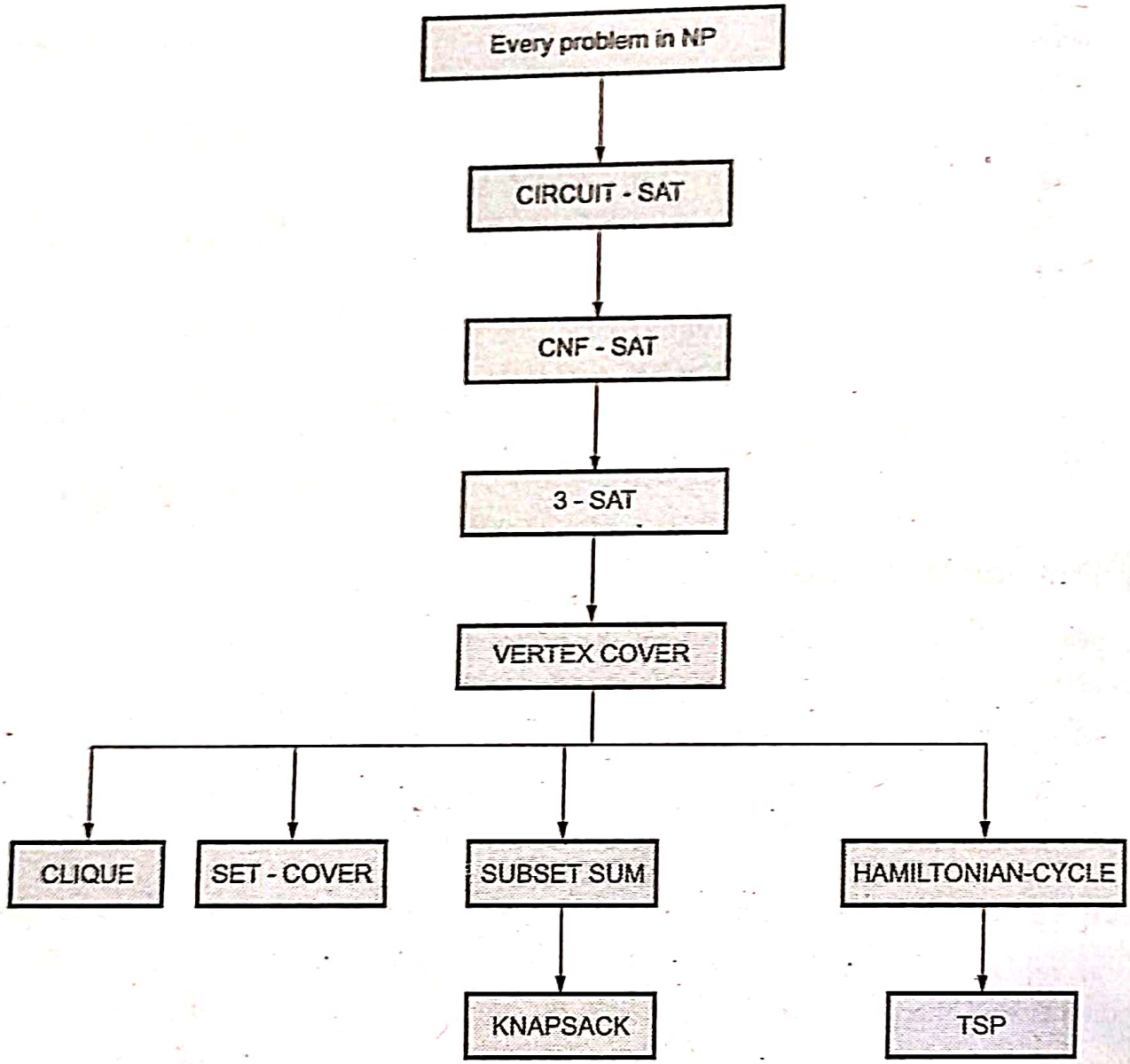


Fig. 2.8.1 Reduction in NP completeness

2.9 NP Complete Problems

SPPU : May-10,14,16,17, Dec.-15,16, Marks 8

In this section we will discuss two problems namely Vertex cover and the 3SAT problem which are actually NP Complete problems. Their proof of NP completeness is based on **reduction technique**. That means there are some problems which are already proved as NP Complete problems and using these problems we will prove that the vertex cover and the 3SAT problems are NP Complete problems.

