Polynomial means something like (x^k)
Exponential means something like (k^x)
where k is constant and x is a variable.

TRACTABLE & INTRACTABLE PROBLEMS

★ Tractable Problem:- A problem which can be solved in polynomial time or there is an efficient algorithm that solves it in polynomial time, is known as tractable problem.

★ Intractable Problem:- A problem that cann't be solved in polynomial time or there is no efficient algorithm to solve it in polynomial time, is known as intractable problem.

★ Types of complexity classes:-

① P-Class- A problem which can be solved in polynomial time is known as "P-class" problem.
Ex- sorting / searching.

→ P-class problems can be solved & verified in polynomial time.
→ It is easy to solve and easy to verify.

② NP-class (Non-deterministic Polynomial time):

A problem which can not be solved in polynomial time but can be verified in polynomial time, known as NP-class problem.

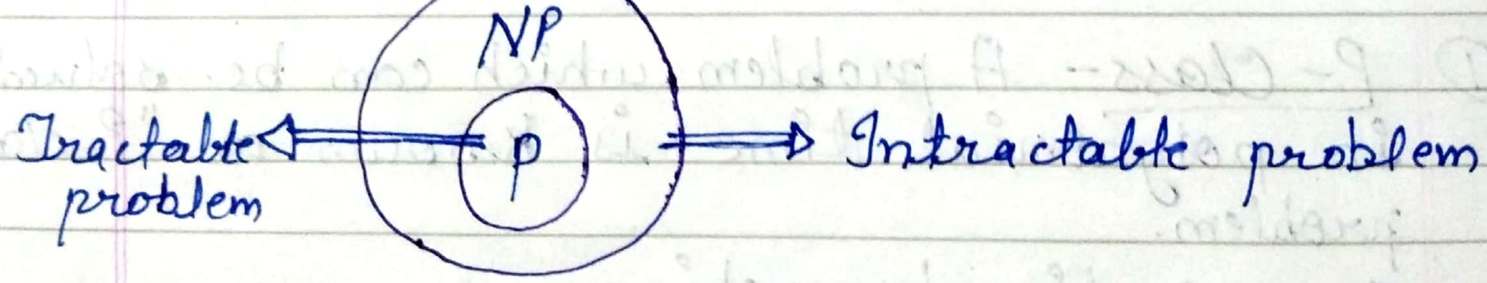
OR

NP is a set of problems that can be solved in polynomial time using nondeterministic algorithm.

NP problems can be verified in polynomial time.

Ex:-

0/1 knapsack problem, su-do-ku, TSP etc.



$\boxed{P \subseteq NP}$

★ Reduction-

Let A and B are two problems then problem A reduces to problem B iff there is a way to solve A by deterministic algorithm that solve B in polynomial time.

$$\begin{tikzcd} A \arrow[r, "\text{Reduce}", "\text{Polynomial time}"'] & B \end{tikzcd}$$

If A is reducible to B we denote it by $\boxed{A \propto B}$

Properties-

① If A is reducible to B and B is in P then A in P.

② A is not in P implies B is not in P.

③ NP-Hard-

A problem is NP-hard if every problem in NP can be polynomially reduced to it.

④ NP-Complete-

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

"The class of NP-complete problem is the intersection of the NP and NP-hard class"

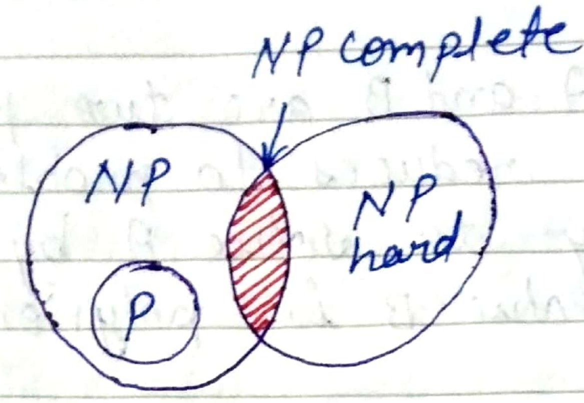


Fig - Relation between P, NP, NP-hard and NP- complete.

Decidability and Undecidability :-

$$\begin{tikzcd}
 & \text{PROBLEM} \arrow[dl] \arrow[dr] & \\
\text{Solvable} \arrow[d] \arrow[dr] & & \text{Unsolvable} \\
\text{Decidable } \{\text{algo \& procedure}\} & \text{Undecidable } \{\text{only procedure}\} &
\end{tikzcd}$$

★ Solvable :-

→ A problem is said to be soluable if we find a solution.

→ A problem is also soluable if we can prove mathematically "problem is not solvable."

