UNIT 2 COMPLEXITY

Structure **Page Nos.**

2.0 INTRODUCTION

In the previous unit, we introduced you to the fact that there are a large number of problems which cannot be solved by algorithmic means and discussed a number of issues about such problems.

The advantage of such a study is our becoming aware of the fact that in stead of attempting to write an algorithm for every problem that we are required to solve using a computer, we should first study the *essential nature* of the problem. In case the problem under consideration is not solvable by algorithmic means, we may adopt other computational techniques including use of heuristics, numerical and/or statistical techniques. Even out of problems, which though theoretically have algorithmic solutions, yet require such large amount of resources, that this type of problems are designated as *infeasible* for the purpose of computational solution. Out of the problems, which are feasibly solvable, there are problems each of which may have more than one algorithms to solve the problem. For us, it is desirable to know which one is better among the available ones. For example, we can use the algorithms viz, Bubble sort, Insertion sort, Heapsort and Quicksort, for sorting a list of numbers. Their designs are different but the outcome is the same for all, for a given list of numbers. As, there are more than one algorithms available to us to sort a list of numbers, it is natural for us to think of using the algorithm which solves a particular sorting problem, in some way *better* than the others. In context of practical disciplines like computer applications, an *efficient* solution is generally taken as a *better* solution. Efficiency of an algorithm can be considered in terms of the efficient use of computer resources, such as processor time and memory space used. In addition to the efficiency of execution of algorithms, other factors like *time* (taken by a team of software engineers and/or programmers) *required for developing algorithms* and *reliability* may also be taken into consideration as factors towards overall efficiency of an algorithm.

Meanwhile, we have actually succeeded in making our discipline a science, and in a remarkably simple way: merely by deciding to call it "computer science"…

…We have seen computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and specially because it produces objects of beauty…

Donald E. Knuth
in
Turing Award
Lecture (1974)

However, most of the time, in respect of efficiency of algorithms, we are only concerned with the time and space requirements of execution of algorithms.

In this unit, we will discuss the issue of efficiency of computation of an algorithm in terms of the *amount of time* used in its execution. On the basis of analysis of an algorithm, the amount of time that is estimated to be required in executing an algorithm, will be referred to as the **time complexity** of the algorithm. The time complexity of an algorithm is measured in terms of some (basic) *time unit* (not second or nano-second). Generally, time taken in executing one move of a TM, is taken as (basic) time unit for the purpose. Or, alternatively, time taken in executing some elementary operation like addition, is taken as one unit. More complex operations like

multiplication etc, are assumed to require an *integral* number of basic units. As mentioned earlier, given many algorithms (solutions) for solving a problem, we would like to choose the most efficient algorithm from amongst the available ones. For *comparing efficiencies of algorithms*, that solve a particular problem, *time complexities of algorithms are* considered as *functions of the sizes of the problems* (to be discussed). The *time complexity functions of the algorithms are compared in terms of their* growth rates (*to be defined*) as growth rates are considered important measures of *comparative efficiencies*.

The concept of the **size of a problem,** though a fundamental one, yet is difficult to define precisely. Generally, the size of a *problem*, is measured in terms of the size of the *input*. The concept of the size of an input of a problem may be explained informally through examples. In the case of multiplication of two nxn (squares) matrices, the size of the problem may be taken as n^2, i.e, the number of elements in each matrix to be multiplied. For problems involving polynomials, the degrees of the polynomials may be taken as measure of the sizes of the problems.

Also, we may have an intuitive idea about the term **growth rate** *and its significance* in the comparative study of algorithms that can be designed to solve problems. For the time being, in stead of attempting a formal definition, we illustrate the concept of *growth rate of time complexity function* of an algorithm and its significance through the following example.

Let us consider two algorithms to solve a problem P, having time-complexities respectively as $f_1(n) = 1000n^2$ and $f_2(n) = 5n^4$, where size of the problem is assumed to be n. Then

f_1 (n) f_2 (n) for n 14 and

f_1 (n) f_2 (n) for n 15.

Also, the increase in the ratio $(f_2 (n)/f_1 (n))$ is *faster* than increase in n. Thus, informally, growth rate of f_2 (n) is more than the growth rate of f_1 (n). In one sense, the algorithm having time complexity f_2 (n) is *inferior* to the algorithm having time complexity $f_1(n)$ as growth rate of $f_2(n)$ is *faster* than that of f_1 (n).

A number of well-known notations for the formal treatment of the growth rate will be introduced later on within this section itself.

For a problem, a solution with time complexity which can be expressed as a polynomial of the size of the problem, is considered to have an **efficient solution**. Unfortunately, not many problems that arise in practice, admit any efficient algorithms, as these problems can be solved, if at all, by only non-polynomial time algorithms. A problem which does not have any (*known*) polynomial time algorithm is called an **intractable** problem.

At this stage, it is important to be aware of the following relevant facts

(i) **A non-polynomial function need not always be exponential:**. For example, the function $f(n) = n\log_2 n$ is neither polynomial function nor exponential function of n, but, somewhere between the two .

(ii) **The term *solution* in its general form: *need not* be an *algorithm***. If by tossing a coin, we get the correct answer to each instance of a problem, then the process of tossing the coin and getting answers constitutes a solution. But, the process is not an algorithm. Similarly, we solve problems based on **heuristics**, i.e, good

For details, refer Page 415, Introduction to Automata Theory, Languages, and Computation (Second Edition) by Hopcroft, Motwani &Ullman, *Pearson Eduction Inc (2001).*

guesses which, generally but not necessarily always, lead to solutions. All such cases of solutions are not algorithms, or algorithmic solutions. To be more explicit, by an algorithmic solution A of a problem L (*considered as a language*) from a problem domain *, we mean that among other conditions, the following are satisfied:

(a) A is a step-by-step method in which for each instance of the problem, there is a definite sequence of execution steps (*not involving any guess work*).

(b) A *terminates* for each x$\in$ *, irrespective of whether x $\in$ L or x L.

In this sense of algorithmic solution, only a ***solution by a Deterministic TM*** *is called an* ***algorithm****. A solution by a* ***Non-Deterministic TM may not be an algorithm.***

(iii) However, for every NTM solution, there is a Deterministic TM (DTM) solution of a problem. Therefore, if there is an NTM solution of a problem, then there is an algorithmic solution of the problem. *However, the symmetry may end here*.

The *computational equivalence* of Deterministic and Non-Deterministic TMs does not state or guarantee any *equivalence in respect of requirement of resources* like time and space by the Deterministic and Non-Deterministic models of TM, for solving a (solvable) problem. To be more precise, if a problem is solvable in polynomial-time by a Non-Deterministic Turing Machine, then it is, of course, *guaranteed* that there is a deterministic TM that solves the problem, but it is *not guaranteed* that there exists a Deterministic TM that solves the problem *in polynomial time.* Rather, **this fact forms the basis for one of the deepest open questions of Mathematics, which is stated as '*whether P = NP?*'**(*P and NP to be defined soon*).

The question put in simpler language means: Is it possible to design a Deterministic TM to solve a problem in polynomial time, for which, a Non-Deterministic TM that solves the problem in polynomial time, has already been designed?

We summarize the above discussion from the intractable problem's definition onward. Let us begin with definitions of the notions of P and NP.

P denotes the class of all problems, for each of which there is at least one *known* polynomial time Deterministic TM solving it.

NP denotes the class of all problems, for each of which, there is at least one known Non-Deterministic polynomial time solution. However, this solution may not be reducible to a polynomial time algorithm, i.e, to a polynomial time DTM.

Thus starting with two distinct classes of problems, viz**, tractable** problems and **intractable** problems, we introduced two classes of problems called **P and NP**. Some interesting relations known about these classes are:

(i) P = set of tractable problems
(ii) P NP.

(*The relation (ii) above* simply follows from the fact that every Deterministic TM is a special case of a Non-Deterministic TM).

However, it is not known whether P=NP or P NP. This forms the basis for the subject matter of the rest of the chapter. As a first step, we introduce some notations to facilitate the discussion of the concept of computational complexity.