Steps for proving NP - complete :

Consider that, we have to prove that B is in NP -

Step 1 : Select an NP complete language say A.

Step 2 : Construct a function f that maps the members of A to members of B.

Step 3 : Show that x is in A if and only if f(x) is in B.

Step 4 : Now show that the function f can be computed in **polynomial time.**

Step 5 : This if A is NP complete and it can be reduced to B in polynomial time, then B comes out to be NP complete.

Prove that SAT problem is NP complete

Proof :

1) SAT is NP.

2) Circuit SAT or (C-SAT) reduces to SAT. The reduction function f is as follows -

i) For every input wire add a new variable.

ii) For every output wire add a new variable.

iii) An equation is prepared for each gate.

iv) These sets of equations are separated by $\wedge$ values and adding final **output variable** at the end.

This transformation can be done in linear time or **polynomial time.** For instance - Consider following circuit for C-SAT problem.

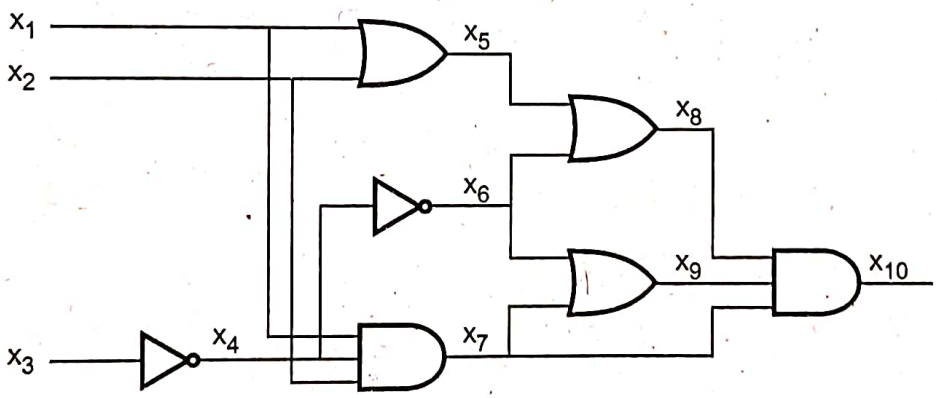


Fig. 2.9.1

$$x_5 = x_1 \vee x_2$$

$$x_4 = \neg x_3$$

$$x_6 = \neg x_4$$

$$x_7 = x_1 \wedge x_2 \wedge x_4$$

$$x_8 = x_5 \vee x_6$$

$$x_9 = x_6 \vee x_7$$

$$x_{10} = x_7 \wedge x_8 \wedge x_9$$

$$\therefore \quad f = x_5 \wedge x_4 \wedge x_6 \wedge x_7 \wedge x_8 \wedge x_9 \wedge x_{10}$$

This shows that we can reduce arbitrary instance of circuit-SAT problem to a specialized instance of SAT in polynomial time. And as we know that circuit-SAT is NP complete and reduction of circuit-SAT to SAT is in polynomial time, we must say that SAT is also an NP complete problem.

The 3-SAT problem

A 3-SAT problem is a problem which takes a Boolean formula S with each clause having exactly three literals and check is S is satisfied or not.

Prove that 3-SAT is NP complete

Proof : The language 3-SAT is a restriction of SAT. We replace each clause C that represents the SAT problem to a function f by family of D_C of clauses that represents satisfiability.

For example say

$$C = a \vee b \vee c \vee d \vee e$$

One can simulate this by

$$D_C = (a \vee b \vee x) \wedge (\bar{x} \vee c \vee y) \wedge (\bar{y} \vee d \vee e)$$

where x and y are new variables.

Need to verify :

1) If C is FALSE, then D_C is FALSE; and

2) If C is TRUE, then one can make D_C TRUE.

If f is satisfiable then there is assignment where each clause C is TRUE. This can be extended to make D_C TRUE.

Further if f is evaluated to FALSE, then some clauses say C' must be FALSE and thus corresponding family $D_{C'}$ evaluates to FALSE.

This conversion process can be done in **polynomial time.** Thus we have shown that SAT reduces to 3-SAT in polynomial time. As we know that SAT is a NP complete problem, so we must say that 3-SAT is also NP complete problem.

