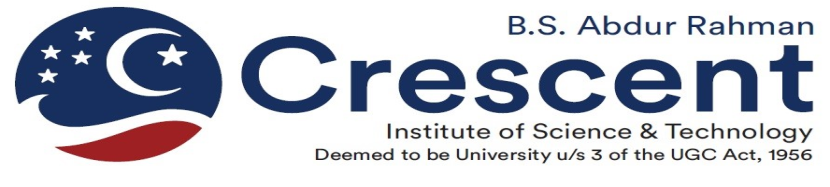
****

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**CSD 2202 – ANALYSIS OF ALGORITHMS**

**DR.S.REVATHI**

**MODULE V**

**NP COMPLETE AND NP HARD PROBLEMS**

**DECIDABLE, UNDECIDABLE, TRACTABLE AND INTRACTABLE PROBLEM**

TYPES OF ALGORITHMS

Decidable Problems (Algorithm exists) Undecidable problems(No alg exists)

Tractable Intractable

If there exists at least one if a problem is not solved in polynomial time

Polynomial time algorithm then it is intractable

Ex. O(nk), O(nlogn) Ex. O(cn)

**Polynomial time** **Exponential time**

Linear search – n 0/1 knapsack – 2n

Binary search – logn TSP - 2n

Insertion sort – n2 sum of subset - 2n

Merge sort – nlogn graph coloring - 2n

Matrix multiplication – n3 Hamilton cycle - 2n

**Tractable :**

The problem could be solved in polynomial time

Ex. O(nk) – k may be 1000 also

O(n) , O(n2) are easy

* Even a problem with time complexity O(n100) also fall in the tractable category. But it takes lot of time.
* Chances are there, the people in future will reduce the time complexity
* Easy to write a program and solve it.

**Intractable :**

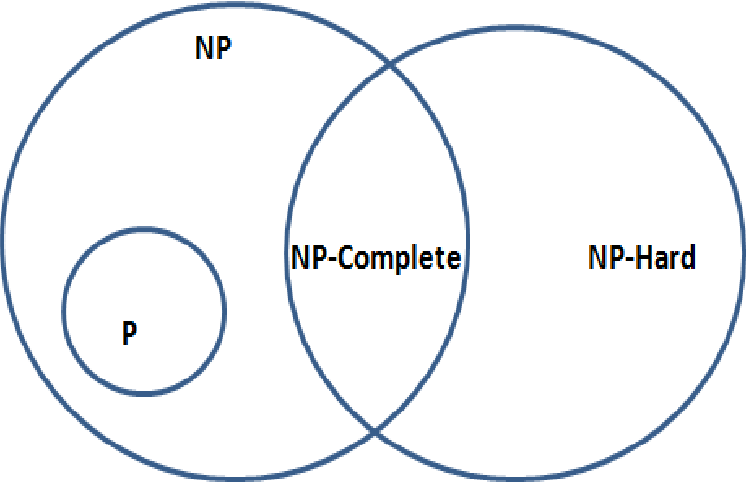
If a problem is decidable and if we don’t have any polynomial time algorithm till now, then this problem is intractable.

Ex. O(n22n) – exponential time algorithm time complexity

* If n increases, the time taken to run the problem also increases and the people difficult to run the problem stop waiting for the solution
* Solutions are exist in the world, but fail to get good solutions

P, NP, NP-HARD AND NP-COMPLETENESS

Answer : P, NP, NP-Hard and NP-Complete



**P**- **Polynomial time solving .**

Problems which can be solved in polynomial time, which take time like O(n), O(n2), O(n3).

Eg: finding maximum element in an array or to check whether a string is palindrome or not. so there are many problems which can be solved in polynomial time.

**NP**- **Non deterministic Polynomial time solving**.

Problem which can't be solved in polynomial time like TSP( travelling salesman problem) or An easy example of this is subset sum: given a set of numbers, does there exist a subset whose sum is zero?.