2.1 OBJECTIVES

At the end of this unit, you should be able to:

explain the concepts of time complexity, size of a problem, growth rate of a function;

define and explain the well-known notations for growth rates of functions, viz O, , ,o, ;

explain criteria for classification of problems into undefincable defineable but not solvable, solvable but not feasible, P, NP, NP-hard and NP-Complete etc.;

define a number of problems which are known to be NP-complete problems;

explain polynomial-reduction as a technique of establishing problems as NP-hard;

establish NP-completeness of a number of problems.

2.2 NOTATIONS FOR GROWTH RATES OF FUNCTIONS

2.2.1 The Constant Factor in Complexity Measure

The time required by a solution or an algorithm for solving a (solvable) problem, depends *not only* on the size of the problem/input and the number of operations that the algorithm/solution uses, *but also* on the hardware and software used to execute the solution. However, the effect of change/improvement in hardware and software on the time required may be closely approximated by a *constant*.

Suppose, a supercomputer executes *instructions* one million times faster than another computer. Then irrespective of the size of a (solvable) problem and the solution used to solve it, the supercomputer solves the *problem* roughly million times faster than the computer, if the same solution is used on both the machines to solve the problem. Thus we conclude that the time requirement for execution of a solution, changes roughly by a *constant factor* on change in hardware, software and environmental factors.

An **important consequence of the above discussion** is that if the time taken by one machine in executing a solution of a problem is a polynomial (or exponential) function in the size of the problem, then time taken by every machine is a polynomial (or exponential) function respectively, in the size of the problem. *Thus, functions differing from each other by constant factors, when treated as time complexities should not be treated as different, i.e., should be treated as complexity-wise equivalent.*

2.2.2 Asymptotic Considerations

Computers are generally used to solve problems involving *complex* solutions. The complexity of solutions may be either because of the large number of involved computational steps and/or large size of input data. The plausibility of the claim apparently follows from the fact that, when required, computers are used *generally not to* find the product of two 2x2 matrices *but to find* the product of two nxn matrices for large n running into hundreds or even thousands.

Similarly, computers, when required, are generally used *not to find roots* of quadratic equations but for finding roots of complex equations including polynomial equations of *degrees more than hundreds or sometimes even thousands.*

The above discussion leads to the conclusion that when considering time complexities $f_1(n)$ and $f_2(n)$ of (computer) solutions of a problem of size n, we need to consider and compare the behaviors of the two functions only for large values of n. If the relative behaviors of two functions for smaller values conflict with the relative behaviours for larger values, then we may ignore the conflicting behaviour for smaller values. For example, if the earlier considered two functions

$f_1(n) = 1000\ n^2$ and
$f_2(n) = 5n^4$

represent time complexities of two solutions of a problem of size n, then despite the fact that

$f_1(n)$ $f_2(n)$ for n 14,

we would still prefer the solution having $f_1(n)$ as time complexity because

$f_1(n)$ $f_2(n)$ for all n 15.

This explains the reason for the presence of the phrase 'n k' in the definitions of the various measures of complexities discussed below:

2.2.3 Well Known Asymptotic Growth Rate Notations

In the following we discuss some well-known growth rate notations. These notations denote relations from functions to functions.

For example, if functions

f, g: $N \rightarrow N$ are given by

$f(n) = n^2 - 5n$ and

$g(n) = n^2$

then

$O(f(n)) = g(n)$ or $O(n^2 - 5n) = n^2$

(the notation O to be defined soon).

To be more precise, each of these notations is a mapping that associates a *set of* functions to each function. For example, if f (n) is a polynomial of degree k then the set O (f (n)) includes all polynomials of degree less than or equal to k.

The five well-known notations and how these are pronounced:

(i) O(O (n^2) is pronounced as 'big-oh of n^2' or sometimes just as oh of n^2)

(ii) ((n^2) is pronounced as 'big-omega of n^2 or sometimes just as omega of n^2')

(iii) ((n^2) is pronounced as 'theta of n^2')

(iv) o (o (n^2) is pronounced as 'little-oh of n^2')

(v) ((n^2) is pronounced as 'little- omega of n^2')

Remark 2.2.3.1

In the discussion of any one of the five notations, generally two functions say f and g are involved. The functions have their domains and Codomains as N, the set of natural numbers, i.e,

$f: N \rightarrow N$
$g: N \rightarrow N$

These functions may also be considered as having domain and codomain as R.

Remark 2.2.3.2

The purpose of these asymptotic growth rate notations and functions denoted by these notations, is to facilitate the recognition of *essential character of a complexity function* through some simpler functions delivered by these notations. For example, a complexity function $f(n) = 5004\ n^3 + 83\ n^2 + 19\ n + 408$, has essentially the same behaviour as that of $g(n) = n^3$ as the problem size n becomes larger and larger. But $g(n) = n^3$ is much more comprehensible than the function f(n). Let us discuss the notations, starting with the notation **O.**

2.2.4 The Notation O

Provides asymptotic *upper bound* for a given function. Let f(x) and g(x) be two functions each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be O (g(x)) (*pronounced as big-oh of g of x)* if there exist two positive integer/real number Constants C and k such that

f (x) C g(x) for all x k (A)

(*The restriction of being positive on integers/reals is justified as all complexities are positive numbers*)

Example 2.2.4.1: For the function defined by

$f(x) = 2x^3 + 3x^2 + 1$
show that

(i) $f(x) = O(x^3)$
(ii) $f(x) = O(x^4)$
(iii) $x^3 = O(f(x))$
(iv) x^4 $O(f(x))$
(v) $f(x)$ $O(x^2)$

Solutions

Part (i)

Consider

$f(x) = 2x^3 + 3x^2 + 1$
$2x^3 + 3x^3 + 1\ x^3 = 6x^3$ for all x 1

(by replacing each term x^i by the highest degree term x^3)

there exist C = 6 and k = 1 such that

f(x) C. x^3 for all x k

Thus we have found the required constants C and k. Hence f(x) is $O(x^3)$.

Part (ii)

As above, we can show that

f(x) 6 x^4 for all x 1.

However, we may also, by computing some values of f(x) and x^4, find C and k as follows:

$f(1) = 2+3+1 = 6$; $(1)^4 = 1$
$f(2) = 2.2^3 + 3.2^2 + 1 = 29$; $(2)^4 = 16$
$f(3) = 2.3^3 + 3.3^2 + 1 = 82$; $(3)^4 = 81$

for C = 2 and k = 3 we have
f(x) 2. x^4 for all x k

Hence f(x) is $O(x^4)$.

Part (iii)

for C = 1 and k = 1 we get
x^3 C $(2x^3 + 3x^2 +1)$ for all x k

Part (iv)

We prove the result by contradiction. Let there exist positive constants C and k such that

x^4 C $(2x^3 + 3x^2 +1)$ for all x k
x^4 C $(2x^3 +3x^3+x^3) = 6Cx^3$ for x k
x^4 6 C x^3 for all x k.

implying x 6C for all x k
But for x = max of { 6 C + 1, k}, the previous statement is not true.
Hence the proof.

Part (v)

Again we establish the result by contradiction.
Let O $(2 x^3+3x^2+1) = x^2$
Then for some positive numbers C and k
$2x^3 + 3x^2+1$ C x^2 for all x k,
implying
x^3 C x^2 for all x k *(∵ x^3 $2x^3+3x^2+1$ for all x 1)*
implying
x C for x k
Again for x = max C $1, k$
The last imaquality does not hold. Hence the result.

Example: The big-oh notation can be used to estimate S_n, the sum of first n positive integers

Hint: S_n=1+2+3+…….+n n+n +……+ n = n^2

Therefore, $S_n = O(n^2)$.

Remark 2.2.4.2

It can be easily seen that for given functions f(x) and g(x), if there exists one pair of C and k with f(x) C.g (x) for all x k, then there exist infinitely many pairs (C_i, k_i) which satisfy

f(x) C_i g(x) for all x k_i.