2.9.1 The 3 SAT Problems

Before understanding the 3 SAT problem we will get introduced with the satisfiability problem.

1. CNF - SAT problem

This problem is based on Boolean formula. The Boolean formula has various Boolean operations such as OR(+), AND (·) and NOT. There are some notations such as $\rightarrow$ (means implies) and $\leftrightarrow$ (means if and only if).

A Boolean formula is in **Conjuctive Normal Form (CNF)** if it is formed as collection of subexpressions. These subexpressions are called **clauses.**

For example

$(\bar{a} + b + d + \bar{g})\ (c + \bar{e})\ (\bar{b} + d + \bar{f} + h)\ (a + c + e + \bar{h})$

This formula evaluates to 1 if b, c, d are 1.

The CNF-SAT is a problem which takes Boolean formula in CNF form and checks whether any assignment is there to Boolean values so that formula evaluates to 1.

2.9.2 Vertex Cover Problem

The vertex cover problem is to find vertex cover of minimum size in a given graph. The word vertex cover means each vertex covers its incident edges. Thus by vertex cover we expect to choose the set of vertices which cover all the edges in a graph.

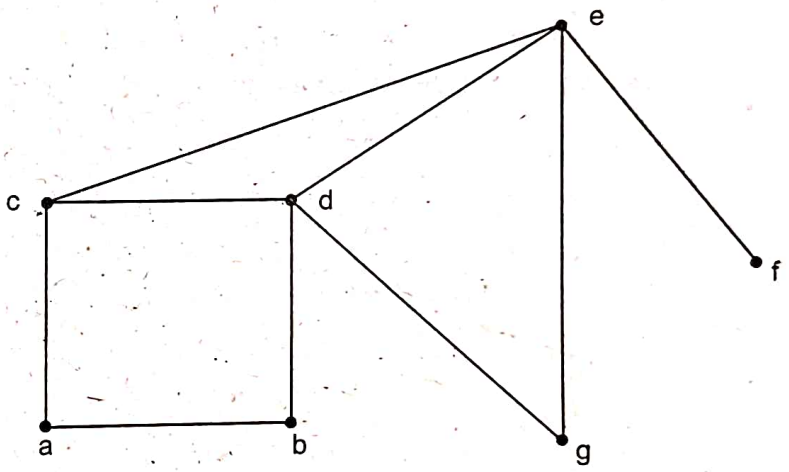


Fig. 2.9.2

For example : Consider a graph G as given below.

Now we will select some arbitrary edge and delete all the incident edges. Repeat this process until all the incident edges get deleted.

Step 1 : Select an edge a-b and delete all the incident edges to vertex a or b.

Step 2 : Select an edge c-e and delete all the incident edges to vertex c or e.

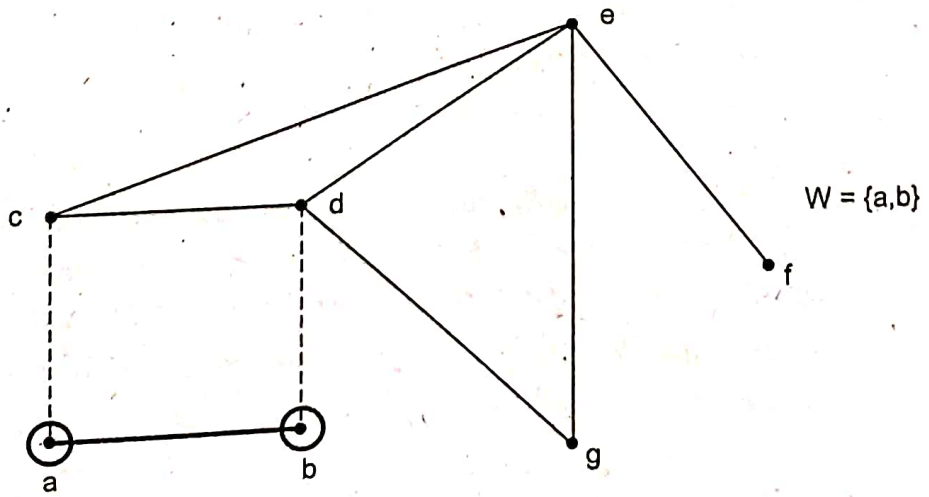


Fig. 2.9.3

Step 3 : Select an edge d-g. All the incident edges are already deleted.

