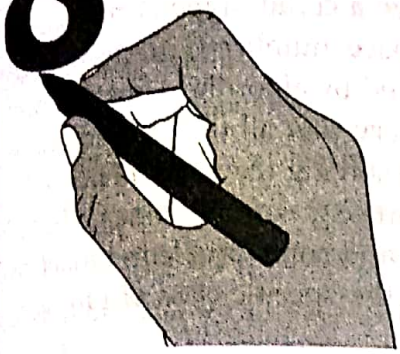
Chapter 8



NP COMPLETENESS

CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Tractable and Intractable Problems, Concept of Polynomial Time and Super Polynomial Time Complexity
- Complexity Classes: P, NP, NP-Hard and NP-Complete
- NP Complete Problems, NP Completeness and Reducibility, Cooks Theorem, Proofs of NP Completeness (CNF-SAT, Vertex Cover and Subset Sum)
- Approximation Algorithms: Concept, Vertex Cover Problem, Subset Sum Problem

Introduction

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. Up to now we were considering on the problems that can be solved by algorithms in worst-case polynomial time. There are many problems and it is not necessary that all the problems have the apparent solution. This concept, somehow, can be applied in solving the problem using the computers. The computer can solve: some problems in limited time e.g. sorting, some problems requires unmanageable amount of time e.g. Hamiltonian cycles, and some problems cannot be solved e.g. Halting Problem. In this section we concentrate on the specific class of problems called NP complete problems.

Tractable and Intractable Problems

We call problems as **tractable** or easy, if the problem can be solved using polynomial time algorithms. The problems that cannot be solved in polynomial time but requires super-polynomial time algorithm are called **intractable** or hard problems. There are many problems for which no algorithm with running time better than exponential time is known some of them are, travelling salesman problem, Hamiltonian cycles, and circuit satisfiability, etc.
Here are examples of tractable problems (ones with known polynomial-time algorithms):

- Searching an unordered list
- Searching an ordered list
- Sorting a list
- Multiplication of integers (even though there's a gap)
- Finding a minimum spanning tree in a graph (even though there's a gap)

euta max find garne algo dekhaune

Here are examples of intractable problems (ones that have been proven to have no polynomial-time algorithm). Some of them require a non-polynomial amount of output, so they clearly will take a non-polynomial amount of time e.g.:

- **Towers of Hanoi:** we can prove that any algorithm that solves this problem must have a worst-case running time that is at least $2^n - 1$.
- List all permutations (all possible orderings) of n numbers.

toh ko algo tutorials point
ma xa toh(disk-1,source,aux,dest) esto xa
vne disk lai source bata auxiliary ma lagne
vneko

Concept of Polynomial Time and Super Polynomial Time Complexity

Let's start by reminding ourselves of some common functions, ordered by how fast they grow.

Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
n-log-n	$O(n \times \log n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(k^n)$, e.g. $O(2^n)$
Factorial	$O(n!)$
Super-exponential	$O(n^n)$

Computer Scientists divide these functions into two classes:

Polynomial Running Time

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is $O(n^k)$ for some non-negative integer k, where n is the complexity of the input. Polynomial-time algorithms are said to be "fast." Most familiar mathematical operations such as addition, subtraction, multiplication, and division, as well as computing square roots, powers, and logarithms, can be performed in polynomial time. Computing the digits of most interesting mathematical constants, including pi and e, can also be done in polynomial time.

Example: O(1), O(log n), O(n), O(n × log n), $O(n^2)$, $O(n^3)$, etc. belongs to polynomial time complexity.

Super Polynomial Time Complexity

An algorithm whose order-of-magnitude time performance is not bounded from above by a polynomial function of n is called super polynomial time complexity algorithm. A function of the form k^n is genuinely exponential. But now some functions which are worse than polynomial but not quite exponential, such as O(n log n), are also (incorrectly) called exponential. And some functions which are worse than exponential, such as the super exponentials, e.g. $O(n^n)$, will also (incorrectly) be called exponential. A better word than 'exponential' would be 'super-polynomial'.

Complexity Classes: P, NP, NP-Hard and NP-Complete

Class P Problem

The class P consists of those problems that can be solved by a deterministic Turing machine in Polynomial time. P problems are obviously tractable. 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.