Because for **any C_i** C and **any k_i** k, the above inequality is true., if f(x) c.g(x) for all x k.

2.2.5. The Notation

Provides an asymptolic *lower bound* for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be (g(x)) (*pronounced as big-omega of g of x)* if there exist two positive integer/real number Constants C and k such that

f(x) C (g(x)) whenever x k

Example 2.2.5.1: For the functions

$f(x) = 2x^3 + 3x^2 + 1$ and $h(x) = 2x^3-3x^2+2$

show that

(i) f(x) = (x^3)

(ii) h(x)= (x^3)

(iii) h(x)= (x^2)

(iv) x^3 = (h(x))

(v) x^2 (h(x))

Solutions:

Part (i)

For C =1, we have

f(x) C x^3 for all x 1

Part (ii)

$h(x) = 2x^3-3x^2+2$

Let C and k > 0 be such that

$2x^3-3x^2+2$ C x^3 for all x k

i.e (2-C) x^3-3x^2+2 0 for all x k

Then C = 1 and k 3 satisfy the last inequality.

Part (iii)

$2x^3-3x^2+2$ = (x^2)

Let the above equation be true.
Then there exists positive numbers C and k
s.t.

$2x^3-3x^2+2 \quad C\,x^2 \quad$ for all $x \quad k$
$2\,x^3-(3 + C)\,x^2 + 2 \quad 0$

It can be easily seen that lesser the value of C, better the chances of the above inequality being true. So, to begin with, let us take C = 1 and try to find a value of k s.t

$2x^3-4x^2+2 \quad 0.$

For $x \quad 2$, the above inequality holds
k=2 is such that

$2x^3-4x^2+2 \quad 0$ for all $x \quad k$

Part (iv)

Let the equality

$x^3 = \quad (2x^3-3x^2+2)$

be true. Therefore, let C>0 and k > 0 be such that

$x^3 \quad C(2(x^3-3/2\,x^2+1))$

For C = ½ and k = 1, the above inequality is true.

Part (v)

We prove the result by contradiction.

Let $x^2 = \quad (3x^3-2x^2+2)$

Then, there exist positive constants C and k such that

$x^2 \quad C\,(3x^3 - 2x^2 + 2) \quad$ for all $x \quad k$

i.e $(2C +1)\,x^2 \quad 3C\,x^3 + 2 \quad C\,x^3$ for all $x \quad k$

$\frac{2C \quad 1}{C} \quad x \quad$ for all $x \quad k$

But for any $x \quad 2 \quad \frac{(2C \quad 1)}{C}$,

The above inequality can not hold. Hence contradiction.

2.2.6 The Notation

Provides simultaneously *both* asymptotic *lower* bound and asymptotic *upper* bound for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers. Then f(x) said to be (g(x)) (*pronounced as big-theta of g of x*) if, there exist positive constants C_1, C_2 and k such that $C_2\ g(x)$ $f(x) \quad C_1\ g(x)$ for all $x \quad k$.

(*Note the last inequalities represent two conditions to be satisfied simultaneously viz* $C_2\ g(x) \quad f(x)$ *and* $f(x) \quad C_1\ g(x)$)

We state the following theorem without proof, which relates the three functions O, ,

Theorem: For any two functions f(x) and g(x), f(x) = (g(x)) if and only if f(x) = O (g(x)) and f(x) = (g(x)).

Examples 2.2.6.1: For the function
$f(x) = 2x^3 + 3x^2 + 1$, show that

(i) f(x) = (x^3)

(ii) f(x) (x^2)

(iii) f(x) (x^4)

Solutions

Part (i)

for $C_1 = 3$, $C_2 = 1$ and k = 4

1. $C_2 x^3$ f(x) $C_1 x^3$ for all x k

Part (ii)

We can show by contradiction that no C_1 exists.

Let, if possible for some positive integers k and C_1, we have $2x^3+3x^2+1$ $C_1 . x^2$ for all x k

Then
x^3 $C_1 x^2$ for all x k
i.e,
x C_1 for all x k
But for
x= max C_1 $1, k$
The last inequality is not true

Part (iii)

f(x) (x^4)

We can show by contradiction that there does not exist C_2 s.t

$C_2 x^4$ $(2x^3 + 3x^2 + 1)$

If such a C_2 exists for some k then $C_2 x^4$ $2x^3 + 3x^2 + 1$ $6x^3$ for all x k 1,

implying

$C_2 x$ 6 for all x k

But for x = $\frac{6}{C_2}$ 1

the above inequality is false. Hence, proof of the claim by contradiction.

2.2.7 The Notation o

The asymptotic upper bound provided by big-oh notation may or may not be tight in the sense that if $f(x) = 2x^3 + 3x^2 + 1$

Then for $f(x) = O(x^3)$, though there exist C and k such that
$f(x)$ $C(x^3)$ for all x k
yet there may also be some values for which the following equality also holds
$f(x) = C(x^3)$ for x k

However, if we consider

$f(x) = O(x^4)$

then there can not exist positive integer C s.t

$f(x) = C x^4$ for all x k
The case of $f(x) = O(x^4)$, provides an example for the next notation of small-oh.

The Notation o

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers

Further, let $C > 0$ **be any number**, then $f(x) = o(g(x))$ (pronounced as little oh of g of x) if there exists natural number k satisfying
$f(x) < C g(x)$ for all x k 1 (B)

Here we may note the following points

(i) In the case of little-oh the constant C does not depend on the two functions f (x) and g (x). Rather, we can *arbitrarily* choose C >0

(ii) The inequality (B) is strict whereas the inequality (A) of big-oh is not necessarily strict.

Example 2.2.7.1: For $f(x) = 2x^3 + 3x^2 + 1$, we have

(i) $f(x) = o(x^n)$ for any n 4.
(ii) $f(x)$ $o(x^n)$ for n 3

Solution

Let $C > 0$ **be given and** to find out k satisfying the requirement of little-oh.
Consider

$2x^3 + 3x^2 + 1 < C x^n$
$= 2 + \frac{3}{x} \frac{1}{x^3} < C x^{n-3}$

Case when n = 4
Then above inequality becomes

$2 + \frac{3}{x} \frac{1}{x^3} < C x$

if we take k = max $\frac{7}{C}$,1

then
$2x^3 + 3x^2 + 1 < C\,x^4$ for x k.

In general, as $x^n > x^4$ for n 4,

therefore
$2x^3 + 3x^2 + 1 < C\,x^n$ for n 4
for all x k
with k = max $\frac{7}{c}$,1

Part (ii)

We prove the result by contradiction. Let, if possible, $f(x) = 0(x^n)$ for n 3.
Then there exist positive constants C and k such that $2x^3+3x^2+1< C\,x^n$
for all x k.

Dividing by x^3 throughout, we get

$$2+ \frac{3}{x} \quad \frac{1}{x^2} < C\,x^{n-3}$$

n 3 and x k
As C is arbitrary, we take
C = 1, then the above inequality reduces to

$$2+ \frac{3}{x} \quad \frac{1}{x^2} < C.\,x^{n-3} \quad \text{for n 3 and x k 1.}$$

Also, it can be easily seen that

x^{n-3} 1 for n 3 and x k 1.

$$2+ \frac{3}{x} \quad \frac{1}{x^2} \quad 1 \qquad \text{for n 3}$$

However, the last inequality is not true. Therefore, the proof by contradiction.

Generalizing the above example, we get the

Example 2.2.7.2: If f(x) is a polynomial of degree m and g(x) is a polynomial of degree n. Then
f(x) = o(g(x)) if and only if n>m.

we state (without proof) below two results which can be useful in finding small-oh upper bound for a given function

More generally, we have

Theorem 2.2.7.3: Let f(x) and g(x) be functions in definition of small-oh notation.

Then f(x) = o(g(x) if and only if

$$\text{Lim} \quad \frac{f(x)}{g(x)} = 0$$

Lim x $\longrightarrow$

Next, we introduce the last asymptotic notation, namely, small-omega. The relation of small-omega to big-omega is similar to what is the relation of small-oh to big-oh.