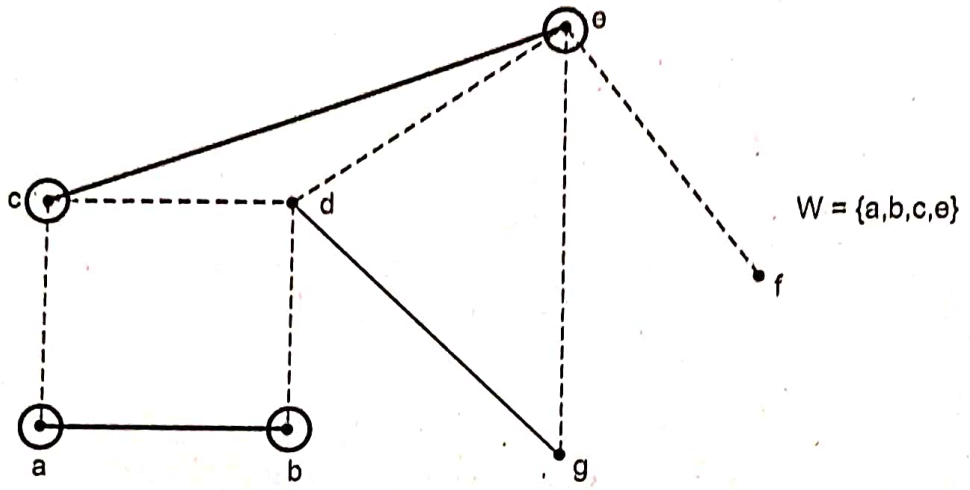


Fig. 2.9.4

Thus we obtain vertex cover as {a, b, c, d, e, g}.

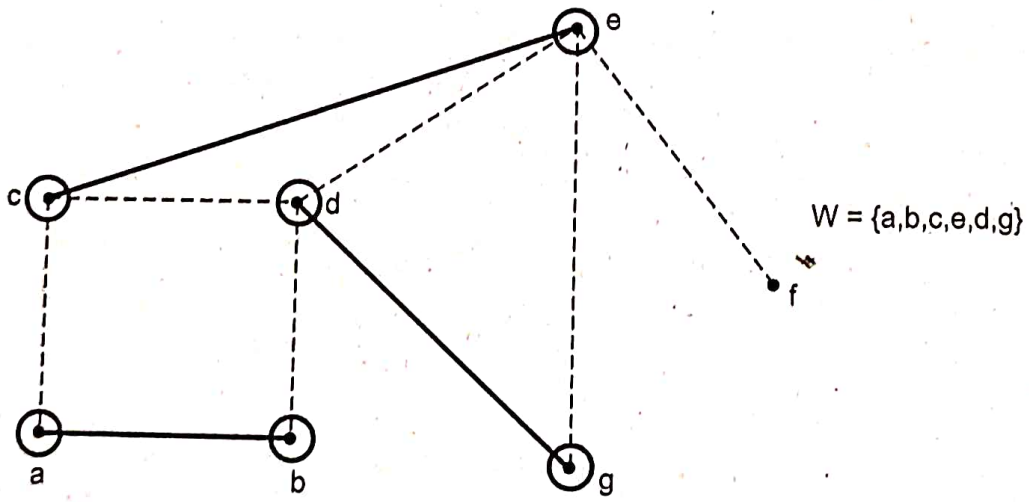


Fig. 2.9.5

Theorem : The vertex cover is NP-complete.

Proof :

To prove that vertex cover is NP-complete we will apply reduction technique. That means reduce 3-SAT to vertex cover. As we know 3-SAT to basically a Boolean expression s containing 3 variables in each clause and value of S is true.

To prove this theorem consider S be some Boolean formula in CNF (conjunctive Normal Form) having 3 variables in each clause no for each variable a we will create,

Fig. 2.9.6

If the clause is a + b + c we will create a triangle (as there are 3 - variables)

Using 3-SAT we will build a graph as follows for given expression.