★ Unsolvable :-

A problem is unsolvable neither we can solve the problem nor we are in condition to say that problem can not be solved.

★ Decidable Problem-

A problem is said to be decidable if we can always construct a corresponding algorithm that can answer the problem correctly. It means a problem is said to be decidable if there exists both algorithm and procedure for it.

Decidability in terms of Turing Machine- A problem is said to be decidable problem if there exists a corresponding T.M which halts on every input with an answer yes/NO. These problems are termed as Turing decidable since TM always halts on every input, accepting or rejecting it.

Ex-

① Acceptance problem for DFA- Given a DFA does it accept a given string?

② Emptiness problem for DFA- Given a DFA does it accept any word?

③ Equivalence problem for DFA- Given two DFAs, do they accept same language?

PAGE NO. :

★ Partially decidable or Semi-decidable Problems –

Semi-decidable problems are those for which a TM halts on the input accepted by it but it can either halt or loop forever on the input which is rejected by the TM. Such problems are termed as Turing recognizable problem.

Ex –

① Membership of CFL
② Emptiness of CFL.

★ Undecidable Problems –

The problem for which we cann't construct an algorithm that can answer the problem correctly in finite time are termed as undecidable problems. These problems may be partially decidable but they will never be decidable i.e. there will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer at all.

Ex –

① Whether a CFG generates all the strings or not?
→ As a CFG generates infinite strings, we can't ever reach up to the last string and hence it is undecidable.

Scanned with CamScanner

PAGE NO.:

② whether two CFL L & M equal?
→ since we cann't determine all the strings of any CFG, we can predict that two CFG are equal or not.

③ Ambiguity of CFG?
→ There exist no algorithm which can check whether for the ambiguity of a CFL. we can only check if any particular string of the CFL generates two different parse trees then the CFL is ambiguous.

④ Membership problem of a TM.

⑤ Finiteness of a TM

⑥ Emptiness of a TM.

Rice's Theorem
Every non-trivial (answer is not known) problem on Recursive Enumerable languages is undecidable.e.g.; If a language is Recursive Enumerable, its complement will be recursive enumerable or not is undecidable.

Cook's Theorem

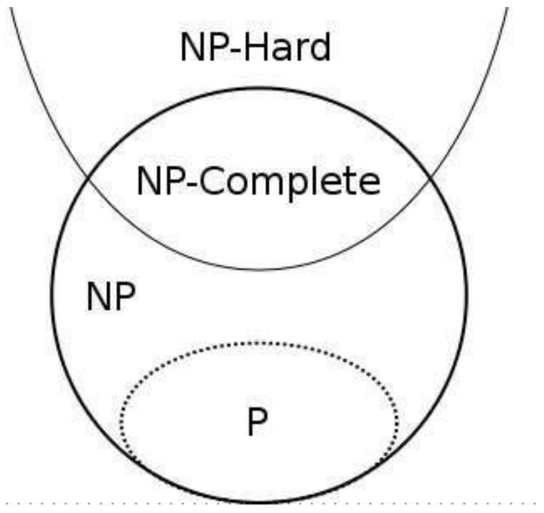
It states that Any NP problem can be converted to SAT in polynomial time

If a set **S** of strings is accepted by some non-deterministic Turing machine within polynomial time then it is NP-complete.

What are NP, P, NP-complete and NP-Hard problems?

P is set of problems that can be solved by a deterministic Turing machine in **P**olynomial time. NP is set of decision problems that can be solved by a **N**on-deterministic Turing Machine in **P**olynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time). Informally, NP is set of decision problems which can be solved by a polynomial time via a "Lucky Algorithm", a magical algorithm that always makes a right guess among the given set of choices.

NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:
1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).
A problem is NP-Hard if it follows property 2 mentioned above, doesn't need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.



Decision vs Optimization Problems

NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems.

Optimization Problem

Optimization problems are those for which the objective is to maximize or minimize some values. For example,

- 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.

What is Reduction?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether y belongs to L_2 or not. The idea is to find a transformation from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.

Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra's algorithm to find shortest path, we can take log of all weights and use Dijkstra's algorithm to find the minimum product path rather than writing a fresh code for this new problem.