2.2.8 The Notation

Again the asymptotic lower bound may or may not be tight. However, the asymptotic bound *cannot* be tight. *The formal definition of is follows:*

Let f(x) and g(x) be two functions each from the set of natural numbers or the set of positive real numbers to set of positive real numbers.

Further

Let C > 0 **be any number**, then
f(x) = (g(x))
if there exist a positive integer k s.t

f(x) > C g(x) for all x k

Example 2.2.8.1: If $f(x) = 2x^3 + 3x^2 + 1$
then
f(x) = (x)
and also
f(x) = (x^2)

Solution:

Let C be any positive constant.

Consider

$2x^3 + 3x^2 + 1 > C\,x$

To find out k 1 satisfying the conditions of the bound .

$2x^2 + 3x + \frac{1}{x} > C$ *(dividing throughout by x)*
Let k be integer with k C+1

Then for all x k

$2x^2 + 3x + \frac{1}{x}$ $2x^2 + 3x > 2k^2 + 3k > 2C^2 + 3C > C$. (∵ k C+1)
f(x) = (x)

Again, consider, for any C > 0,

$2x^3 + 3x^2 + 1 > C\,x^2$
then
$2x + 3 + \frac{1}{x^2} > C$ Let k be integer with k C+1
Then for x k we have
$2x + 3 + \frac{1}{x^2}$ $2x + 3 > 2k + 3 > 2C + 3 > C$
Hence

f(x) = (x^2)

In general, we have the following two theorems (stated without proof).

Theorem 2.2.8.2: If f(x) is a polynomial of degree n, and g(x) is a polynomial of degree n, then

f(x) = (g(x)) if and only if m > n.

More generally

Theorem 2.2.8.3: Let f(x) and g(x) be functions in the definitions of little-omega Then f(x) = g(x) if and only if

$$\lim_{x} \frac{f(x)}{g(x)}$$

or

$$\lim_{x} \frac{g(x)}{f(x)} \quad 0$$

Ex.1) Show that n! = $O(n^n)$.

Ex.2) Show that $n^2 + 3\log n = O(n^2)$.

Ex.3) Show that $2^n = O(5^n)$.

2.3 CLASSIFICATION OF PROBLEMS

Where there are problems, there is life

Zinaviev
The Radiant Future.

The fact of being engaged in solving problems may be the only sure indication of a living entity being alive *(though, the distinction between entities being alive and not being alive is getting fuzzier day by day*). The problems, attempted to be solved, may be due to the need for survival in a hostile and competitive environment or may be because of intellectual curiosity of knowing more and more of the nature. In the previous unit, we studied a number of problems which are not solvable by computational means. **We can go still further and categorize the problems, which we encounter or may encounter, into the following broad classes:**

(I) **Problems which can not even be defined formally.**

By a formal definition of a problem, we mean expressing in terms of mathematical entities like sets, relations and functions etc , the information concerning the problem, in respect of at least

a) Possible inputs
b) Possible outcomes
c) Entitles occurring and operations on these entities in the (dynamic) problem domains.

In this sense of *definition of a problem*, what to talk of solving, most of the problems can not even be defined. Think of the following problems

a) Why the economy is not doing well?
b) Why there is hunger, illiteracy and suffering despite international efforts to eradicate these?
c) Why some people indulge in corrupt practices despite being economically well?

These are some of problems, the definition of each of which require enumeration of potentially infinite parameters, and hence are almost impossible to define.

II **Problems which can be formally defined but can not be solved by computational means.** We discussed some of these problems in the previous unit.

III **Problems which, though theoretically can be solved by computational means, yet are infeasible, i.e**, these problems require so large amount of computational resources that practically it is not feasible to solve these problems by computational means. These problems are called **intractable or infeasible** problems. The distinguishing feature of the problems is that for each of these problems any solution has time complexity which is exponential, Please or at least non-polynomial, function of the problem size.

IV **Problems that are called feasible or theoretically not difficult to solve by computational means**. The distinguishing feature of the problems is that for each instance of any of these problems, there exists a **Deterministic Turing Machine** that solves the problem having time-complexity as a polynomial function of the size of the problem. **The class of problem is denoted by P.**

V **Last, but probably most interesting class include large number of problems, for each of which, it is not known whether it is in P or not in P.** These problems fall somewhere between class III and class IV given above However, for each of the problems in the class, it is known that **it is in NP, i.e. each can** be solved by at least one **Non-Deterministic** Turing Machine, the time complexity of which is a polynomial function of the size of the problem.

A problem from the class NP can equivalently but in more intuitive way, be defined as one for which a potential solution, if given, can be **verified** in polynomial time whether the potential solution is actually a solution or not.

The problems in this class, are called **NP-Complete problems** (to be formally defined later). *More explicitly, a problem is NP-complete if it is in NP and for which no polynomial-time Deterministic TM solution is known so far.*

Most interesting aspect of NP-complete problems, is that for each of these problems *neither, so far*, it has been possible to design a Deterministic polynomial-time TM solving the problem *nor* it has been possible to show that Deterministic polynomial - time TM solution *can not* exist.

The idea of NP-completeness was introduced by Stephen Cook in 1971 and the **satisfiablity problem** defined below is the first problem that was proved to be NP-complete, of course, by S. Cook.

Next, we enumerate some of the NP-complete problems without justifying why these problems have been placed in the class. Justification for some of these problems will be provided in later sections.

A good source for the study of NP-complete problems and of related topics is Garey & Johnson[+]

[*] *Cook S.A: The complexity of Theorem providing procedures, proceedings of the third annual ACM symposium on the Theory of computing*, New York: Association of Computing Machinery, 1971, pp. 151-158.

+ Garey M.R. and Johnson D.S. : Computers and Intractability: A guide to the Theory of NP-Completeness, *H.Freeman, New York, 1979.*

Problem1: Satisfiability problem (or, for short, SAT) states: Given a Boolean expression, is it satisfiable?

Explanation: A Boolean expression involves

(i) Boolean variables $x_1, x_2,..., x_i, \ldots$,each of which can assume a value either TRUE (generally denoted by 1) or FALSE (generally denoted by 0) and

(ii) Boolean/logical operations: NOT(x_1) (generally denoted by x_i or $\overline{x_i}$), AND (denoted generally by), and OR (denoted by). Other logical operators like and can be equivalently replaced by some combinations of , and .

(iii) Pair of parentheses

(iv) A set of syntax rules, which are otherwise well known.

For example

($(x_1$ $x_2)$ x_3) is (legal) Boolean expression.

Next, we explain other concepts involved in SAT.

Truth Assignment: For each of the variables involved in a given Boolean expression, associating a value of either 0 or 1, gives a truth assignment, which in turn gives a truth-value to the Boolean expression.

For example: Let $x_1= 0$, $x_2=1$, and $x_3=1$ be one of the eight possible assignments to a Boolean expression involving x_1, x_2 and x_3
Truth-value of a Boolean expression.

Truth value of ($(x_1$ $x_2)$ x_3) for the truth–assignment $x_1=0$, $x_2=1$ and $x_3=1$ is
((0 1) 1) = (0 0) =0

Satisfiable Boolean expression: A Boolean expression is said to be satisfiable if at least one truth assignment makes the Boolean expression True

For example: $x_1=1$, $x_2=0$ and $x_3= 0$ is one assignment that makes the Boolean expression ($(x_1$ $x_2)$ x_3) True. Therefore, ($(x_1$ $x_2)$ x_3) is satisfiable.

Problem 2: CSAT or CNFSAT Problem: given a Boolean expression in CNF, is it satisfiable?

Explanation: A Boolean formula FR is said to be in Conjunctive Normal Form (i.e., CNF) if it is expressed as C_1 C_2 …. C_k where each C_i is a disjunction of the form

$$x_{i1} \quad x_{i2} \quad \ldots \quad x_{im}$$

where each x_{ij} is a literal. A **literal** is either a variable x_i or negation $\overline{x_i}$ of variable x_i.

