Introduction to Complexity Theory

The field of complexity theory deals with how fast can one solve a certain type of problem. Or more generally how much resource does it take: time, memory-space, number of processors etc. The most common resource is time: number of steps. This is generally computed in terms of n, the length of the input string. We will use an informal model of a computer and an algorithm. All the definitions can be made precise by using a model of a computer such as a Turing Machine

CLASSES OF PROBLEMS

We can categorize the problems into the following broad classes

1. Problems which cannot even be defined formally.
2. Problems which can be formally defined but cannot be solved by computational means.
3. problems which, though theoretically can be solved by computational means, yet are infeasible ie., these problems require so-large amount of computational resources that practically is not feasible to solve these problems by computational means.
   These problems are called **Intractable** or **infeasible problems**
4. 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 solved the problem having time-complexity as a polynomial function of the size of the problem. The class of problem is denoted by P
5. Last class Includes large number of problems for each of which it is not known whether It is In P or not in P.

Decision Problems
6.
Decision problems are the computational problems for which the intended ouput is either "yes" or "no". In other words, a decision problem is a problem with yes / no answers. Hence in a decision problem, we can equivalently talk of the language associated with the decision problem, namely, the set of inputs for which the answer is yes.

Typically, we assume that the input is coded in binary, so the set of all possible inputs is $\{0, 1\}^*$ and the language associated with a decision problem Q is

$$L(Q) = \{x \in \{0, 1\}^* \mid \text{the answer is yes for problem Q on input } x\}$$

The classes P and NP:

The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem. Most of the problems examined in previous modules are in P.

The class NP consists of those problems that are 'verifiable' in polynomial time. If we were somehow given a 'certificate' of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.

Example:

PATH
INPUT: graph G, nodes a and b
Question: Is there a path from a to b in G?

This problem is in P. To see if there is a path from node a to node b, one might determine all the nodes reachable from a by doing for instance a breadth-first search or Dijkstra's algorithm.

Verification Algorithm:

A verification algorithm is an algorithm A, that takes two inputs: an ordinary input x, and a certificate y, and outputs a 1 on certain combinations of x and y.

Verification algorithm A verifies an input string x if there exists a certificate y such that

$A(x, y) = 1$.

The language verified by verification algorithm A is

$$L = \{\text{input string } x \mid \text{there exists certificate string } y \text{ such that } A(x,y) = 1\}$$

Example:

In the hamiltonian cycle problem, given a directed graph $G = (V, E)$, a certificate would be a sequence $\langle v_1, v_2, v_3 \ldots v_{|V|} \rangle$ of $|V|$ vertices. We could easily check in polynomial time that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, 3 \ldots |V| - 1$ and that $(v_{|V|}, v_1) \in E$ as well.

Polynomial-time Verification Algorithm:

A verification algorithm A for a language L is a polynomial-time verification algorithm for L if

- for each $x \in L$, there is a certificate y of size polynomial in the size of x such that $A(x, y) = 1$, and $A(x,y)$ returns 1 in time Polynomial in x.
- Since A is a verification algorithm for L, for every x not in L there is no certificate y for which $A(x,y) = 1$.

P, NP and NP-complete :

Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate. So we can believe that $P \subseteq NP$. The open question is whether or not P is a proper subset of NP.

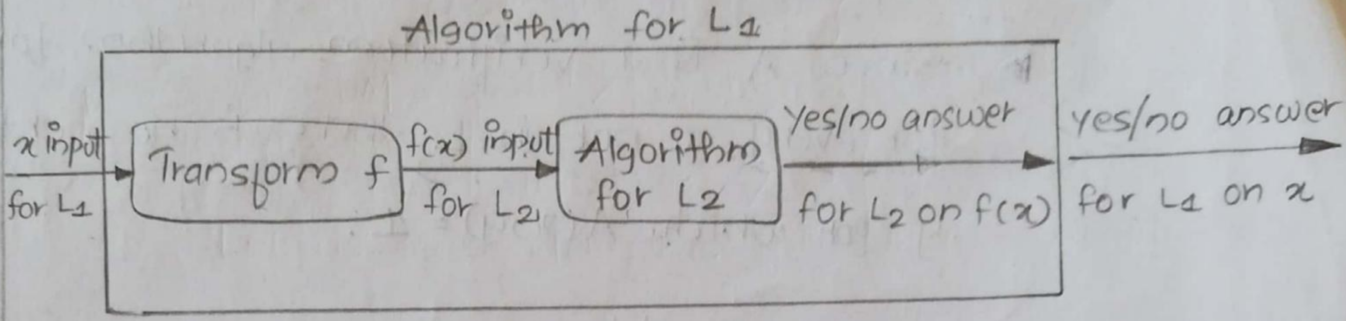
Pg no 6. PTO

Informally, a problem is in the class NPC — and we refer to it as being NP-complete — if it is in NP and is as "hard" as any problem in NP. If any NP-complete problem can be solved in polynomial time, then every problem in NP has a polynomial time algorithm.

