Subject :- ADSAA **Semester :- V**

Introduction to NP Hard and NP Complete Problems

1. Introduction

In computer science, there exist several famous unresolved problems, and P=NP is one of the most studied ones. Until now, the answer to that problem is mainly "no". And, this is accepted by the majority of the academic world. We probably wonder why this problem is still not resolved.

In this tutorial, we explain the details of this academic problem. Moreover, we also show both P and NP problems. Then, we also add definitions of NP Hard and NP Complete. And in the end, hopefully, we would have a better understanding of why P=NP is still an open problem.

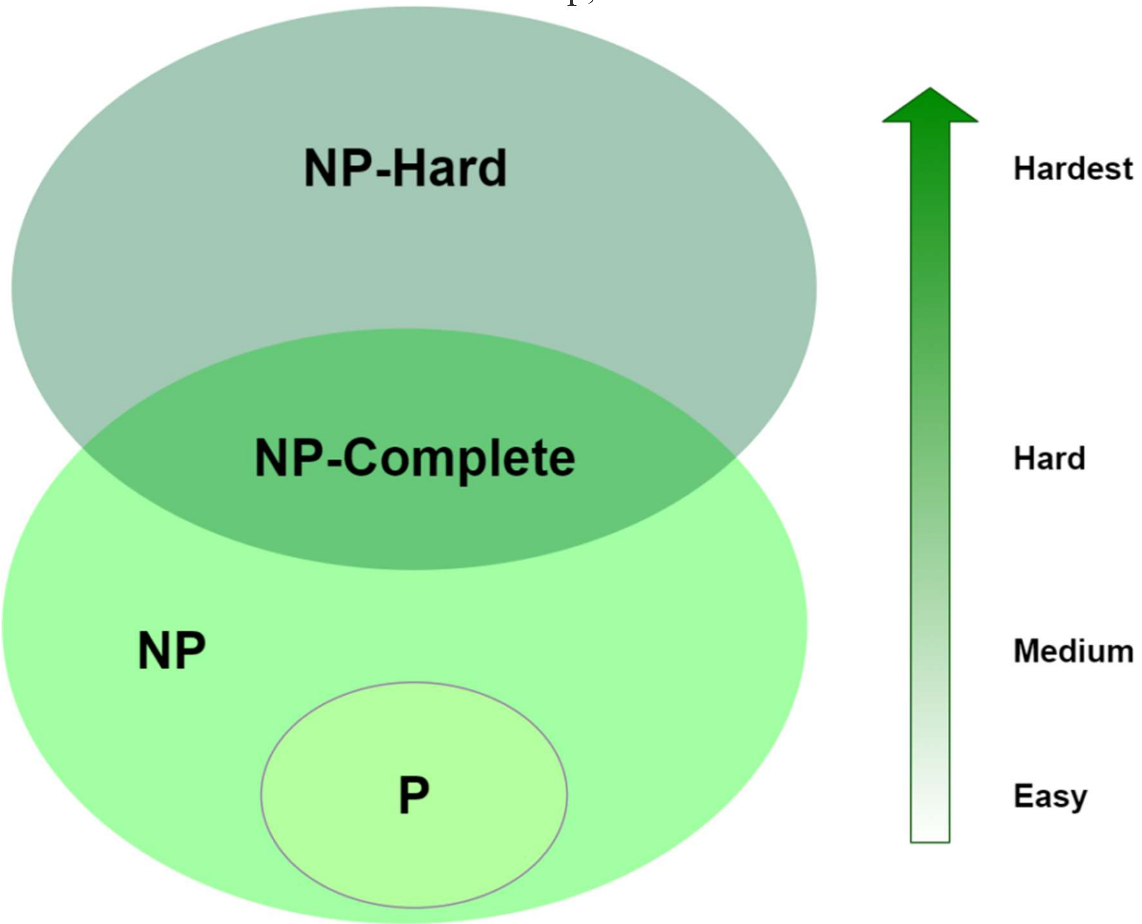
2. Classification

To explain P,NP and others, let's use the same mindset that we use to classify problems in real life. While we could use a wide range of terms to classify problems, in most cases we use an "Easy-to-Hard" scale.

Now, **in theoretical computer science, the classification and complexity of common problem definitions have two major sets;** P which is "Polynomial" time and NP which "Non-deterministic Polynomial" time. There are also NP Hard and NP complete sets, which we use to express more sophisticated problems. In the case of rating from easy to hard, we might label these as "easy", "medium", "hard", and finally "hardest":

- Easy -> P
- Medium -> NP
- Hard -> NP Complete
- Hardest -> NP Hard

And we can visualize their relationship, too:



Using the diagram, we assume that P and NP are not the same set, or, in other words, we assume that P != NP. This is our apparently-true, but yet-unproven assertion. Of course, another interesting aspect of this diagram is that we've got some overlap between NP and NP Hard. We call NP Complete when the problem belongs to both of these sets.