Each C_i is called a **conjunct.** It can be easily shown that every logical expression can equivalently be expressed in CNF

Problem 3: Satisfiability (or for short, 3SAT) Problem: given a Boolean expression in 3-CNF, is it satisfiable?

Further Explanation: If each conjunct in the CNF of a given Boolean expression contains exactly three distinct literals, then the CNF is called 3-CNF

Problem 4: Primality problem: given a positive integer n, is n prime?

Problem 5: Traveling salesman Problem (TSP)

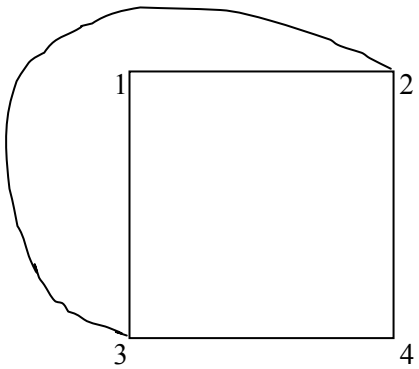
Given a set of cities C= {C_1, C_2, …. C_n} with n >1, and a function d which assigns to each pair of cities (C_i, C_j) some cost of traveling from C_i to C_j. Further, a positive integer/real number B is given. The problem is to find a route (covering each city exactly once) with cost at most B.

Problem 6: Hamiltonian circuit problem (H C P): given an undirected graph G = (V, E), does G contain a Hamiltonian circuit?

Further Explanation: A Hamiltonian circuit of a graph G = (V, E) is a set of edges that connects the nodes into a single cycle, with each node appearing exactly once. *We may note that the number of edges on a Hamiltonian circuit must equal the number of nodes in the graph.*

Further, it may be noted that HCP is a special case of TSP in which the cost between pairs of nodes is the same, say 1.

Example: Consider the graph



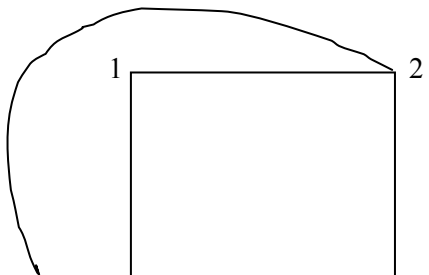
Then the above graph has one Hamiltonian circuit viz (1, 2, 4, 3, 1)

Problem 7: The vertex cover problem (V C P) (also known as Node cover problem): Given a graph G = (V,E) and an integer k, is there a vertex cover for G with k vertices?

Explanation: A vertex cover for a graph G is a set C of vertices so that each edge of G has an endpoint in G. For example, for the graph shown above, {1, 2, 3} is a vertex cover. It can be easily seen that every superset of a vertex cover of a graph is also a vertex cover of the graph.

Problem 8: K-Colourability Problem: Given a graph G and a positive integer k, is there a k-colouring of G?

Explanation: A k-colouring of G is an assignment to each vertex of one of the k colours so that no two adjacent vertices have the same color. It may be recalled that two vertices in a graph are adjacent if there is an edge between the two vertices



3 4

As the vertices 1, 2, 3 are mutually adjacent therefore, we require at least three colours for k-colouring problem.

Problem 9: **The complete subgraph problem (CSP Complete Sub) or clique problem: Given a graph G and positive integer k, does G have a complete subgraph with k vertices?**

Explanation: For a given graph G = (V, E), two vertices v_1 and v_2 are said to be **adjacent** if there is an edge connecting the two vertices in the graph.

A **subgraph** H= (V_1, E_1) of a graph G = (V, E) is a graph such that V_1 V and E_1 E. In other words, each vertex of the subgraph is a vertex of the graph and each edge of the subgraph is an edge of the graph**.**

Complete Subgraph of a given graph G is a subgraph in which every pair of vertices is adjacent in the graph

For example in the above graph, the subgraph containing the vertices {1, 2, 3} and the edges (1, 2), (1, 3), (2, 3) is a complete subgraph or a clique of the graph. However, the whole graph is not a clique as there is no edge between vertices 1 and 4.

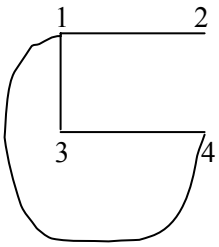
Problem 10: Independent set problem: Given a graph G = (V, E) and a positive integer k, is there an independent set of vertices with at least k elements?

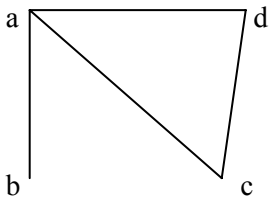
Explanation: A subset V_1 of the set of vertices V of graph G is said to be independent, if *no two* distinct vertices in V_1 are adjacent. For example, in the above graph V_1 = {1, 4} is an independent set.

Problem 11: **The subgraph isomorphism problem: Given graph G_1 and G_2, does G_1 contain a copy of G_2 as a subgraph?**

Explanation: Two graphs H_1 = (V_1, E_1) and H_2 = (V_2, E_2) are said to be **isomorphic** if we can rename the vertices in V_2 in such a manner that after renaming, the graph H1 and H2 look identical *(not necessarily pictorially, but as ordered pairs of sets)*

For Example



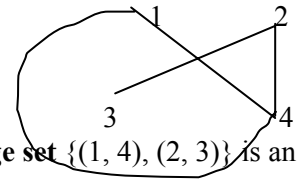


are isomorphic graph because after mapping 1 a, 2 b, 3 c and 4 d, the two graphs become identical.

Problem 12: **Given a graph g and a positive integer k, does G have an "edge cover" of k edges?**

Explanation: For a given graph G = (V,E), a subset E_1 of the set of edges E of the graph, is said to be an edge cover of G, if every vertex is an end of at least one of the edges in E_1.

For Example, for the graph



The two-edge set {(1, 4), (2, 3)} is an edge cover for the graph.

Problem 13: Exact cover problem: For a given set $\mathcal{P} = \{S_1, S_2, \ldots, S_m\}$, where each S_i is a subset of a given set S, is there a subset Q of $\mathcal{P}$ *such that for each x* in S, there is exactly one S_i in Q for which x is in S_i ?

Example: Let S = {1, 2, …,10}
and $\mathcal{P}$ = { S_1, S_2, S_3, S_4, S_5} s.t
S_1 = {1, 3, 5}
S_2 = {2, 4, 6}
S_3 = {1, 2, 3, 4}
S_4 = {5, 6, 7, 9, 10}
S_5 = {7, 8, 9, 10 }
Then Q = { S_1, S_2, S_5} is a set cover for S.

Problem 14: The knapsack problem: Given a list of k integers n_1, n_2… n_k, can we partition these integers into two sets, such that sum of integers in each of the two sets is equal to the same integer?

2.4 REDUCTION, NP-COMPLETE AND NP-HARD PROBLEMS

Earlier we (informally) explained that a problem is called NP-Complete if P has at least one Non-Deterministic polynomial-time solution and further, so far, no polynomial-time Deterministic TM is known that solves the problem.

In this section, we formally define the concept and then describe a general technique of establishing the NP-Completeness of problems and finally apply the technique to show some of the problems as NP-complete. We have already explained how a problem can be thought of as a language L over some alphabet . Thus the terms *problem and language* may be interchangeably used.

For the formal definition of NP-completeness, *polynomial-time reduction*, as defined below, plays a very important role.

In the previous unit, we discussed *reduction technique* to establish some of the problems as undecidable. The method *that was used for establishing undecidability* of a language using the technique of reduction, may be briefly described as follows:

Let P_1 be a problem which is *already known* to be undecidable. We want to check whether a problem P_2 is undecidable or not. If we are able to design an algorithm which transforms or constructs an instance of P_2 for each instance of P_1, then P_2 is also undecidable.