$(a+b+c)\ (a+b+\bar{c})\ (\bar{a}+c+\bar{b})$

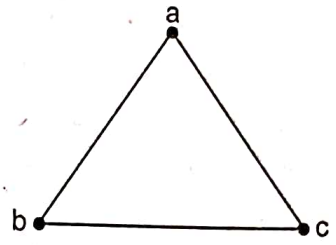


Fig. 2.9.7

The graph has vertex cover of size K = n + 2m, where n is number of variables in 3-SAT, m is number of clauses in CNF, K is total number of vertices that are present in the set of vertex cover.

For a clause (a + b + c) we can build a graph as :

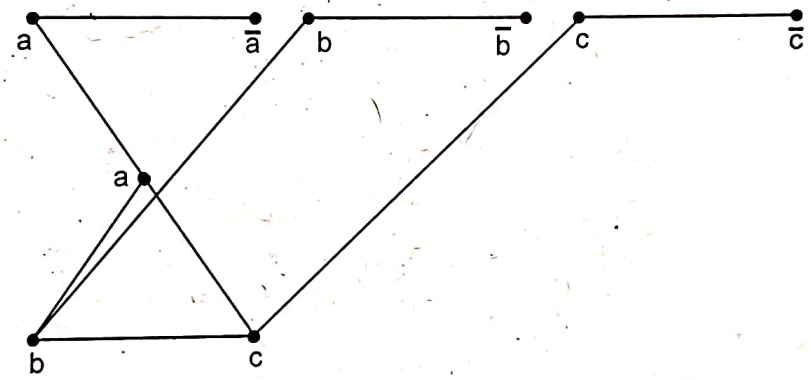


Fig. 2.9.8

Thus for given expression $(a+b+c)\ (a+b+\bar{c})\ (\bar{a}+c+\bar{d})$ we get following graph.

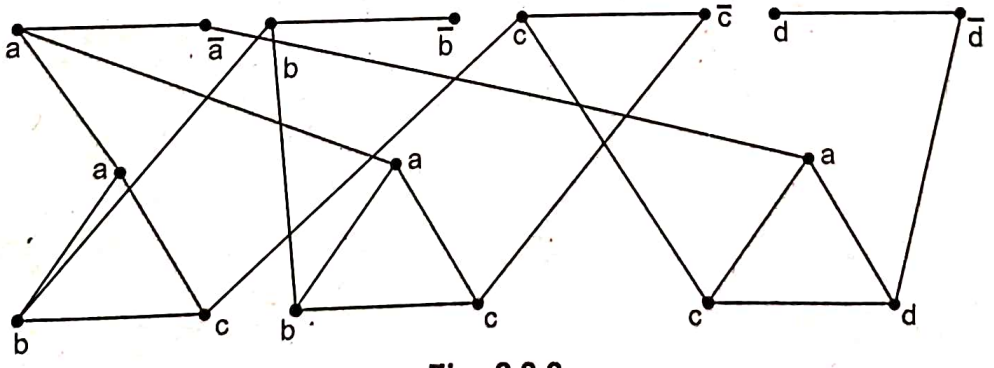


Fig. 2.9.9

Here n = Number of variable = 4

m = Number of clauses = 3

$\therefore$ K = n + 2m = 4 + 2(3) = 10

In vertex cover we get K = 10 vertices which cover all the vertices. The vertex cover is shown by following graph in which the covering vertices are rounded.

Thus K = n + 2m is proved as K = 10. This is how 3-SAT can be reduced to vertex cover. The 3-SAT is NP-complete. Hence vertex is NP-complete is proved.

Thus K = n + 2m is proved as K = 10. This is how 3-SAT can be reduced to vertex cover. The 3-SAT is NP-complete. Hence vertex is NP-complete is proved.