Reduction:

Let L_1 and L_2 be two decision problems. Suppose algorithms 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 $y \in L_2$ or not.

The idea is to find a transformation f from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.



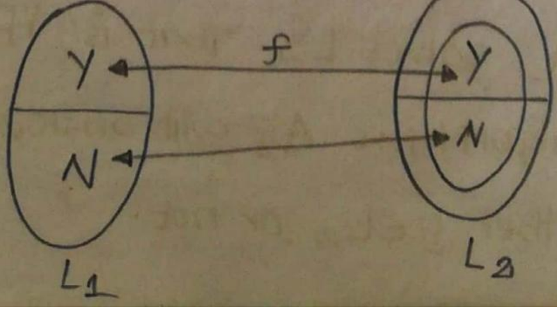
Polynomial – time Reduction:

Let L_1 and L_2 be languages that are subsets of $\{0,1\}^*$. We say that L_1 is polynomial – time reducible to L_2 if there exists a function f

$$f : \{0,1\}^* \longrightarrow \{0,1\}^*$$

with the following properties.

- f transforms an input x for L_1 into an input $f(x)$ for L_2 such that $f(x)$ is a yes – input for L_2 if and only if x is a yes input for L_1. We require a yes – input of L_1 maps to a yes-input of L_2, and a no-input of L_1 maps to a no-input of L_2.



- $f(x)$ is computable in polynomial time.

If such an f exists, we say that L_1 is Polynomial-time reducible to L_2, and write $L_1 \leq_p L_2$.

$$L_1 \leq_p L_2$$

Languages in NP:

Let us consider the following examples of decision problems.

- HAM-CYCLE $= \{\langle G\rangle \mid G$ is a Hamiltonian graph$\}$
- CIRCUIT-SAT $= \{\langle C\rangle \mid C$ is a satisfiable boolean ckt$\}$
- SAT $= \{\langle \phi\rangle \mid \phi$ is satisfiable boolean formula$\}$
- CNF-SAT $= \{\langle \phi\rangle \mid \phi$ is a satisfiable boolean formula in CNF$\}$
- 3-CNF-SAT $= \{\langle \phi\rangle \mid \phi$ is a satisfiable boolean formula in CNF$\}$
- CLIQUE $= \{\langle G, K\rangle \mid G$ is an undirected graph with a clique of size $k\}$
- IS $= \{\langle G, K\rangle \mid G$ is an undirected graph with an independent set of size $k\}$
- VERTEX-COVER $= \{\langle G, K\rangle \mid$ undirected graph G has a vertex cover of size $k\}$

- $TSP = \{\langle G, c, k\rangle \mid G = (V, E)$ is a complete graph, $c : V \times V \to \mathbb{Z}$ is a cost function, $k \in \mathbb{Z}$ and G has a traveling salesman tour with cost at most $k\}$
- $\text{SUBSET-SUM} = \{\langle S, t\rangle \mid$ there is a subset $S' \subseteq S$ such that $t = \sum_{s \in S'} s\}$

Is P = NP?

One of the most important problems in computer science is whether P=NP or P≠NP? Observe that P⊆NP. Given a problem $A \in P$, and a certificate to verify the ~~validate~~ validity of a yes-input (an instance of A), we can simply solve A in polynomial time (since $A \in P$). It implies $A \in NP$.

Intuitively, NP⊆P is doubtful. After all, just able to verify a certificate in polynomial time does not necessary mean we can able to tell whether an input is an yes-input of no-input in polynomial time.

However, 30 years after the P=NP? problem was first proposed, we are still no closer to solving it and do not know the answer. The search for a solution though, has provided us with deep insights into what distinguishes an 'easy' problem ~~for~~ from a 'hard' one.

The class CO-NP:

$L \in NP$
$\overline{L} \in NP$

Note that if $L \in NP$, there is no guarantee that $\overline{L} \in NP$ (since having certificates for yes-inputs, does not mean that we have certificates for the no-inputs).

The class of decision problems L such that $\overline{L} \in NP$ is called CO-NP.