**Reducibility**

If we can convert one instance of a problem A into problem B (NP problem) then it means that A is reducible to B.

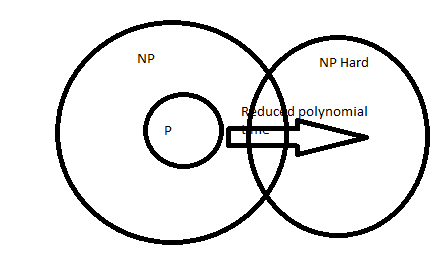
**Reduction Properties:**

1. If A is reducible to B and B in P then A in P
2. A is not in P implies B is not in P

**NP-hard**

Now suppose we found that A is reducible to B, then it means that B is at least as hard as A.

A problem is NP-hard if every problem in NP can be polynomial reduce to it.



**Definition** : A problem is classified as NP-Hard when an algorithm for solving it can be translated to solve *any*NP problem. Then we can say, this problem is *at least* as hard as any NP problem, but it could be much harder or more complex.

If every problem in NP can be polynomial time reducible to a problem ‘A’, then ‘A’ is called NP hard.

**Implication:** If A(hard) could be solved in polynomial time, then every problem in NP is “P”(in polynomial solvable time)

If the problem in NP-hard and is solved in polynomial time, then all the problem in NP also able to solve, then NP becomes P. Still it is difficult to solve

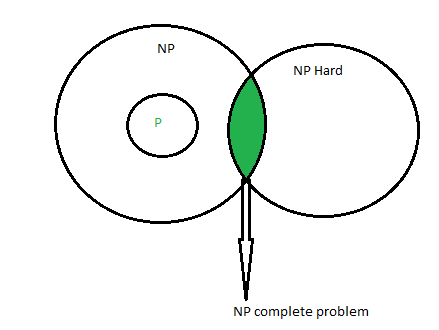
P = NP

The following problems are NP-Hard

* The circuit-satisfiability problem
* Set Cover
* Vertex Cover
* Travelling Salesman Problem

**NP-Complete**

A problem is NP-complete if it is in NP and it is in NP-hard



NP-Complete problems are problems that live in both the NP and NP-Hard classes. This means that NP-Complete problems can be verified in polynomial time and that any NP problem can be reduced to this problem in polynomial time.

The group of problems which are both in NP and NP-hard are known as NP-Complete problem.Now suppose we have a NP-Complete problem R and it is reducible to Q then Q is at least as hard as R and since R is an NP-hard problem. therefore Q will also be at least NP-hard , it may be NP-complete also.

Example of NP-complete problems

1. Integer Programming
2. Circuit-satisfiability
3. Formula satisfiability
4. 3-CNF satisfiability
5. Clique
6. Vertex cover
7. Subset-sum
8. Hamilton-cycle
9. Travelling Salesman Problem

TECHNIQUES FOR DEALING THE NP-COMPLETE PROBLEMS IN PRACTICE.

**Techniques for dealing with NP-Complete Problems in practice are**

1. Backtracking
2. Branch & bound
3. Approximation Algorithms

**Example of NP-complete problems**

* Circuit-satisfiability -Formula-atisfiability -3-CNF satisfiability
* Clique -Vertex cover -Subset-sum
* Hamilton-cycle -Travelling Salesman Problem
* Sub graph isomorphism - Integer Programming

**DECISION PROBLEMS AND OPTIMIZATION PROBLEMS**

## Optimization Problem

Optimization problems are those for which the objective is to maximize or minimize some values. For example, the types of problems which are not able to solve in polynomial time, hard.

Example 1: Travelling Salesman Problem: A graph G, what is the shortest path covering all vertices exactly once.

* When the problem is hard, then go for dynamic programming, might not able to solve in polynomial time.

Example 2: 0/1 knapsack problem: Given the capacity, profit and weight, find out the maximum profit

* Dynamic programming algorithm takes long time.
* Finding the minimum number of colors needed to color a given graph.
* Finding the shortest path between two vertices in a graph.