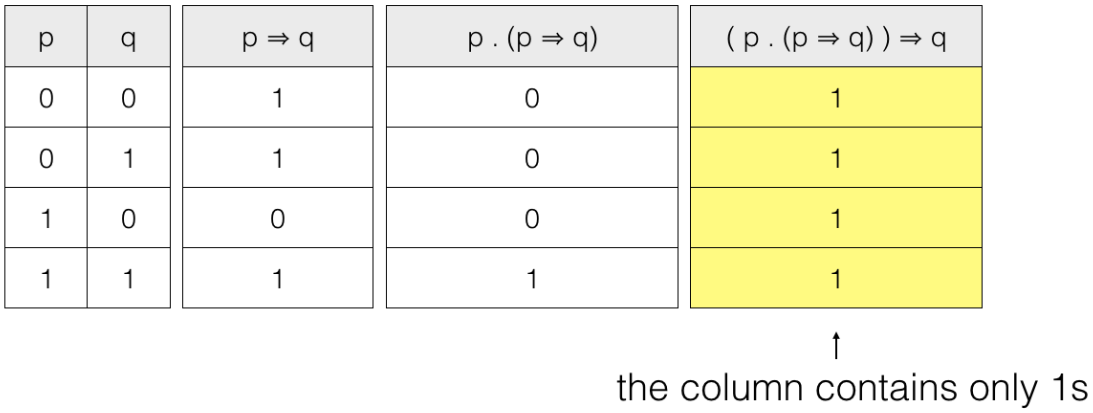
How to prove that a given problem is NP complete?
From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

The 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) is the first NP-Complete problem proved by Cook

SAT problem(Boolean satisfiability problem)----

It is a formula defined inductivily over Boolean variables and the gates AND ,OR & NOT.
For example x′y′z′+xy′z+xyz′+xyz is a Boolean expression

A truth assignment is defined as value of each variable assigned as 0 or 1 resulted to be 1for every variable.



p	q	$p \Rightarrow q$	$p . (p \Rightarrow q)$	$(p . (p \Rightarrow q)) \Rightarrow q$
0	0	1	0	1
0	1	1	0	1
1	0	0	0	1
1	1	1	1	1

↑
the column contains only 1s

3-SAT problem- { Q | Q is a 3CNF Boolean satisfiability variable}

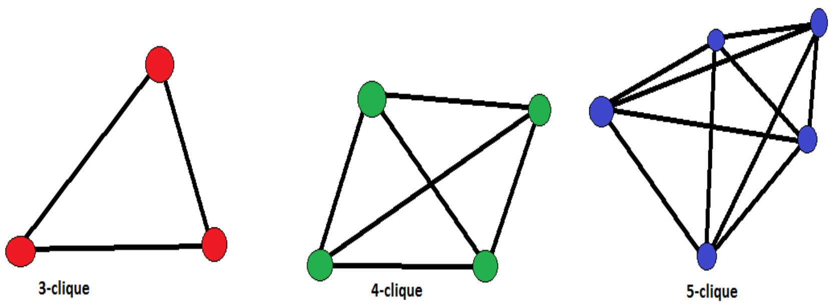
Note---- If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

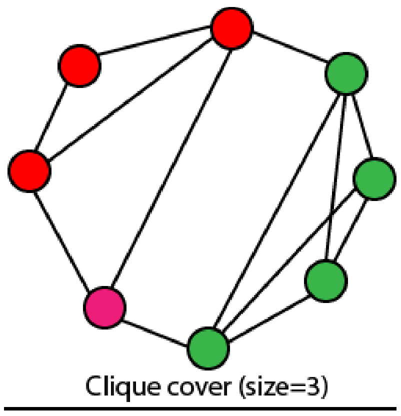
Problems

1. Clique decision Problem-

Clique--

Definition: - In Clique, every vertex is directly connected to another vertex, and the number of vertices in the Clique represents the Size of Clique.





Analysis

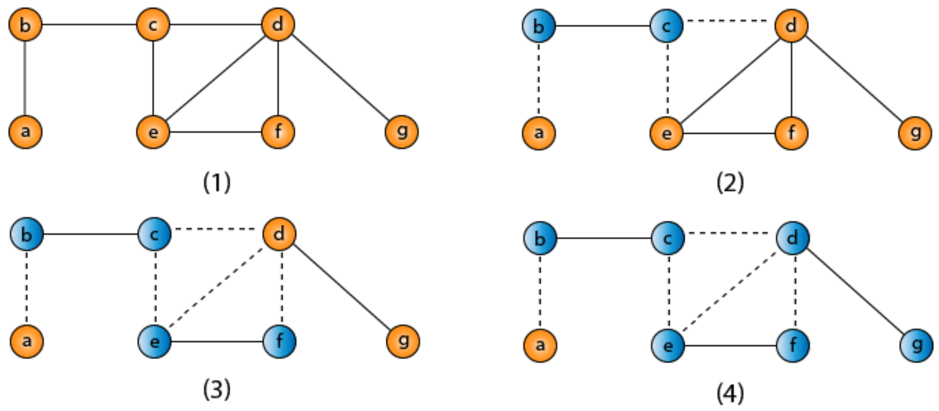
Max-Clique problem is a non-deterministic algorithm. In this algorithm, first we try to determine a set of **k** distinct vertices and then we try to test whether these vertices form a complete graph.