Prime $\notin NP$ 2, 3, 5, 7
$\overline{\text{composite}} \in NP$

Example: COMPOSITE $\in NP$ so PRIME $= \overline{\text{COMPOSITE}} \in$ CO-NP

$\boxed{256}$

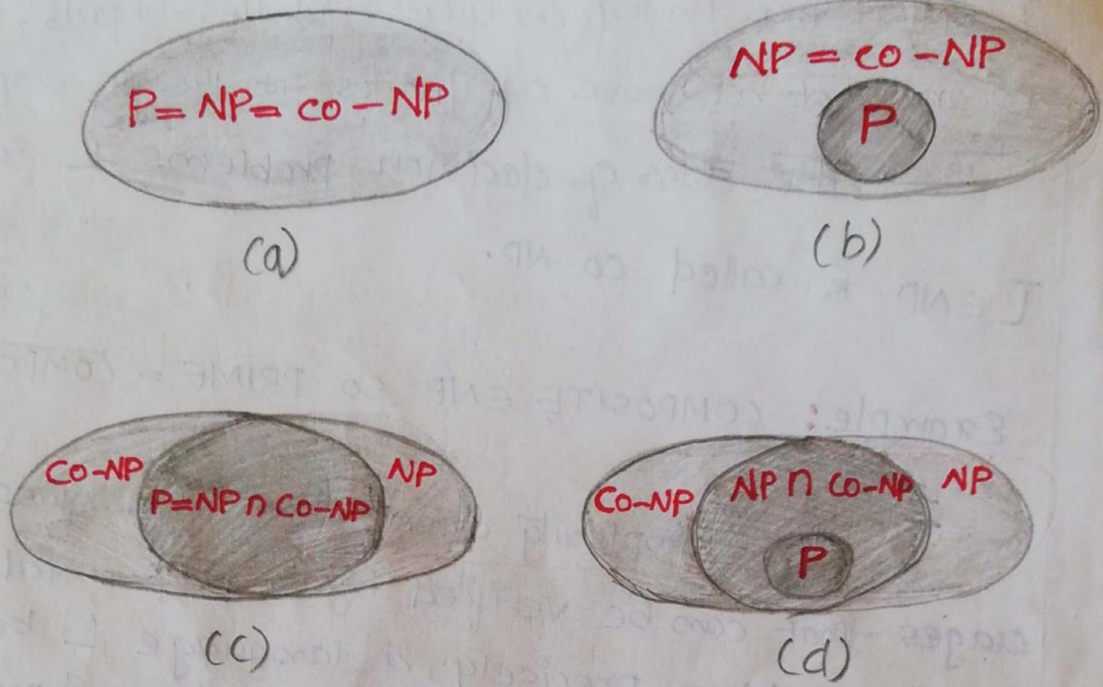
The complexity class NP is the class of languages that can be verified by a polynomial-time algorithm. More precisely, a language L belongs to NP if and only if there exist a two-input polynomial-time algorithm 'A' and a constant 'c' such that

$$L = \left\{ x \in \{0,1\}^* : \begin{array}{l} \text{there exist a certificate } y \text{ with} \\ |y| = O(|x|^c) \\ \text{such that } A(x,y) = 1 \end{array} \right\}.$$

We say that algorithm A verifies language L in polynomial time.

We can define the complexity class CO-NP as the set of languages L such that $\overline{L} \in NP$. Once again, no one knows whether $P = NP \cap$ CO-NP or whether there is some language in $NP \cap$ CO-NP $-$ P

Possibilities for relationships among complexity clas



In each diagram, one region enclosing another indicates a proper-subset relation.

(a) P = NP = Co-NP. Most researchers regard this possibility as the most unlikely.

(b) If NP is closed under complement, then NP = Co-NP, but it need not be the case that P = NP.

(c) P = NP ∩ co-NP, but NP is not closed under complement.

(d) NP ≠ Co-NP and P ≠ NP ∩ Co-NP. Most researchers regard this possibility as the most likely.

NP-Hard:

A language $L \subseteq \{0,1\}^*$ is NP-complete if

1. $L \in NP$, and
2. $L' \leq_p L$ for every $L' \in NP$

If a language L satisfies property 2, but not necessarily property 1, we say that L is NP-Hard.

NP-hardness is a class of problems that are, informally, "at least as hard as the hardest problems in NP". More precisely, a problem H is NP-Hard when every problem L in NP can be reduced in polynomial time to H.

Theorem:

If any NP-complete problem is polynomial time solvable, then P=NP. If any problem in NP is not polynomial time solvable, then all NP-complete problems are not polynomial time solvable.