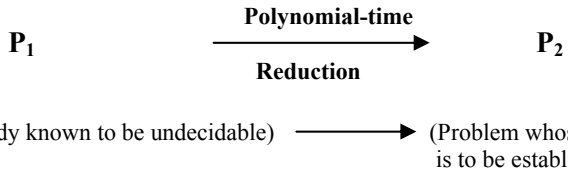
The process of transformation of the instances of the problem **already known to the undecidable to** instances of the problem, the undecidability is to checked, is called **reduction.**

Some-what similar, but, slightly different, rather special, reduction called **polynomial-time reduction** is used to establish NP-Completeness of problems.

A Polynomial-time reduction is a polynomial-time algorithm which constructs the instances of a problem P_2 from the instances of some other problems P_1.

A method of establishing the NP-Completeness (*to be formally defined later*) of a problem P_2 constitutes of designing a *polynomial time reduction* that constructs an instance of P_2 for each instance of P_1, where P_1 is already known to be NP-Complete.

The direction of the mapping must be clearly understood as shown below.



Though we have already explained the concept of NP-Completeness, yet for the sake of completeness, we give below the formal definition of NP-Compleness

Definition: NP-Complete Problem: A Problem P or equivalently its language L_1 is said to be NP-complete if the following two conditions are satisfied:

(i) The problem L_2 is in the class NP
(ii) For any problem L_2 in NP, there is a polynomial-time reduction of L_1 to L_2

In this context, we introduce below another closely related and useful concept**.**

Definition: NP-Hard Problem A problem L is said to be NP-hard if for any problem L_1 in NP, there is a polynomial-time reduction of L_1 to L

In other words, a *problem L is hard if* only condition (ii) of NP-Completeness is satisfied. But the problem has may be so hard that establishing L as an NP-class problem is so far not possible.

However, from the above definitions, it is clear that every NP-complete problem L must be NP-Hard and additionally should satisfy the condition that L is an NP-class problem.

In the next section, we discuss NP-completeness of some of problems discussed in the previous section.

2.5 ESTABLISHING NP-COMPLETENESS OF PROBLEMS

In general, the process of establishing a problem as NP-Complete is a two-step process. **The first step**, which in most of the cases is quite simple, constitutes of **guessing** possible solutions of the instances, one instance at a time, of the problem and then **verifying** whether the guess actually is a solution or not.

The second step involves designing a polynomial-time algorithem which reduces instances of an already known NP-Complete problem to instances of the problem, which is intended to be shown as NP-Complete.

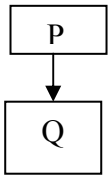
However, to begin with, there is a major hurdle in execution of the second step. The above technique of reduction can not be applied unless we already have established at least one problem as NP-Complete. Therefore, for the first NP-Complete problem, the NP-Completeness has to be established in a different manner.

As mentioned earlier, Stephen Cook (1971) established Satisfiablity as the first NP-Complete problem. The proof was based on explicit reduction of the language of any non-deterministic, polynomial-time TM to the satisfiablity problem.

The proof of Satisfiablity problem as the first NP-Complete problem, is quite lengthy and we skip the proof. Interested readers may consult any of the text given in the reference.

Assuming the satisfiablity problem as NP-complete, the rest of the problems that we establish as NP-complete, are established by reduction method as explained above.

A diagrammatic notation of the form



indicates: *Assuming P is already established as NP-Complete, the NP-Completeness of Q is established by through a polynomial-time reduction from P to Q*

A scheme for establishing NP-Completeness of some the problems mentioned in Section 2.2, is suggested by Fig. 2.1 given below

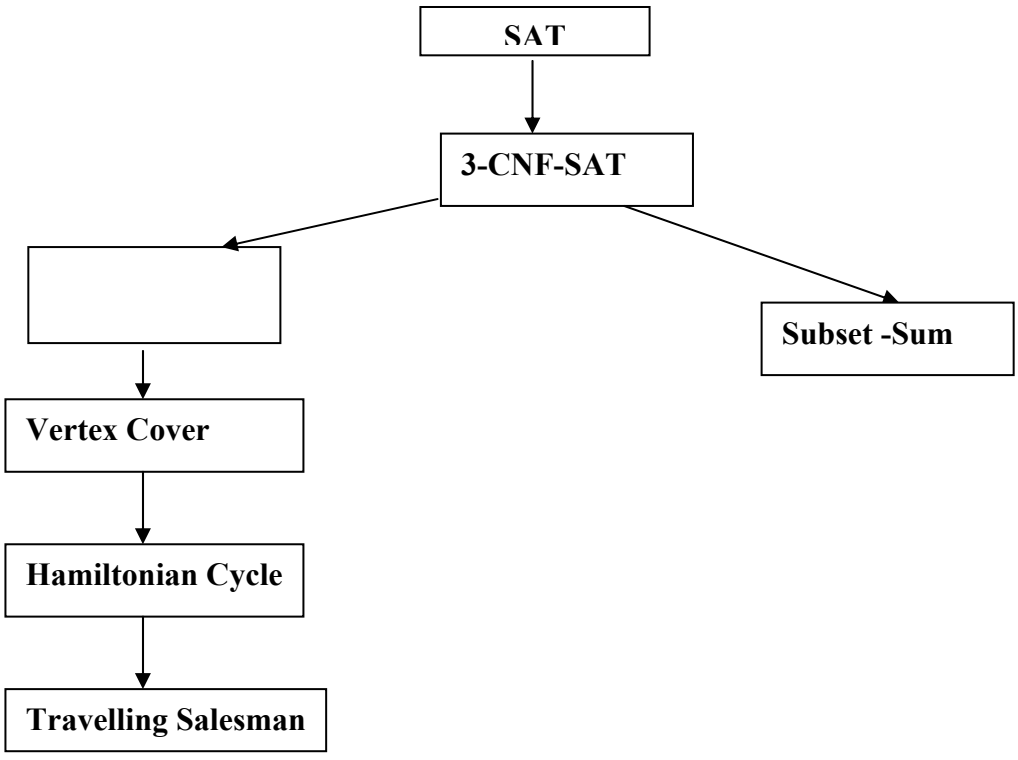


Fig. 2.1

Example 2.4.1: Show that the Clique problem is an **NP-complete** problem.

Proof : The verification of whether every pairs of vertices is connected by an edge in E, is done for different paris of vertices by a Non-deterministic TM, i.e, in parallel. Hence, it takes only polynomial time because for each of n vertices we need to verify at most n (n+1) /2 edges, the maximum number of edges in a graph with n vertices.

We next show that 3- CNF-SAT problem can be transformed to clique problem in polynomial time.
Take an instance of 3-CNF-SAT. An instance of 3CNF-SAT consists of a set of *n* clauses, each consisting of exactly 3 literals, each being either a variable or negated variable. It is satisfiable if we can choose literals in such a way that:

at least one literal from each clause is chosen

if literal of form *x* is chosen, no literal of form x is considered.