There is no polynomial time deterministic algorithm to solve this problem. This problem is NP-Complete.

2. <u>Vertex cover problem</u>

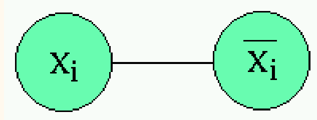
A Vertex Cover of a graph G is a set of vertices such that each edge in G is incident to at least one of these vertices.

The decision vertex-cover problem was proven NP-Complete. Now, we want to solve the optimal version of the vertex cover problem, i.e., we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C*

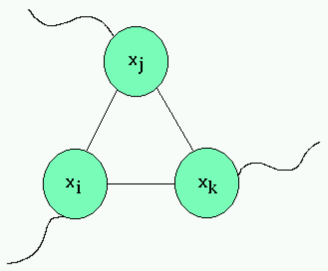


Proof:
It was proved in 1971, by Cook, that 3SAT is NP-complete. Next, we know that VERTEX COVER is in NP because we could verify any solution in polytime with a simple n^2 examination of all the edges for endpoint inclusion in the given vertex cover.

For the reduction, we are going to take an instance of 3SAT (a boolean formula) and reduce it to a vertex cover instance that has a cover if and only if the 3SAT formula has a satisfying assignment. The first thing we will do is force a choice for each variable to either True or False by having a pair of vertices for every literal and it's negation. So an x_i in U will yield two vertices in G.

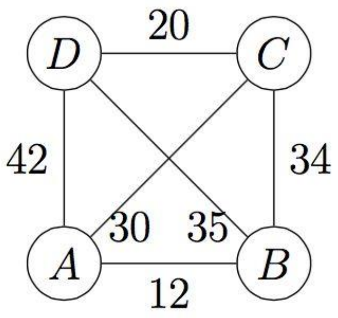


Next, we create a 'gadget' to represent the clauses. What we will do is for each clause (x_i, x_j, x_k) we'll create a three vertex triangle where the each vertex is connected to the other two and to the corresponding literal from the section above.



Finally, we must choose an appropriate K for the instance of VERTEX COVER. We'll choose l + 2m where l is the number of literals and m is the number of clauses.

3. **Travelling Salesman Problem (TSP):** Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.
   Note the difference between Hamiltonian Cycle and TSP. The Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in figure on right side.

The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.
Following are different solutions for the traveling salesman problem.

Naive Solution:
1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate cost of every permutation and keep track of minimum cost permutation.
4) Return the permutation with minimum cost.
Time Complexity: Θ(n!)

Using the above recurrence relation, we can write dynamic programming based solution. There are at most $O(n*2^n)$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2*2^n)$. The time complexity is much less than O(n!), but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

4. **Hamiltonian Path Problem**—it is a path in a directed or undirected graph that visits each vertex exactly once. The problem to check whether a graph (directed or undirected) contains a Hamiltonian Path is NP-complete, so is the problem of finding all the Hamiltonian Paths in a graph. Following images explains the idea behind Hamiltonian Path more clearly.

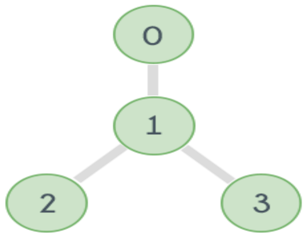


Fig. 1

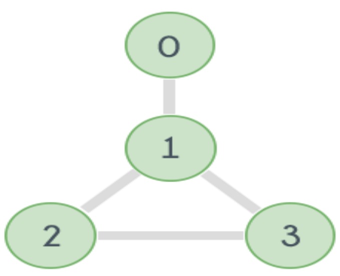


Fig. 2

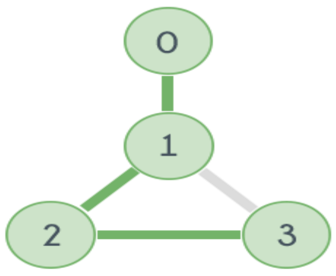


Fig. 3

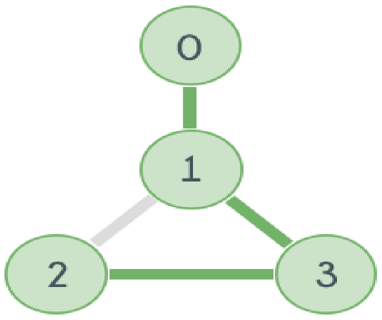


Fig. 4

In general, the problem of finding a Hamiltonian path is NP-complete , so the only known way to determine whether a given general graph has a Hamiltonian path is to undertake an exhaustive search.