Example: Adding two numbers is really easy. Surely, as the number gets larger the computation becomes harder to us human. But to a computer adding large numbers are fairly simple. We can say computers can add two numbers in Polynomial time. These types of problem which can be solved in polynomial time by a computer are known as **P problems.**

NP-Class

NP is set of decision problems that can be solved by a Non-deterministic Turing Machine in Polynomial 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). The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct. Every problem in this class can be solved in exponential time using exhaustive search.

Example: Let's take a little complex problem like prime factorization. We know that every composite number can be expressed as a product of two or more prime factors. Our normal PCs can handle this problem within seconds for numbers up to a billion. But as the numbers grow this problem becomes lot harder even for the fastest of computers. So this is not solvable in **Polynomial time.** So factorization is not solvable in polynomial time but the solution is verifiable in polynomial time. These problems are known as NP problems.

answer po nikalna sakiyenw tw polynomial time ma tarw answer correct ho ki haewn tyo verify garna sakihalinxani simply multiply grdyo number composite ho ki haenw chinyo

NP-Complete

NP-Complete problem is a complexity class which represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time. NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:

- L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
- Every problem in NP is reducible to L in polynomial time

NP-COMPLETE ma reduce hudae ma NP-complete vaidenw

Intuitively this means that we can solve Y quickly if we know how to solve X quickly. Precisely, Y is reducible to X, if there is a polynomial time algorithm f to transform instances y of Y to instances x = f(y) of X in polynomial time, with the property that the answer to y is yes, if and only if the answer to f(y) is yes.

Y X ma convert vairaxa ani instance of Y y vyo tyoni x ma convert hunxa so fxn tyo x=f(y) hunevyo

Examples: An interesting example is the graph isomorphism problem, the graph theory problem of determining whether a graph isomorphism exists between two graphs. Two graphs are isomorphic if one can be transformed into the other simply by renaming vertices'. Consider these two problems:

- **Graph Isomorphism:** Is graph G1 isomorphic to graph G2?
- **Sub-graph Isomorphism:** Is graph G1 isomorphic to a sub-graph of graph G2?

The Sub-graph Isomorphism problem is NP-complete. The graph isomorphism problem is suspected to be neither in P nor NP-complete, though it is in NP. This is an example of a problem that is thought to be hard, but is not thought to be NP-complete.

The easiest way to prove that some new problem is NP-complete is first to prove that it is in NP, and then to reduce some known NP-complete problem to it. Therefore, it is useful to know a variety of NP-complete problems. The list below contains some well-known problems that are NP-complete when expressed as decision problems.

afu reduce hudae ma hudewn NP-COMPLETE tarw NP-COMPLETE Problem lai nae afu ma reduce garyaesi tw vaihlyoni

- Boolean Satisfiability problem (SAT)
- Knapsack problem
- Hamiltonian path problem
- Traveling salesman problem (decision version)
- Sub-graph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Dominating set problem

NP-hard

aafu np hard huna lai np complete aafu ma reduce huan prne raexa tyoni polynomial time ma

Intuitively, these are the problems that are at lea: as hard as the NP-complete problems. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems. The precise definition here is that a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time.
But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.
Example: The halting problem is an NP-hard problem. This is the problem that given a program P and input I, will it halt? This is a decision problem but it is not in NP. It is clear that any NP-complete problem can be reduced to this one. As another example, any NP-complete problem is NP-hard.

