DAA Notes: Complexity Classes

220001081,220001082,220002018

April 5, 2024

Complexity Classes

Complexity classes are categories that classify computational problems based on the amount of computational resources required to solve them. They provide a framework for understanding the inherent difficulty of computational problems and the feasibility of solving them efficiently. The four main types of complexity classes include **P**, **NP**, **NP-Hard**, and **NP-Complete**.

P Class

A problem is in class **P** if there exists an algorithm that solves it in polynomial time, i.e., the running time of the algorithm is bounded by a polynomial function of the size of the input.

Example 1:

An Eulerian circuit is a cycle in a graph that visits every edge exactly once. Given a graph G, the problem is to determine whether such a cycle exists. There exist polynomial-time algorithms, such as Hierholzer's algorithm or Fleury's algorithm, that can find an Eulerian circuit if one exists in the graph. These algorithms have polynomial-time complexity with respect to the size of the input graph, making the problem of finding Eulerian circuits solvable in polynomial time.

There are some problems which don't have any algorithm which solve those problems in polynomial time. Those problems do not belong to **P** class.

Example 2:

The Halting Problem is a classic example of a problem that is not in the complexity class P. It asks whether a given program, when executed on a given input, will eventually halt or run indefinitely. Alan Turing proved that there is no algorithm that can solve the Halting Problem for all possible inputs. This means there is no single algorithm that, given any program and input, can determine whether the program will halt in a finite amount of time. Formally, the Halting Problem can be stated as follows: Given a program P and an input x, determine whether P halts when executed on input x. The Halting Problem is undecidable, meaning there is no algorithm that can solve it for all cases.

Why do we need to know about Classes?

If we know the problem class, we can know that if we can design optimal/ polynomial solution.

If the problem belongs to NP Complete class (Which you will get to know), we can say there does not exist a known solution in polynomial Time Complexity.

1. An approximation method
2. Special case that belongs to P

In that case:

- We can have approximate solutions for the problem where:
  We allow ϵ amount of error and give solution with error less than or equal to ϵ.

- We can design algorithms to solve on specific type of inputs.
  For example: In the case of Hamiltonian path problem; we can solve it for a Bipartite graph.

***Note*1** : P Class problems are also called as Tractable problems.

***Note*2** : We can design a deterministic Turing machine to solve a P class problem.

***Note*3** : A Non deterministic Turing machine is equivalent with deterministic Turing machine with respect to the language accepted. But they are not equivalent with respect to the Time Complexity.

NP Class

- The set of problems that are verifiable in polynomial time by a deterministic Turing machine.
- If a Non deterministic Turing machine can solve a problem in polynomial time then the solution can be verified in polynomial time by deterministic Turing machine and vice versa.
- P class $\subseteq$ NP class

Note : Whether NP class $\subseteq$ P class i.e P=NP is an open question.

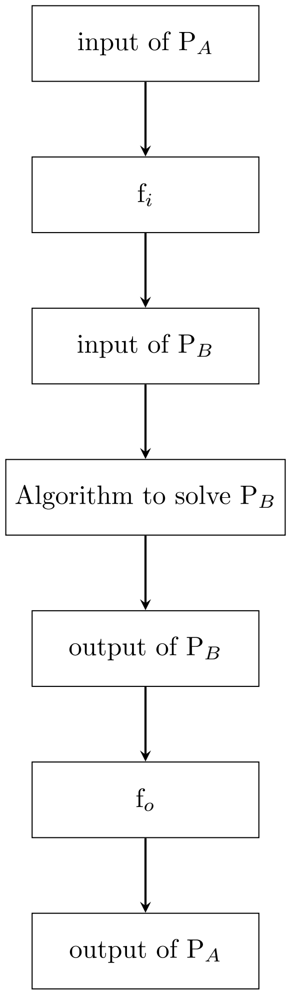
Example Problem

$< G, k >$: Whether there exists a clique of size k in G, where G=(V,E) is a graph and k is an integer.
***Clique* :** Complete induced sub-graph of G=(V,E).
Let us assume a solution for this problem to be $<$ v_1, v_2, $\ldots$, v_k $>$
To verify this solution we just need to check if there exists an edge between v_i and v_j in E, where $1 \le i < j \le k$ i.e the Time Complexity is $O(k^2)$.
That is it can be verifiable in polynomial time

NP Hard Class

Reducibility

$P_A \le_p P_B$ means that the problem P_A is polynomial time reducible to problem P_B.