## Decision Problem

There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**. For example,

* Whether a given graph can be colored by only 4-colors.
* Finding Hamiltonian cycle in a graph is not a decision problem, whereas

checking a graph is Hamiltonian or not is a decision problem.

**FIRST PROBLEM PROVED AS NP-COMPLETE**

There must be some first NP-Complete problem proved by definition of NP-Complete problems.  [SAT (Boolean satisfiability problem)](http://en.wikipedia.org/wiki/Boolean_satisfiability_problem)is the first NP-Complete problem proved by Cook.

**THE TRAVELLING SALESPERSON IS NP COMPLETE PROBLEM.**

**TSP is NP-Complete**

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

**Proof**

To prove ***TSP is NP-Complete***, first we have to prove that ***TSP belongs to NP***. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus ***TSP belongs to NP***.

Secondly, we have to prove that ***TSP is NP-hard***. To prove this, one way is to show that ***Hamiltonian cycle ≤p TSP*** (as we know that the Hamiltonian cycle problem is NPcomplete).

Assume ***G = (V, E)*** to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph ***G' = (V, E')***, where

E′={(i,j):i,j∈Vandi≠jE′={(i,j):i,j∈Vandi≠j

Thus, the cost function is defined as follows −

t(i,j)={01if(i,j)∈Eotherwiset(i,j)={0if(i,j)∈E1otherwise

Now, suppose that a Hamiltonian cycle ***h*** exists in ***G***. It is clear that the cost of each edge in ***h*** is **0** in ***G'*** as each edge belongs to ***E***. Therefore, ***h*** has a cost of **0** in ***G'***. Thus, if graph ***G*** has a Hamiltonian cycle, then graph ***G'*** has a tour of **0** cost.

Conversely, we assume that ***G'*** has a tour ***h'*** of cost at most **0**. The cost of edges in ***E'*** are **0** and **1** by definition. Hence, each edge must have a cost of **0**as the cost of ***h'*** is **0**. We therefore conclude that ***h'*** contains only edges in ***E***.

We have thus proven that ***G*** has a Hamiltonian cycle, if and only if ***G'*** has a tour of cost at most **0**. TSP is NP-complete.

**VERIFICATION ALGORITHM**

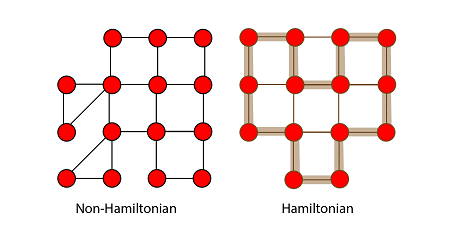
Verification algorithm is any algorithm which verifies whether the answer is right or not(wrong).

An important aspect of any algorithm is that it is correct: it always produces the expected output for the range of inputs and it eventually terminates.

As it turns out, it's difficult to prove that an algorithm is correct. Programmers often use empirical analysis to find faults in an algorithm, but only formal reasoning can prove total correctness.

Consider the Hamiltonian cycle problem. Given an undirected graph G, does G have a cycle that visits each vertex exactly once? There is no known polynomial time algorithm for this dispute.

**Note: -** It means you can't build a Hamiltonian cycle in a graph with a polynomial time even if there is no specific path is given for the Hamiltonian cycle with the particular vertex, yet you can't verify the Hamiltonian cycle within the polynomial time



Let us understand that a graph did have a Hamiltonian cycle. It would be easy for someone to convince of this. They would similarly say: "the period is hv3, v7, v1....v13i.

We could then inspect the graph and check that this is indeed a legal cycle and that it visits all of the vertices of the graph exactly once. Thus, even though we know of no efficient way to solve the Hamiltonian cycle problem, there is a beneficial way to verify that a given cycle is indeed a Hamiltonian cycle.

**Note:-**For the verification in the Polynomial-time of an undirected Hamiltonian cycle graph G. There must be exact/specific/definite path must be given of Hamiltonian cycle then you can verify in the polynomial time

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_