NP-HARD huna lai NP huane prxa vnne xaenw

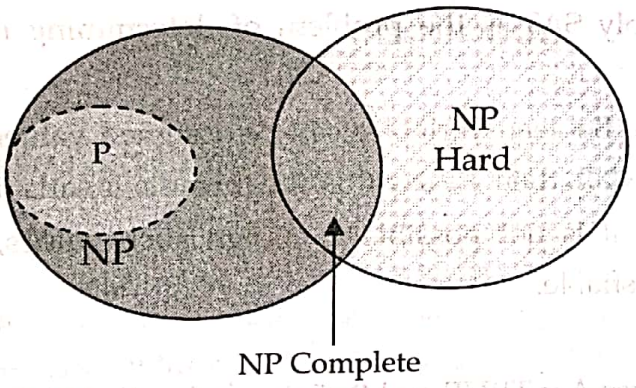


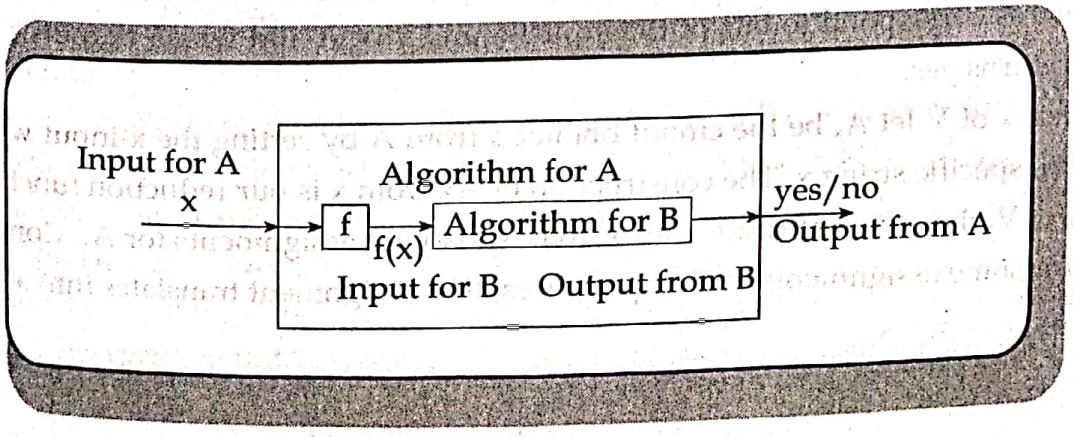
Fig: Relationships between classes P, NP, NP Complete and NP Hard

Polynomial Time Reduction

Given two decision problems A and B, a polynomial time reduction from A to B is a polynomial time function f that transforms the instances of A into instances of B such that the output of algorithm for the problem A on input instance x must be same as the output of the algorithm for the problem B on input instance f(x) as shown in the figure below. If there is polynomial time computable function f such that it is possible to reduce A to B, then it is denoted as $A \leq_p B$. The function f described above is called reduction function and the algorithm for computing f is called reduction algorithm.

esma A B ma reduce vairaxa teivyerw instances of A x raexa jun chai y ma convert vayo by polynomial time f such that y=f(x) xa y chai B ko instance vyo ani balla algorithm for B layerw chaiyeko desicion output niskinxa yo overall fer ALGORITHM OF A vyo

Cooks Theorem



In computational complexity theory, the Cook–Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable.

An important consequence of this theorem is that if there exists a deterministic polynomial time algorithm for solving Boolean satisfiability, then every NP problem can be solved by a deterministic polynomial time algorithm. The question of whether such an algorithm for Boolean satisfiability exists is thus equivalent to the P versus NP problem, which is widely considered the most important unsolved problem in theoretical computer science.

Theorem: The satisfiability problem (SAT) is NP-Complete

What is SAT?

Boolean Satisfiability or simply SAT is the problem of determining if a Boolean formula is satisfiable or unsatisfiable.

- **Satisfiable:** If the Boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the formula is satisfiable.
- **Unsatisfiable:** If it is not possible to assign such values, then we say that the formula is unsatisfiable.

yo tw boolean satisfiability ko intro matra ho

Example:

F= A Λ $\bar{B}$ is satisfiable, because A = TRUE and B=False makes F=TRUE

G= A Λ $\bar{A}$ is unsatisfiable because it gives always false result.

Proof: There are two parts to proving that the Boolean satisfiability problem (SAT) is NP-complete. One is to show that SAT is an NP problem. The other is to show that every NP problem can be reduced to an instance of a SAT problem by a polynomial-time many-one reduction.

sir ko herne

SAT is NP-hard

Proof: (This is not actual proof as given by cook, this is just a sketch)

Take a problem V ε NP, let A be the algorithm that verifies V in polynomial time (this must be true since V ε NP). We can program A on a computer and therefore there exists a logical circuit whose input wires correspond to bits of the inputs x and y of A and which outputs 1 precisely when A(x, y) returns yes.

For any instance x of V let A_x be the circuit obtained from A by setting the x-input wire values according to the specific string x. The construction of Ax from x is our reduction function. If x is a yes instance of V, then the certificate y for x gives satisfying assignments for A_x. Conversely, if A_x outputs 1 for some assignments to its input wires, that assignment translates into a certificate for x.