Alright, so, we've mapped P, NP, NP Complete and NP Hard to "easy", "medium", "hard" and "hardest", but how does we place a given algorithm in each category? For that, we'll need to get a bit more formal through the next section.

Through the rest of the article, we generally prefer not to use units like "seconds" or "milliseconds". Instead, we prefer proportional expressions like n, n^2, log2(n) and n^n using Big-O notation. Those mathematical expressions give us a clue about the algorithmic complexity of a problem.

3. Problem Definitions

Let’s quickly review some common Big-O values:

- O(1) – constant-time
- O(log2 (n)) – logarithmic-time
- O(n) – linear-time
- O(n^2) – quadratic-time
- O(n^k) – polynomial-time
- O(k^n) – exponential-time
- O(n !) – factorial-time

where k is a constant and n is the input size. The size of n also depends on the problem definition. For example, using a number set with a size of n, the search problem has an average complexity between linear-time and logarithmic-time depending on the data structure in use.

3.1. Polynomial Algorithms

The first set of problems are polynomial algorithms that we can solve in polynomial time, like logarithmic, linear or quadratic time. If an algorithm is polynomial, we can formally define its time complexity as:

T(n)= O(C * n^k) where C > 0 and k > 0 where C and k are constants and n is input size. **In general, for polynomial-time algorithms k is expected to be less than n**. Many algorithms complete in polynomial time:

- All basic mathematical operations; addition, subtraction, division, multiplication
- Testing for primacy
- Hashtable lookup, string operations, sorting problems
- Shortest Path Algorithms; Djikstra, Bellman-Ford, Floyd-Warshall
- Linear and Binary Search Algorithms for a given set of numbers

As we talked about earlier, all of these have a complexity of O(n^k) for some k, and that fact places them all in P. Of course, we don’t always have just one input, n. But, so long as each input is a polynomial, multiplying them will still be a polynomial. For example, in graphs, we use E for edges and V for vertices, which gives us O(E * V) for Bellman-Ford’s shortest path algorithm. Even if the size of the edge set is E = V^2, the time complexity is still a polynomial, O(V^3), so we're still in P.

We can't always pinpoint the Big-O for an algorithm. Outside of Big-O, we can think about the problem description. Consider, for example, the game of checkers. What is the complexity of determining the optimal move on a given turn? If we constrain the size of the board to 8 X 8, then this is believed to be a polynomial-time problem, placing it in P. But if we say it's an N X N board, it's no longer in P. In this case, how we constrain the search space affects where we place it. Similarly, the Hamiltonian-Path problem has polynomial-time solutions for only some types of input graphs.

Or another example is the stable roommate problem; it's polynomial-time to match without a tie, but not when ties are allowed or when we include roommate preferences like married couples. (These variants are actually NP-Complete, which we'll cover in a moment.) Still another factor to consider is the size of k relative to n. If the input size is going to be near k, then the algorithm is going to behave more like an exponential.

3.2. NP Algorithms

The second set of problems cannot be solved in polynomial time. However, they can be verified (or certified) in polynomial time. We expect these algorithms to have an exponential complexity, which we'll define as:

T(n) = O(C1* k^(C2*n)) where ,C1 > 0 and C2 >0 where C1, C2 and k are constants and n is the input size. T(n) is a function of exponential-time when at least C1=1 and C2=1 . As a result, we get O(k^n). There are several algorithms that fit this description. Among them are:

- Integer Factorization and
- Graph Isomorphism

Both of these have two important characteristics: Their complexity is O(k^n). for some k and their results can be verified in polynomial time. Those two facts place them all in NP, that is, the set of "Non-deterministic Polynomial" algorithms. Now, formally, we also state that these problems must be decision problems – have a yes or no answer – though note that practically speaking, all function problems can be transformed into decision problems. This distinction helps us to nail down what we mean by "verified".

To speak precisely, then, an algorithm is in NP if it can't be solved in polynomial time and the set of solutions to any decision problem can be verified in polynomial time by a "Deterministic Turing Machine". What makes Integer Factorization and Graph Isomorphism interesting is that while we believe they are in NP, there's no proof of whether they are in P and NP Complete. Normally, all NP Complete algorithms are in NP, but they have another property that makes them more complex compared to NP problems.

Let's continue with that difference in the next section.

