CSPC42 - Design and Analysis of Algorithms

ALGOS ASSIGNMENT - 2

Name: Rajneesh Pandey

Roll Number: 10611910o

Branch: CSE(B)

NP, NP-Complete and NP-Hard Problems

We have been writing about efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.

Difference between NP-Hard and NP-Complete:

NP-hard	NP-Complete
NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) that can be reducible into X in polynomial time.	NP-Complete problems can be solved by a annon-deterministic Algorithm/Turing Machine in polynomial time.
To solve this problem, do not have to be in NP .	To solve this problem, it must be both NP and NP-hard problems.
Do not have to be a Decision problem.	It is exclusively a Decision problem.
Example: Halting problem, Vertex cover problem, Circuit-satisfiability problem, etc.	Example: Determine whether a graph has a Hamiltonian cycle, Determine whether a Boolean formula is satisfiable or not, etc.

In computer science, there exist several famous unresolved problems, and $\mathcal{P} = \mathcal{NP}$ 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 $\mathcal{P}$ and $\mathcal{NP}$ problems. Then, we also add definitions of $\mathcal{NP}$-$Complete$ and $\mathcal{NP}$-$Hard$. And in the end, hopefully, we would have a better understanding of why $\mathcal{P} = \mathcal{NP}$ is still an open problem.

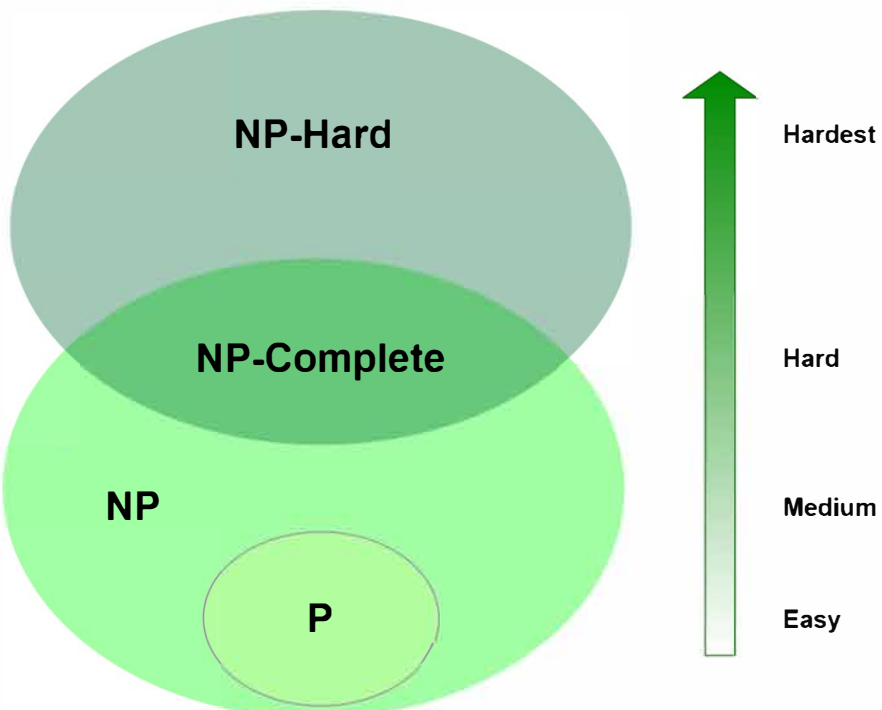
2. Classification

To explain $\mathcal{P}$, $\mathcal{NP}$, 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;** $\mathcal{P}$ which is "Polynomial" time and $\mathcal{NP}$ which "Non-deterministic Polynomial" time. There are also $\mathcal{NP}$-$Hard$ and $\mathcal{NP}$-$Complete$ 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 $\rightarrow \mathcal{P}$
- Medium $\rightarrow \mathcal{NP}$
- Hard $\rightarrow \mathcal{NP}$-$Complete$
- Hardest $\rightarrow \mathcal{NP}$-$Hard$

And we can visualize their relationship, too:



Using the diagram, we assume that $\mathcal{P}$ and $\mathcal{NP}$ are not the same set, or, in other words, we assume that $\mathcal{P} \neq \mathcal{NP}$. 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 $\mathcal{NP}$ and $\mathcal{NP}$-$Hard$. We call $\mathcal{NP}$-$Complete$ when the problem belongs to both of these sets.