SAT is NP-complete

Proof: To show that SAT is NP-complete we have to show two properties as given by the definition of NP complete problems. The first property i.e. SAT is in NP Circuit satisfiability problem (SAT) is the question "Given a Boolean combinational circuit, is it satisfiable? I.e. does the circuit has assignment sequence of truth values that produces the output of the circuit as 1?" Given the circuit satisfiability problem take a circuit x and a certificate y with the set of values that produce output 1, we can verify that whether the given certificate satisfies the circuit in polynomial time. So we can say that circuit satisfiability problem is NP.

This claims that SAT is NP. Now it is sufficient to show the second property holds for SAT. The proof for the second property i.e. SAT is NP-hard is from above lemma. This completes the proof.

Approximation Algorithms

Concept

An Approximate Algorithm is a way of approach NP-Completeness for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

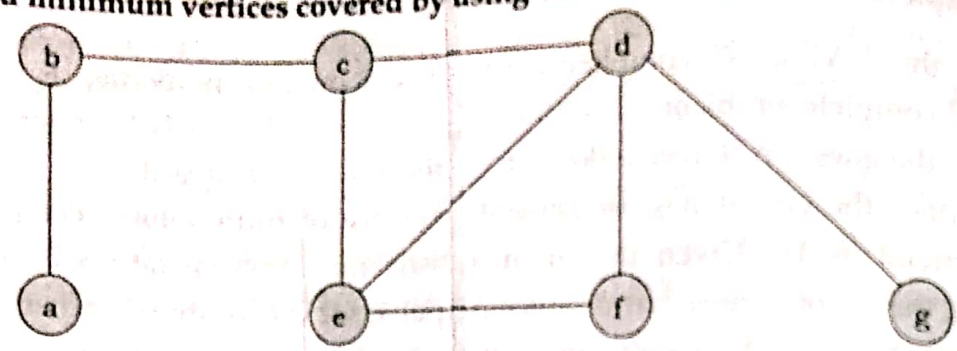
Vertex Cover Problem

An approximate algorithm is a way of dealing with NP-completeness for optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time.

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 NPC. 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*.

The idea is to take an edge (u, v) one by one, put both vertices to C, and remove all the edges incident to u or v. We carry on until all edges have been removed. C is a VC. last ma chai C chai VC(Vertex cover) ho

Example: Find minimum vertices covered by using vertex cover problem.



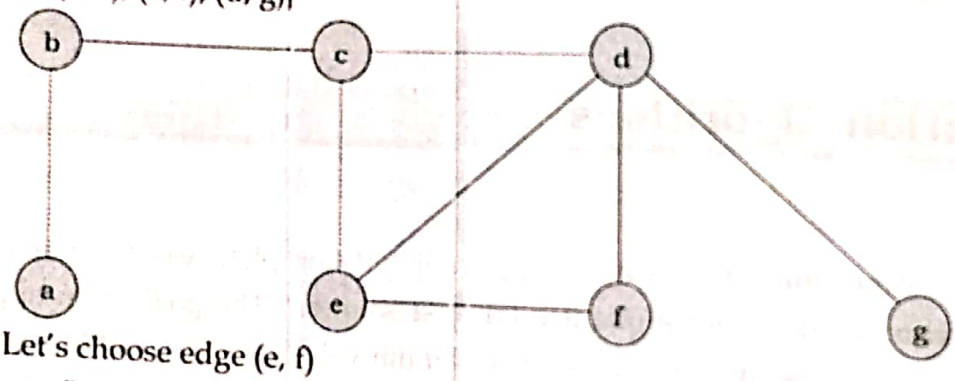
Solution:

E = {(a, b), (b, c), (c, d), (c, e), (d, e), (d, f), (e, f), (d, g)}

Step 1: Lets choose edge C = (b, c)

Eliminate edges incident to vertex b and c

E = {(d, e), (d, f), (e, f), (d, g)}

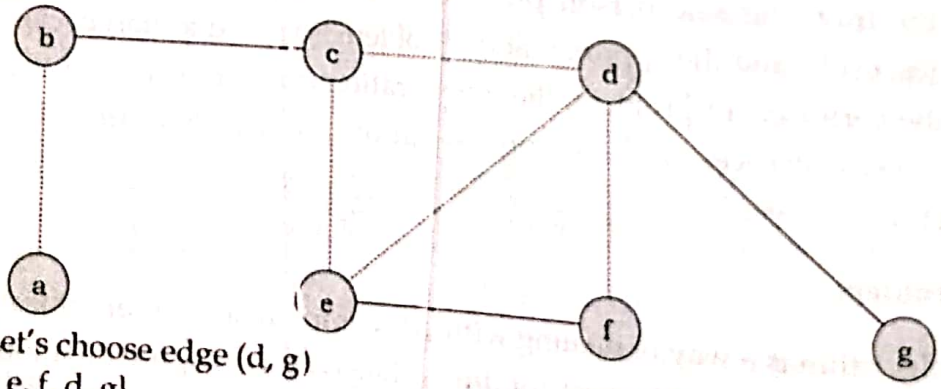


Step 2: Let's choose edge (e, f)