Proof:

Suppose that $L \in P$ and also that $L \in NPC$. For any $L' \in NP$, we have $L' \leq_p L$ by property 2 of the definition of NP-completeness.

A language $L \subseteq \{0,1\}^*$ is NP complete if it satisfies the following two properties:

1. $L \in NP$; and
2. For every $L' \in NP$, $L' \leq_p L$

We use the notation $L \in NPC$ to denote that L is NP-complete.

We know if $L' \leq_p L$ then $L \in P$ implies $L' \in P$, which proves the first statement.

To proves the second statement, suppose that there exists an $L \in NP$ such that $L \notin P$. Let $L' \in NPC$ be any NP-complete language, and for the purpose of contradiction, assume that $L' \in P$. But then we have $L \le_p L'$ and thus $L \in P$.

Proving NP-completeness:

To prove that a problem P is NP-complete, we have following methods:

Method 1: (direct proof)

(a) P is in NP

(b) All problems in NP-complete can be reduced to P.

Method 2: (equivally general but potentially easier)

(a) P is in NP

(b) Find a problem P' that has already been proven to be in NP-complete

(c) Show that $P' \le P$.

NP-Complete Problems

Examples of NP-complete problems

1. Formula Satisfiability
2. Circuit satisfiability
3. 3-CNF Satisfiability
4. clique
5. verlex cover
6. Subset-Sum
7. Hamiltonian cycle
8. Traveling Salesman

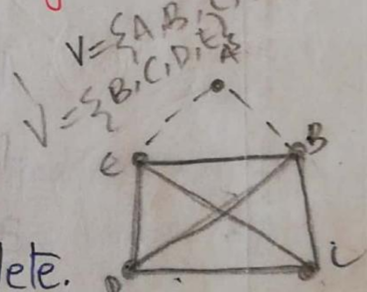
Clique:

A clique in an undirected graph $G=(V,E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The size of a clique is the number of vertices it contains. The clique problem is the optimization problem of finding a clique of maximum size in a graph.

As a decision problem, we ask simply whether a clique of a given size k exists in the graph.

The formal definition is

CLIQUE = {<G, k> : G is a graph containing a clique of size k}



clique of size 4

Theorem

The clique problem is NP-complete.

Proof

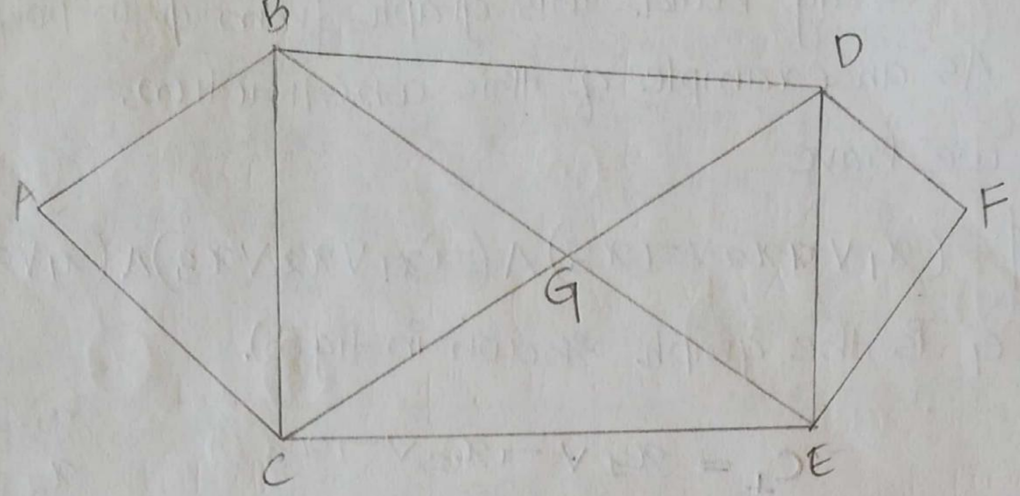
To show that CLIQUE ∈ NP, for a given graph $G=(V,E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for G. We can check whether V' is a clique in polynomial time by checking whether, for each pair $u, v \in V'$, the edge (u,v) belongs to E.

Example:

Fig(a)

$V' = \{A, B, D, E, C, A\}$ ✓ → CLIQUE

$V' = \{A, B, G, D, F, G\}$ ✗



Fig(a).

Next prove that 3-CNF-SAT $\leq_p$ CLIQUE, which shows that the clique problem is NP-hard.