Alright, so, we've mapped $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}$-$Complete$ and $\mathcal{NP}$-$Hard$ 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

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, $log_2(n)$, 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
- $O(log_2(n))$ – 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 $\mathcal{P}$. 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 $\mathcal{P}$.

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 8×8, then this is believed to be a polynomial-time problem, placing it in $\mathcal{P}$. But if we say it's an $N \times N$ board, it's no longer in $\mathcal{P}$. 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 $\mathcal{NP}$-$Complete$, 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: $T(n) = O(C_1 * k^{C_2 * n})$ where $C_1 > 0$, $C_2 > 0$ and $k > 0$ where C_1, C_2 and k are constants and n is the input size. $T(n)$ is a function of exponential-time when at least $C_1 = 1$ and $C_2 = 1$. As a result, we get $O(k^n)$. For example, we'll see complexities like $O(n^n)$, $O(2^n)$, $O(2^{0.000001 * n})$ in this set of problems. 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 $\mathcal{NP}$, 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 $\mathcal{NP}$ 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 $\mathcal{NP}$, there's no proof of whether they are in $\mathcal{P}$ and $\mathcal{NP}$-$Complete$. Normally, all $\mathcal{NP}$-$Complete$ algorithms are in $\mathcal{NP}$, but they have another property that makes them more complex compared to $\mathcal{NP}$ problems.

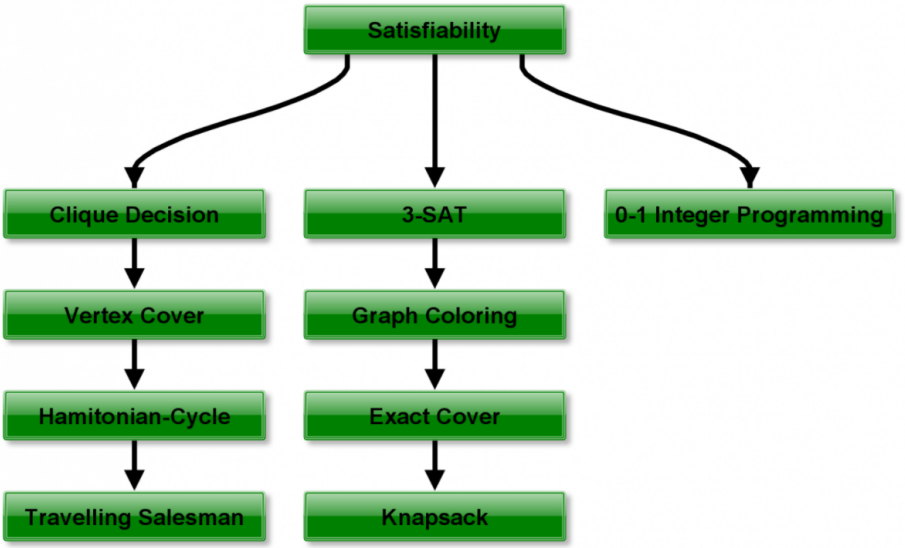
Let's continue with that difference in the next section.

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 $\mathcal{NP}$, 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 $\mathcal{NP}$ problems is a useful distinction called *completeness*. For any $\mathcal{NP}$ problem that's complete, there exists a polynomial-time algorithm that can transform the problem into any other $\mathcal{NP}$-complete problem. This transformation requirement is also called *reduction*.