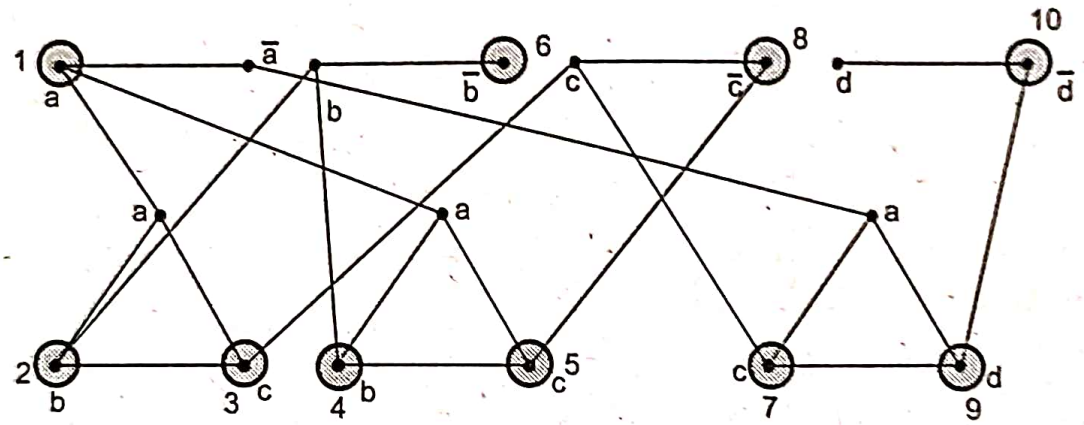


Fig. 2.9.10

Review Questions

1. *Prove that vertex cover problem is NP complete.* **SPPU : May-14, Marks 9**
2. *Prove that vertex cover problem is NP Complete.* **SPPU : Dec.-15 (End Sem.), Marks 8**
3. *What is SAT and 3-SAT problem ? Prove that the 3-SAT problem is NP complete.* **SPPU : May-16 (End Sem), 18, Marks 8**
4. *What NP-complete algorithm ? How do we prove that algorithm is NP complete ? (Give example)* **SPPU : Dec.-16, (End Sem), Marks 6**
5. *Prove that vertex cover problem in NP-complete.* **SPPU : May-17, (End Sem), Marks 8**
6. *What are steps to prove NP completeness of problem ? Prove that vertex cover problem is NP complete.* **SPPU : May-19, Marks 8**
7. *What is Boolean satisfiability problem ? Explain 3SAT problem. Prove 3-SAT is NP complete.* **SPPU : May-19, Marks 8**
8. *State vertex cover problem and prove that vertex cover problem is NP complete.* **SPPU : Dec.-19, Marks 8**

2.10 NP Hard Problem

SPPU : May-18, Marks 8

A problem A is NP-hard if there is an NP-complete problem B, such that B is reducible to A in polynomial time. NP-hard problems are as hard as NP-complete problems. NP-hard problem need not be in NP class.

- 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.

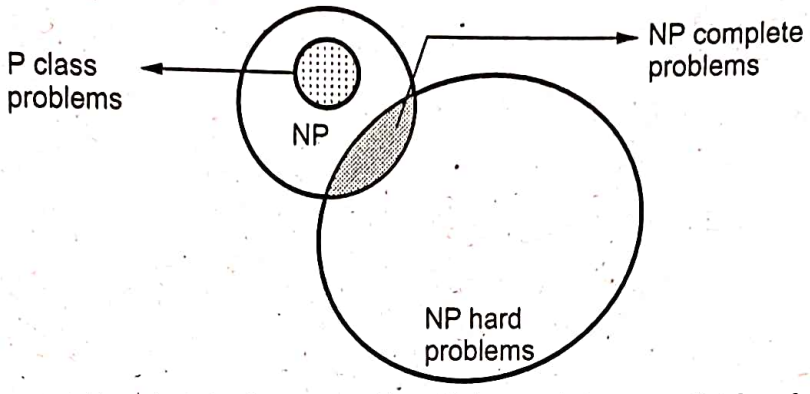


Fig. 2.10.1 Relationship between P, NP, NP-complete and NP-hard problems

- Normally the decision problems are NP-complete but optimization problems are NP-hard. However if problem L_1 is a decision problem and L_2 is optimization problem then it is possible that $L_1 \propto L_2$. 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 ?"