The reduction algorithm begins with an instance of 3-CNF-SAT. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a boolean formula in 3-CNF with k clauses. For $r = 1, 2 \ldots k$, each clause C_r has exactly three distinct literals l_1^r, l_2^r, and l_3^r. We shall construct a graph G such that ϕ is satisfiable if and only if G has a clique of size k.

We construct the graph $G = (V, E)$ as follows.

For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in ϕ, we place a triple of vertices v_1^r, v_2^r, and v_3^r into V. We put an edge between two vertices v_i^r and v_j^s if both of the following hold:

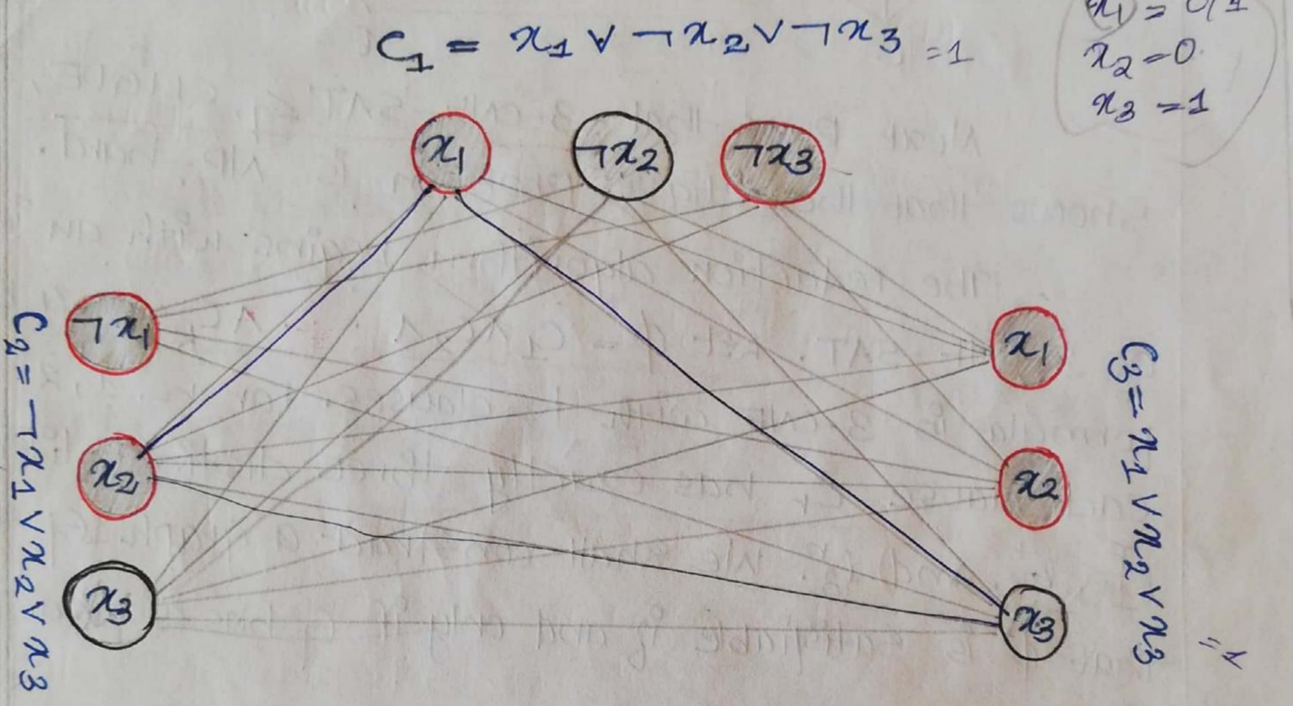
- v_i^r and v_j^s are in different triples, that is, $r \neq s$, and
- their corresponding literals are consistent, that is, l_i^r is not the negation of l_j^s.

We can easily build this graph from ϕ in polynomial time. As an example of this construction,

if we have

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

then G is the graph shown in fig (b).



Fig(b): The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE.

A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3, corresponding to the clique with lightly shaded vertices.

We must show that this transformation of ϕ into G is a reduction. First, suppose that ϕ has a satisfying assignment. Then each clause C_r contains at least one literal l_i^r that is assigned 1, and each such literal corresponds to a vertex v_i^r. Picking one such "true" literal from each clause yields a set V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$, where $r \neq s$, both corresponding literals l_i^r and l_j^s map to 1 by the given satisfying assignment, and thus the literals cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple, and so V' contains exactly one vertex per triple. We can assign 1 to each literal l_i^r such that $v_i^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Each clause is satisfied, and so ϕ is satisfied.