- **Traveling Salesman**
- **Knapsack**, and
- **Graph Coloring**

Curiously, what they have in common, aside from being in $\mathcal{NP}$, is that each can be reduced into the other in polynomial time. These facts together place them in $\mathcal{NP}\text{-}Complete$. The major and primary work of $\mathcal{NP}\text{-}Completeness$ belongs to Karp. And his 21 $\mathcal{NP}\text{-}Complete$ 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 $\mathcal{NP}\text{-}Complete$:



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:

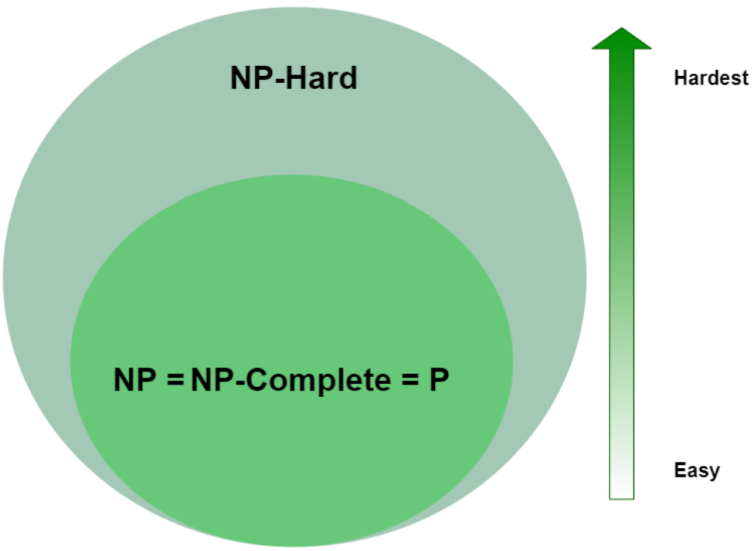
- **K-means** Clustering
- **Traveling Salesman Problem**, and
- **Graph Coloring**

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

This characteristic has led to a debate about whether or not Traveling Salesman is indeed $\mathcal{NP}\text{-}Complete$. Since $\mathcal{NP}$ and $\mathcal{NP}\text{-}Complete$ 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 $\mathcal{NP}\text{-}Hard$.

4. So, Does *P=NP*?

A question that's fascinated many computer scientists is whether or not all algorithms in $\mathcal{NP}$ belong to $\mathcal{P}$:



It's an interesting problem because it would mean, for one, that any $\mathcal{NP}$ or $\mathcal{NP}\text{-}Complete$ problem can be solved in polynomial time. So far, proving that $\mathcal{P}! = \mathcal{NP}$ as proven elusive. Because of the intrigue of this problem, it's one of the Millennium Prize Problems for which there is a $1,000,000 prize.

For our definitions, we assumed that $\mathcal{P}! = \mathcal{NP}$, however, $\mathcal{P} = \mathcal{NP}$ may be possible. If it were so, aside from $\mathcal{NP}$

solvable in polynomial time, moving them into $\mathcal{P} = \mathcal{NP} = \mathcal{NP}\text{-}Complete$ as well.

We can conclude that $\mathcal{P} = \mathcal{NP}$ 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; $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NP}\text{-}Complete$, and $\mathcal{NP}\text{-}Hard$. We also provided a good starting point for future studies and what-if scenarios when $\mathcal{P} = \mathcal{NP}$. Briefly after reading, we can conclude a generalized classification as follows:

- $\mathcal{P}$ problems are quick to solve
- $\mathcal{NP}$ problems are quick to verify but slow to solve
- $\mathcal{NP}\text{-}Complete$ problems are also quick to verify, slow to solve and can be reduced to any other $\mathcal{NP}\text{-}Complete$ problem
- $\mathcal{NP}\text{-}Hard$ problems are slow to verify, slow to solve and can be reduced to any other $\mathcal{NP}$ problem

As a final note, if $\mathcal{P} = \mathcal{NP}$ 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.