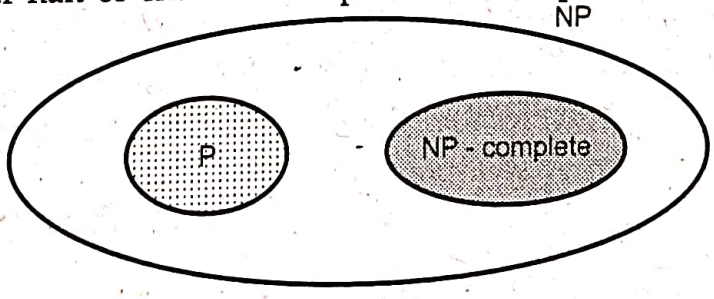


Fig. 2.10.2

Difference between NP hard and NP complete problem

Sr. No.	NP hard	NP complete
1.	NP hard problem (Say A) can be solved if and only if there is NP complete problem (say B) can be reducible into A in polynomial time.	NP complete problems can be solved by deterministic algorithm in polynomial time.
2.	To solve the NP hard problem, it must be in NP class.	To solve NP complete problem, it must be in both NP and NP hard problems.
3.	It is not a decision problem.	It is a decision problem.
4.	For example - Halting problem, Hamiltonian cycle problem, vertex cover problem.	For example - Determine whether a graph has a Hamiltonian cycle, determine whether a Boolean formula is satisfiable or not, circuit-satisfiability problem.

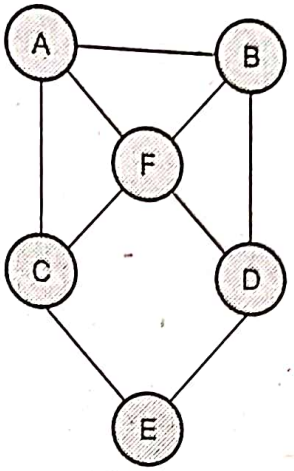
Review Question

1. *Write a short note on NP completeness of algorithm and NP hard.* **SPPU : May-18, Marks 8**

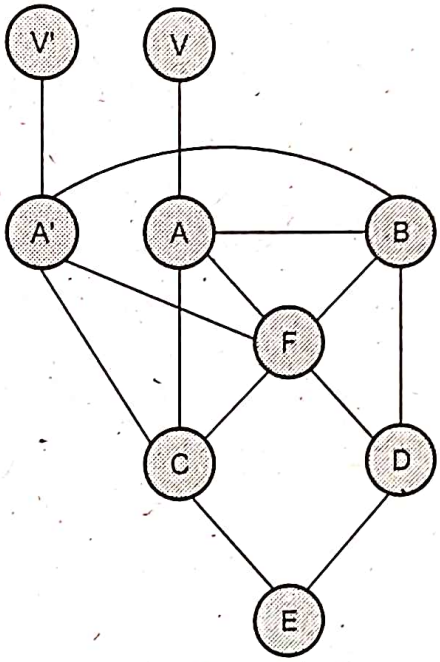
2.11 Hamiltonian Cycle

SPPU : Dec.-19, Marks 8

- Hamiltonian path is a simple open path that contains each vertex in a graph exactly once and if the source vertex and the destination vertex of this Hamiltonian path is the same then it is called Hamiltonian cycle.
- The Hamiltonian path problem is the problem to determine whether a given graph contains a Hamiltonian path.
- To show that this problem is NP-complete we first need to show that it actually belongs to the class NP and then find a known NP-complete problem that can be reduced to Hamiltonian path.
- For a given graph G we can solve Hamiltonian Path by deterministically choosing edges from G that are to be included in the path. Then we traverse the path and make sure that we visit each vertex exactly once. This obviously can be done in polynomial time and hence the problem belongs to NP.
- Now we have to find out an NP complete problem that can be reduced to Hamiltonian path. One such closely related problem is the problem to determine whether the graph contains Hamiltonian cycle (Hamiltonian cycle is basically an Hamiltonian path that begin and end in the same vertex). Hamiltonian cycle is NP complete so we try to reduce this problem to Hamiltonian path.
- For example - Consider a graph G, from which we can construct a graph G'' such that G contains a Hamiltonian cycle if and only if G'' contains Hamiltonian path.
- For instance in Graph G the Hamiltonian cycle is A-B-D-E-C-F-A and G'' contains the Hamiltonian path V- A-B-D-E-C-F-A'-V'.
- Conversely if G'' contains Hamiltonian Path then we can convert it to the Hamiltonian cycle present in G by removing the end points V and V' and mapping A' to A.
- Thus we can show that G contains Hamiltonian cycle if and only if G'' contains Hamiltonian path which concludes the proof that Hamiltonian path is NP complete.
- As every NP complete problem is NP hard problem as well, we conclude that Hamiltonian path problem is also NP hard.



Graph G



Graph G"

Fig. 2.11.1

Review Question

1. *Explain NP hard Hamiltonian cycle problem.* **SPPU : Dec.-19, Marks 8**