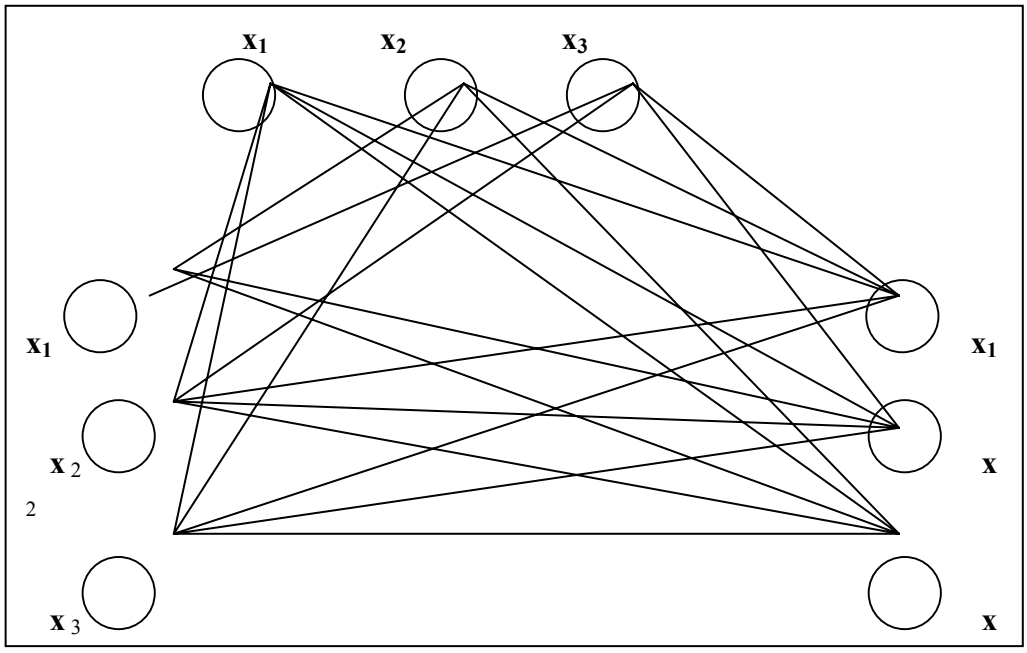


Fig. 2.2

For each of the literals, create a graph node, and connect each node to every node in other clauses, except those with the same variable but different sign. This graph can be easily computed from a boolean formula in 3-CNF-SAT in polynomial time. Consider an example, if we have

$$= (\ x_1 \vee x_2 \vee x_3\ )\ \ (\ x_1 \vee x_2 \vee x_3\ )\ \ (\ x_1 \vee x_2 \vee x_3\ )$$

then G is the graph shown in **Figure 2.2 above.**

In the given example, a satisfying assignment of is ($x_1 = 0, x_2 = 0, x_3 = 1$). A corresponding clique of size k = 3 consists of the vertices corresponding to x_2 from the first clause, x_3 from the second clause, and x_3 from the third clause.

The problem of finding *n*-element clique is equivalent to finding a set of literals satisfying SAT. Because there are no edges between literals of the same clause, such a clique must contain exactly one literal from each clause. And because there are no edges between literals of the same variable but different sign, if node of literal *x* is in the clique, no node of literal of form x is.

This proves that finding *n*-element clique in 3*n*-element graph is **NP-Complete**.

Example 5: Show that the Vertex cover problem is an **NP- complete**.

A *vertex cover* of an undirected graph$G = (V,E)$ is a subset V' of the vertices of the graph which contains at least one of the two endpoints of each edge.

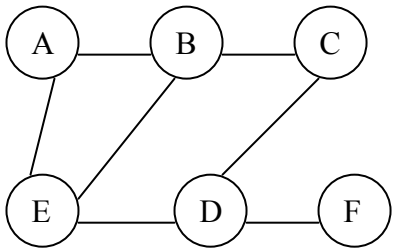


Fig. 2.3

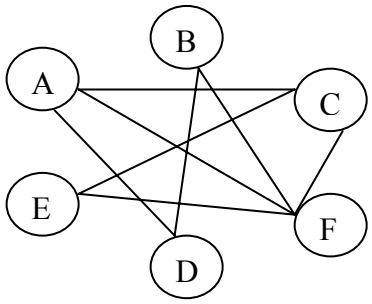


Fig. 2.4

The vertex cover problem is the optimization problem of finding a vertex cover of minimum size in a graph. The problem can also be stated as a decision problem :

VERTEX-COVER = {<*G*, *k*>| graph *G* has a vertex cover of size k }.

A deterministic algorithm to find a vertex cover in a graph is to list all subsets of vertices of size *k* and check each one to see whether it forms a vertex cover. This algorithm is exponential in *k.*

Proof : To show that Vertex cover problem **NP**, for a given graph G = (V, E), we take V' V and verifies to see if it forms a vertex cover. Verification can be done by checking for each edge (u, v) E whether u V' or v V'. This verification can be done in polynomial time.

Now, We show that clique problem can be transformed to vertex cover problem in polynomial time. This transformation is based on the notion of the complement of a graph G. Given an undirected graph G = (V, E), we define the complement of G as G' = (V, E'), where E' = { (u, v) | (u, v) E}. *i.e* G' is the graph containing exactly those edges that are not in G. The transformation takes a graph G and *k* of the clique problem. It computes the complement G' which can be done in polynomial time.

To complete the proof, we can show that this transformation is indeed reduction : the graph has a clique of size *k* if and only if the graph G' has a vertex cover of size |V| - *k.*

Suppose that G has a clique V' V with |V'| = *k*. We claim that V – V' is a vertex cover in G'. Let (u, v) be any edge in E'. Then, (u, v) E, which implies that atleast one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E. Equivalently, atleast one of u or v is in V – V', which means that edge (u, v) is covered by V – V'. Since (u, v) was chosen arbitrarily from E', every edge of E' is covered by a vertex in V – V'. Hence, the set V – V', which has size |V| - *k*, forms a vertex cover for G'.

Conversely, suppose that G' has a vertex cover V' V , where |V'| = |V| - *k*. Then, for all u, v V, if (u, v) E', then u V' or v V' or both. The contrapositive of

this implication is that for all u, v V, if u V' and v V', then (u, v) E. In other words, V – V' is a clique, and it has size |V| - |V'| = *k*.

For example, The graph G(V,E) has a clique {A, B, E} given by Figure 7.4. The complement of graph G is given by G' shown as Figure 7.5 and have independent set given by {C, D, F}.

This proves that finding the vertex cover is **NP-Complete**.

Ex.4) Show that the Partition problem is **NP**.

Ex.5) Show that the k-colorability problem is **NP**.

Ex.6) Show that the Independent Set problem is **NP- complete**.

Ex.7) Show that the Travelling salesman problem is **NP- complete**.

2.6 SUMMARY

In this unit in number of concepts are defined.

P denotes the class of all problems, for each of which there is at least one *known* polynomial time Deterministic TM solving it.

NP denotes the class of all problems, for each of which, there is at least one known Non-Deterministic polynomial time solution. However, this solution may not be reducible to a polynomial time algorithm, i.e, to a polynomial time DTM.

Next, five **Well Known Asymptotic Growth Rate Notations are defined.**

The notation O provides asymptotic *upper bound* for a given function.
Let f(x) and g(x) be two functions each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be O (g(x)) (*pronounced as big-oh of g of x)* if there exist two positive integer/real number Constants C and k such that

f (x) C g(x) for all x k

The notation provides an asymptolic *lower bound* for a given function

Let f(x) and g(x) be two functions, each from the set of natural numbers or set of positive real numbers to positive real numbers.

Then f (x) is said to be (g(x)) (*pronounced as big-omega of g of x)* if there exist two positive integer/real number Constants C and k such that

f(x) C (g(x)) whenever x k

The Notation

Provides simultaneously *both* asymptotic *lower* bound and asymptotic *upper* bound for a given function.

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers. Then f(x) said to be (g(x)) (*pronounced as big-theta of g of x*) if, there exist positive constants C_1, C_2 and k such that C_2 g(x) f(x) C_1 g(x) for all x k.

The Notation o

Let f(x) and g(x) be two functions, each from the set of natural numbers or positive real numbers to positive real numbers

Further, let C > 0 **be any number**, then f(x) = o(g(x)) (pronounced as little oh of g of x) if there exists natural number k satisfying

f(x) < C g(x) for all x k 1

The Notation

Again the asymptotic lower bound may or may not be tight. However, the asymptotic bound *cannot* be tight. *The formal definition of is follows:*

Let f(x) and g(x) be two functions each from the set of natural numbers or the set of positive real numbers to set of positive real numbers.
Further

Let C > 0 **be any number**, then

f(x) = (g(x))

if there exist a positive integer k s.t

f(x) > C g(x) for all x k

In Section 2.2 in defined, 14 well known problems, which are known to be NP-Complete.

In Section 2.3 we defined the following concepts.