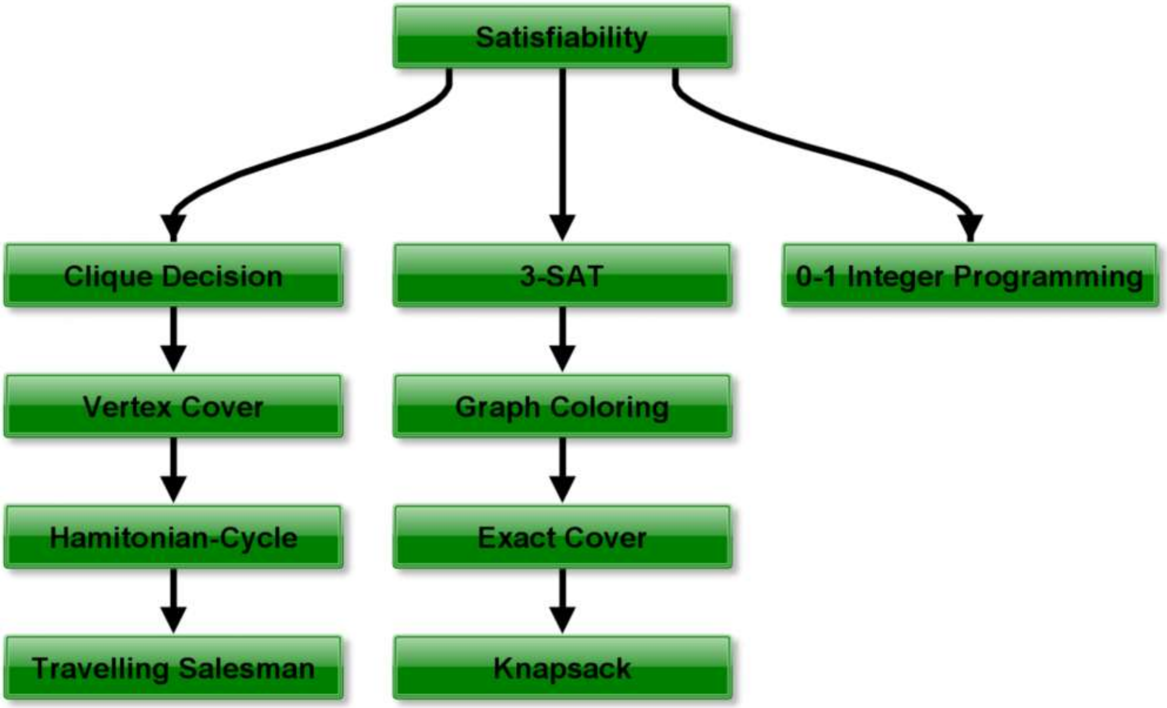
3.3. NP-Complete Algorithms

The next set is very similar to the previous set. Taking a look at the diagram, all of these all belong to NP, but are among the hardest in the set. Right now, there are more than 3000 of these problems, and the theoretical computer science community populates the list quickly. What makes them different from other NP problems is a useful distinction called *completeness*. For any NP problem that's complete, there exists a polynomial-time algorithm that can transform the problem into any other NP-complete problem. This transformation requirement is also called *reduction*.

As stated already, there are numerous NP problems proven to be complete. Among them are:

- Traveling Salesman
- Knapsack, and
- Graph Coloring

Curiously, what they have in common, aside from being in NP, is that each can be reduced into the other in polynomial time. These facts together place them in NP Complete. The major and primary work of NP Completeness belongs to Karp. And his 21 NP Complete problems are fundamental to this theoretical computer science topics. These works are founded on the Cook-Levin theorem and prove that the Satisfiability (SAT) problem is NP Complete:



3.4. NP-Hard Algorithms

Our last set of problems contains the hardest, most complex problems in computer science. They are not only hard to solve but are hard to verify as well. In fact, some of these problems aren't even decidable. Among the hardest computer science problems are:

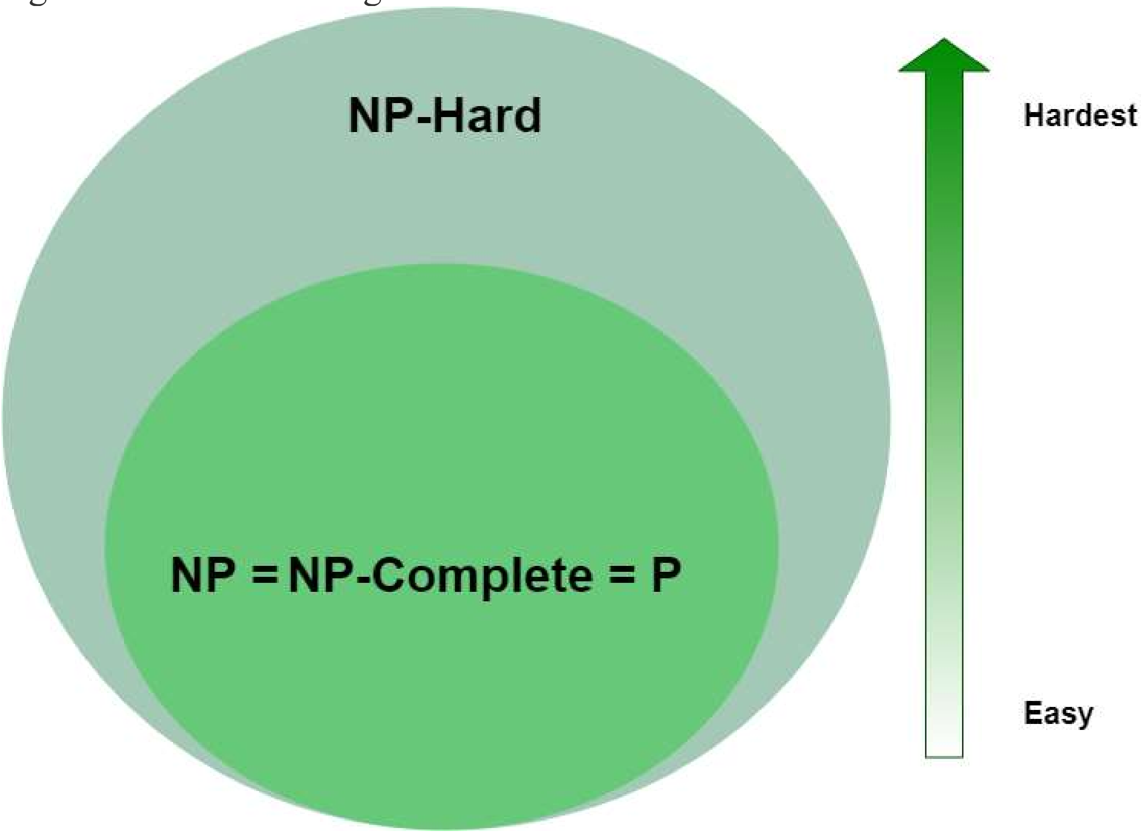
- K-means Clustering
- Traveling Salesman Problem, and
- Graph Coloring

These algorithms have a property similar to ones in NP-Complete – they can all be reduced to any problem in NP. Because of that, these are in NP-Hard and are at least as hard as any other problem in NP. A problem can be both in NP and NP Hard, which is another aspect of being NP Complete .

This characteristic has led to a debate about whether or not Traveling Salesman is indeed NP Complete. Since NP and NP-Complete problems can be verified in polynomial time, proving that an algorithm cannot be verified in polynomial time is also sufficient for placing the algorithm in NP Hard.

4. So, Does *P=NP*?

A question that's fascinated many computer scientists is whether or not all algorithms in belong to :



It's an interesting problem because it would mean, for one, that any NP or NP Complete problem can be solved in polynomial time. So far, proving that P != NP as proven elusive. Because of the intrigue of this problem, it's one of the Millennium Prize Problems.

For our definitions, we assumed that P != NP, however, P=NP may be possible. If it were so, aside from NP or NP Complete problems being solvable in polynomial time, certain algorithms in NP Hard would also dramatically simplify. For example, if their verifier is NP or NP Complete, then it follows that they must also be solvable in polynomial time, moving them into P= NP = NP Complete as well.

We can conclude that P = NP means a radical change in computer science and even in the real-world scenarios. Currently, some security algorithms have the basis of being a requirement of too long calculation time. Many encryption schemes and algorithms in cryptography are based on the number factorization which the best-known algorithm with exponential complexity. If we find a polynomial-time algorithm, these algorithms become vulnerable to attacks.

5. Conclusion

Within this article, we have an introduction to a famous problem in computer science. Through the article, we focused on the different problem sets; *P, NP, NP Complete* and NP Hard. We also provided a good starting point for future studies and what-if scenarios when P =NP. Briefly after reading, we can conclude a generalized classification as follows:

- P problems are quick to solve
- NP problems are quick to verify but slow to solve
- NP Complete problems are also quick to verify, slow to solve and can be reduced to any other NP Complete problem
- NP Hard problems are slow to verify, slow to solve and can be reduced to any other NP problem

As a final note, if P = NP has proof in the future, humankind has to construct a new way of security aspects of the computer era. When this happens, there has to be another complexity level to identify new hardness levels than we have currently.