C = {b, c, e, f}

Eliminate edges incident to vertex e and f

E = {(d, g)}

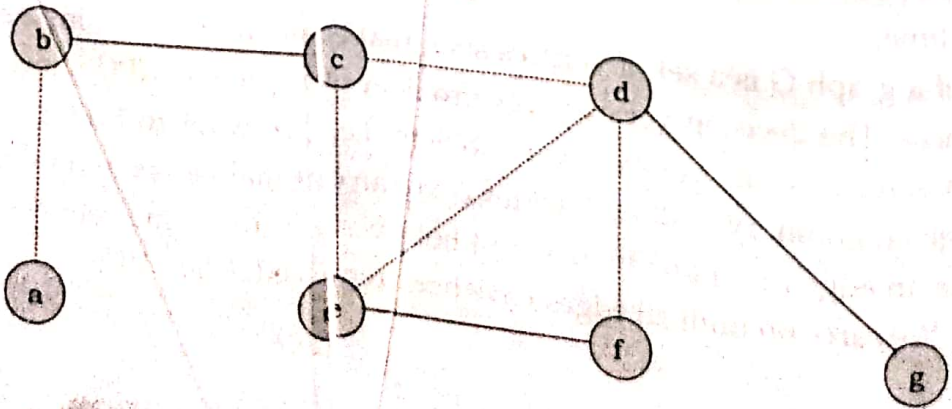


Step 3: Let's choose edge (d, g)

C = {b, c, e, f, d, g}

Eliminate edges incident to vertex d and g

E = {Φ}



Algorithm

```
Approx-Vertex-Cover (G = (V, E))
{
    C = empty-set;
    E'= E;
    While E' is not empty do
    {
        Let (u, v) be any edge in E': (*)
        Add u and v to C;
        Remove from E' all edges incident to u or v;
    }
    Return C;
}
```

Analysis

If E is represented using the adjacency lists the above algorithm takes O(V+E) since each edge is processed only once and every vertex is processed only once throughout the whole operation.

Subset Sum Problem

In the subset sum problem, we are given a finite set $S \subseteq N$ and a target $t \in N$. We ask whether there is a subset $S' \subseteq S$ whose elements sum to t.

We define the problem as a language,

Sub-set sum = {(s, t): there exist a subset $S' \subseteq S$ such that $t = \sum_{S \in S'} S$}

An instance of the Subset sum problem is a pair (S, t), where $S = \{x_1, x_2 \ldots x_n\}$ is a set of positive integers and t is a positive integer. The decision problem asks for a subset of S whose sum is as large as possible, but not larger than t.

DISCUSSION EXERCISE

1. Define the terms "Class P", "Class NP" and "NP-Completeness".
2. Explain cooks theorem.
3. What is approximation algorithm? Explain with suitable example.
4. Define Tractable and Intractable Problems with suitable example.
5. What is a polynomial solvable problem? Explain with suitable example.
6. Differentiate between polynomial and supper-polynomial solvable problems with suitable example.
7. Explain vertex cover problem with suitable example.

8. Write down the algorithm for vertex cover problem and analyze it.
9. Explain subset sum problem with suitable example.
10. Explain and proof that SAT is NP-complete.
11. Explain about SAT with suitable example.
12. Write down the applicable examples for class P, class NP and class NP hard problems.
13. Show that circuit satisfiability problem is NP-complete.
14. Write short not about following:
    i. NP-Hard problem
    ii. NP-Complete problems
15. Show that the Hamiltonian path problem is NP-complete.
16. Show that the decision version of the set –covering problem is NP-complete by reduction from the vertex cover problem.
17. Write the short note on approxirnation algorithm.
18. Explain approximation algorithm with an appropriate example.
19. What are the application areas of vertex cover problem? Explain.
20. Explain Polynomial time reduction with example.

❑❑❑