A Polynomial-time reduction is a polynomial-time algorithm which constructs the instances of a problem P_2 from the instances of some other problems P_1

Definition: NP-Complete Problem: A Problem P or equivalently its language L_1 is said to be NP-complete if the following two conditions are satisfied:

(i) The problem L_2 is in the class NP
(ii) For any problem L_2 in NP, there is a polynomial-time reduction of L_1 to L_2

Definition: NP-Hard Problem A problem L is said to be NP-hard if for any problem L_1 in NP, there is a polynomial-time reduction of L_1 to L

Finally in Section 2.4, we discussed how some of the problems defined in Section 2.2 are established as NP-Complete.

2.7 SOLUTIONS/ANSWERS

Exercise 1: $n!/n^n = (n/n)\ ((n-1)/n)\ ((n-2)/n)\ ((n-3)/n)\ldots(2/n)(1/n)$
$= 1(1-(1/n))\ (1-(2/n))\ (1-(3/n))\ldots(2/n)(1/n)$

Each factor on the right hand side is less than equal to 1 for all value of n. Hence, The right hand side expression is always less than one.

Therefore, $n!/n^n$ 1
or, $n!$ n^n
Therefore, , $n! = O(n^n)$

Exercise 2: For large value of n, $3\log n << n^2$

Therfore, $3\log n / n^2 << 1$

$(n^2 + 3\log n)/ n^2 = 1 + 3\log n/ n^2$

or, $(n^2 + 3\log n)/ n^2 < 2$

or, $n^2 + 3\log n = O(n^2)$.

Exercise 3 : We have, $2^n/5^n < 1$
or, $2^n < 5^n$
Therefore, $2^n = O(5^n)$.

Exercise 4 : Given a set of integers, we have to divide the set in to two disjoint sets such that their sum value is equal .

A deterministic algorithm to find two disjoint sets is to list all possible combination of two subsets such that one set contain *k* elements and other contains remaining (n-k) elements. Then to check if the sum of elements of one set is equal to the sum of elments of another set. Here, the possible number of combination is C(n, *k*). This algorithm is exponential in n.

To show that the partition problem **NP**, for a given set S, we take S_1 S, S_2 S and S_1 S_2 = and verify to see if the sum of all elements of set S_1 is equal to the sum of all elements of set S_2. This verification can be done in polynomial time. Hence, the partition problem is **NP**.

Exercise 5 : The graph coloring problem is to detemine the minimum number of colors needed to color given graph G(V, E) vertices such that no two adjacent vetices has the same color. A deterministic algorithm for this requires exponential time.

If we cast the graph–coloring problem as a decision problem *i.e.* Can we color the graph G with k-colors such that no two adjacent vertices have same color ? We can verify that if this is possible then it is possible in polynomial time.

Hence, The graph –coloring problem is **NP**.

Exercise 6 : An independent set is defined as a subset of a vertices in a graph such that no two vertices are adjacent.

The independent set problem is the optimization problem of finding an independent set of maximum size in a graph. The problem can also be stated as a decision problem :

INDEPENDENT-SET = {<*G*, *k*>| *G* has an independent set of atleast size k }.

A deterministic algorithm to find an independent set in a graph is to list all subsets of vertices of size *k* and check each one to see whether it forms an independent set. This algorithm is exponential in *k.*

Proof : To show that the independent set problem **NP**, for a given graph G = (V, E), we take V' V and verifies to see if it forms an independent set. Verification can be done by checking for u V' and v V', does (u,v) E . This verification can be done in polynomial time.

Now, We show that clique problem can be transformed to independent set problem in polynomial time.The transformation is similar clique to vertex cover. This

transformation is based on the notion of the complement of a graph G. Given an undirected graph G = (V, E), we define the complement of G as G' = (V, E'), where E' = { (u, v) | (u, v) E}. *i.e* G' is the graph containing exactly those edges that are not in G. The transformation takes a graph G and *k* of the clique problem. It computes the complement G' which can be done in polynomial time.
To complete the proof, we can show that this transformation is indeed reduction : the graph has a clique of size *k* if and only if the graph G' has an independent set of size |V| - *k*.

Suppose that G has a clique V' V with |V'| = *k*. We claim that V – V' is an independent set in G'. Let (u, v) be any edge in E'. Then, (u, v) E, which implies that atleast one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E. Equivalently, atleast one of u or v is in V – V', which means that edge (u, v) is covered by V – V'. Since (u, v) was chosen arbitrarily from E', every edge of E' is covered by a vertex in V – V'. So, either u or v is in V – V' and no two adjacent vertices are in V – V'. Hence, the set V – V', which has size |V| - *k*, forms an independent set for G'.

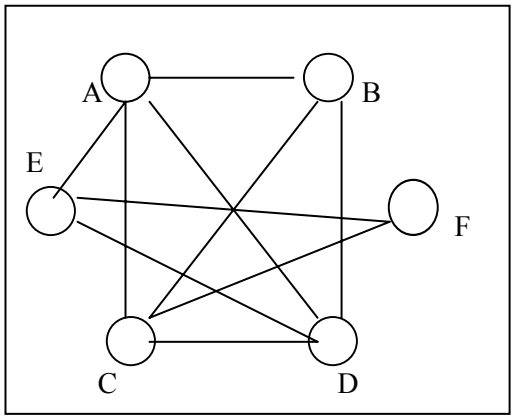


Fig. 2.5

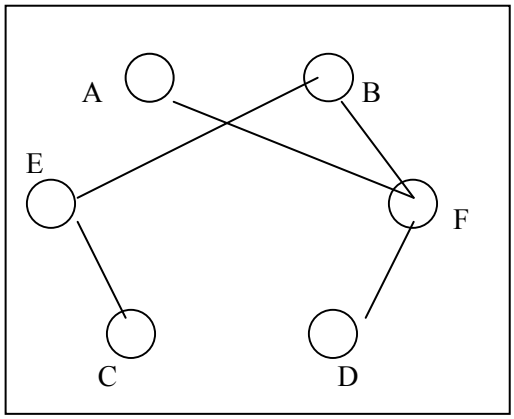


Fig. 2.6

For example, The graph G(V,E) has a clique {A, B, C, D} given by Figure 2.5. The complement of graph G is given by G' shown as Figure 2.6 and have independent set given by {E F}

This transformation can be performed in polynomial time. This proves that finding the independent set problem is **NP-Complete**.

Exercise 7

Proof : To show that travelling salesman problem **NP**, we show that verification of the problem can be done in polynomial time. Given a constant M and a closed circuit path of a weighted graph G = (V, E) . Does such path exists in graph G and total weight of such path is less than M ?, Verification can be done by checking, does (u,v) E and the sum of weights of these edges is less than M. This verification can be done in polynomial time.

Now, We show that Hamiltonian circuit problem can be transformed to travelling problem in polynomial time. It can be shown that , Hamiltonian circuit problem is a special case of the travelling salesman problem. Towards this goal, given any Graph G(V, E), we construct an instance of the |V|-city Travelling salesman by letting $d_{ij} = 1$ if (v_i, v_j) E, and 2 otherwise. We let the cost of travel M equal to |V|. It is immediate that there is a tour of length M or less if and only if there exists a Hamiltonian circuit in G.
Hence, The travelling salesman is **NP-complete**.

2.8 FURTHER READINGS

1. H.R. Lewis & C.H.Papadimitriou: *Elements of the Theory of computation*, PHI, (1981).

2. J.E. Hopcroft, R.Motwani & J.D.Ullman: *Introduction to Automata Theory, Languages*, and Computation (II Ed.) Pearson Education Asia (2001).

3. J.E. Hopcroft and J.D. Ullman: *Introduction to Automata Theory, Language, and Computation,* Narosa Publishing House (1987).

4. J.C. Martin: *Introduction to Languages and Theory of Computation*, Tata-Mc Graw-Hill (1997).

5. K.N. Rosen: *Discrete Mathematics and Its Applications* (*Fifth Edition*) Tata McGraw-Hill (2003).

6. T.H. Coremen, C.E. Leiserson & C. Stein: *Introduction to Alogrithms* (*Second Edition*) Prentice – Hall of India (2002).