Here f_i is a function to convert input of P_A to input of P_B and f_o is a function to convert output of P_B to output of P_A

If there exist any functions f_i and f_o with polynomial time complexities then we can say P_A i polynomial time reducible to P_B.

- If $P_B \in$ P class and $P_A \leq_p P_B$, then $P_A \in$ P class
- If $P_A \in$ NP Hard class and $P_A \leq_p P_B$, then $P_B \in$ NP Hard class

Reducibilty is used to prove NP Hard class problems. If an NP Hard problem is reducible to the given problem then we can say that the given problem is also NP Hard.

Example Problem

ILP : Integer Linear Programming:

We are given N variables $x_1, x_2, x_3, \ldots, x_n$ where $x_1, x_2, x_3, \ldots, x_n \in \mathbf{Z}$

We are also given k constraints $f_1(x_1, x_2, x_3, \ldots, x_n) \leq a_1$, $f_2(x_1, x_2, x_3, \ldots, x_n) \leq a_2$, $f_3(x_1, x_2, x_3, \ldots, x_n) \leq a_3$, $\ldots f_k(x_1, x_2, x_3, \ldots, x_n) \leq a_k$

We need to maximize or minimize another function $f(x_1, x_2, x_3, \ldots, x_n)$.

NP Complete Class

If a problem is simultaneously NP and NP Hard, then it belongs to NP Complete Class.

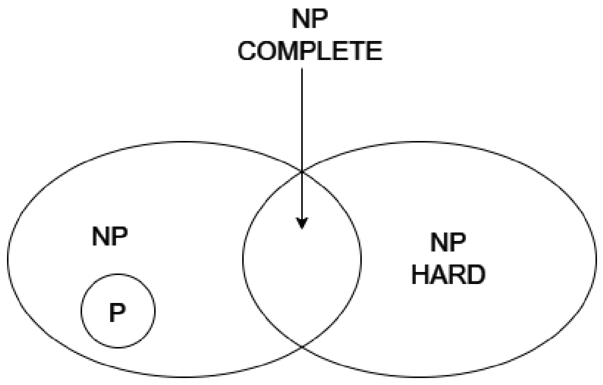
Boolean Circuit Satisfiability Problem

We can demonstrate that an unknown problem is NP-Hard if there exists an NP Hard which can be reduced to given problem. However, to effectively reduce problems, we require a base case. This base case is the binary circuit satisfiability problem.

What does Satisfiability mean? Satisfiability refers to the condition in which a Boolean function, denoted as F, can be made true by assigning appropriate values (either true or false) to its variables. In simpler terms, satisfiability asks whether there exists any combination of variable assignments that causes the Boolean function to evaluate to true.
This is proved to be an NP Hard problem without using reducibilty. So we can use this problem to prove other problems NP Hard.

Summary of the different classes

P Class	NP Class	NP Hard Class	NP Complete Class
Solvable in polynomial time	Verifiable in polynomial time	Can be reducible to another NP Hard Class problem	Both NP and NP Hard problem



Computational Problem VS Decision Problem

In theoretical computer science, a computational problem is a problem that may be solved by an algorithm. For example, the problem of factoring "Given a positive integer n, find a nontrivial prime factor of n."

A decision problem is a computational problem where the answer for every instance is either yes or no. An example of a decision problem is primality testing: "Given a positive integer n, determine if n is prime."

A Computational problem(B) can be reduced to a Decision problem(A).