Computational Complexity Theory(P,NP,NP-Complete and NP-Hard Problems)

Manoj Pokharel
Tribhuvan University
2 PUBLICATIONS **0** CITATIONS

Computational Complexity Theory

Manoj Pokharel

Central Department of Computer Science and IT, Tribhuvan University

June 19, 2020

Abstract

Given a problem, two situations exist, either we can solve the problem or there exists no solution at all. If any problem can be solved the question rises about its complexity. Any problem in computer science is classified mainly in terms of computability and complexity (aka. Computability theory and Complexity theory). This paper focuses on outlining the complexity of the computable problems in computer science and discusses the elusive question of computer science “Is P = NP?”

1.Background

In computer science there exist several problems and these problems can be classified according to their hardness on various categories. These problems can be classified into four complexity classes as: P, NP, NP-Complete and NP-Hard.

In the field of computational complexity, the unsolved and most studied problem is ’Is P = NP?’. Till now the answer to the problem is mainly ‘no’ and is accepted by the majority of the academic world.

All the terms P, NP, NP-Complete and NP-Hard deal with the Big-O notation for algorithms. Big-O is a measure of how quickly an algorithm runs or solves a problem i.e. the upper bound relative to the input. (Yun, 2019). The several complexity classes are categorized based on the hardness (extent of computational resources), of the problem. Some common Big-O values are:

- 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.

2. Classification

To classify a problem into any one of the classes the problem must be computable, there must exist some algorithm that can solve the problem. Computable problems are often referred as “solvable”, “decidable” and “recursive” problems. In other words, a problem is said to be computable iff there exists a Turing machine that computes an answer for the problem. Non-computable problems are those for which there exists no algorithm to solve it. (Sharma, 2020)

All the computable problems can be placed in any one of the classes according to their hardness. Let’s use the real-life approach for classification using “Easy-to-Hard scale” as:

- Easy → P
- Medium → NP

- Hard → NP-Complete
- Hardest → NP Hard

P (Polynomial) problems:

P problems refer to the problems where an algorithm would take a polynomial amount of time to solve the problem. P class problems can be solved in polynomial time by deterministic Turing machine. The Big-O notation of these problems is always a polynomial (i.e. O(1), O(n), $O(n^2)$). In general, the Big-O complexity can be given as $T(n) = O(C * n^k)$ for all the P problems. where,

$C > 0$ and $k>0$ also C and k are the constant and n is input size.

The algorithms that complete in polynomial time are:

- All the basic mathematical operations: addition, subtraction, division, multiplication.
- Testing for primacy
- Hash table lookup, string operations, sorting problems
- Shortest path problems: Dijkstra, Floyd-Warshall etc.

Being the class P problems, all of these have the complexity of $O(n^k)$ for some k. However, we don't always have just one input *n*, but as long as each input is a polynomial, multiplying them will still be a polynomial.

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 P. But if we say it's an N *N board, it's no longer in P. In this case, how we constrain the search space affects where we place it (Baeldung, 2020).

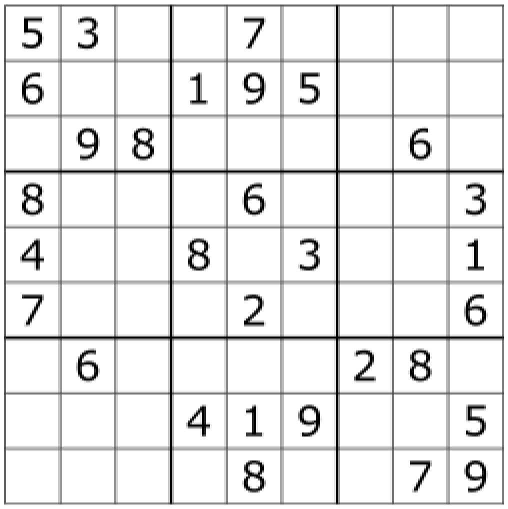
NP (Non-deterministic Polynomial) Problems

These class of problems cannot be solved in the polynomial time. However, they can be verified in polynomial time. In other words, we can state NP problems as problems that satisfy following:

- Decision problems where a solution can be verified by a deterministic Turing machine in polynomial time
- Decision problems where a solution can be found by non-deterministic Turing machine.

We expect these algorithms to have an exponential or factorial time complexity, which in general is defined as $T(n) = O(C_1 * K^{c2*n})$ where, $C_1>0$, $C_2>0$ and $k>0$ also 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 $C1=C2=1$. As a result we get $O(k^n)$ and complexities like $O(n^n)$, $O(2^n)$, $O(2^{0.0001 * n})$ in this set of problems.

Consider the example of a Sudoku game,



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

In order to solve this entire puzzle, the algorithm would have to check each 3x3 matrix to see which numbers are missing, then each row, then each column, and then make sure there are no repetitions of any digit from 0–9. This becomes more complex because the number of digits that are missing is inconsistent in each row, column, and matrix (i.e. top-left matrix is missing 4 digits while top-right matrix is missing 8 digits). Solving this problem would not have a polynomial run-time. However, if you were to feed this puzzle with a possible solution, it would be much less complex to check if there are any repetitions in the rows, columns and matrices. This is a simple check which would have a polynomial run-time.

In essence, NP class problems don't have a polynomial run-time to *solve*, but have a polynomial run-time to *verify* solutions (difficult to solve, easy to check a given answer) (Yun, 2019).

Reduction/Reducibility

Given two problems A and B , we say that problem A is reducible to problem B if there exists an algorithm for solving the problem B, and the same algorithm can be used as a subroutine to solve the problem A efficiently. When this is true, solving A cannot be harder than solving B. "Harder" means requiring higher amount of computational resources.

NP-Complete Problems

The NP-Complete class includes all the problems in class NP that satisfy an additional property of completeness. This distinction of completeness states 'for any problem that is NP-Complete, there exists a polynomial time algorithm that can transform (reduce) the problem into any other NP-Complete problem'.

Simply we can say that, NP-Complete problems belong to NP, but are among the hardest in the set. Right now, there are more than 3000 of these problems and increasing day by day (Baeldung, 2020). The NP problems proven to be complete are:

- Travelling Salesman Problem

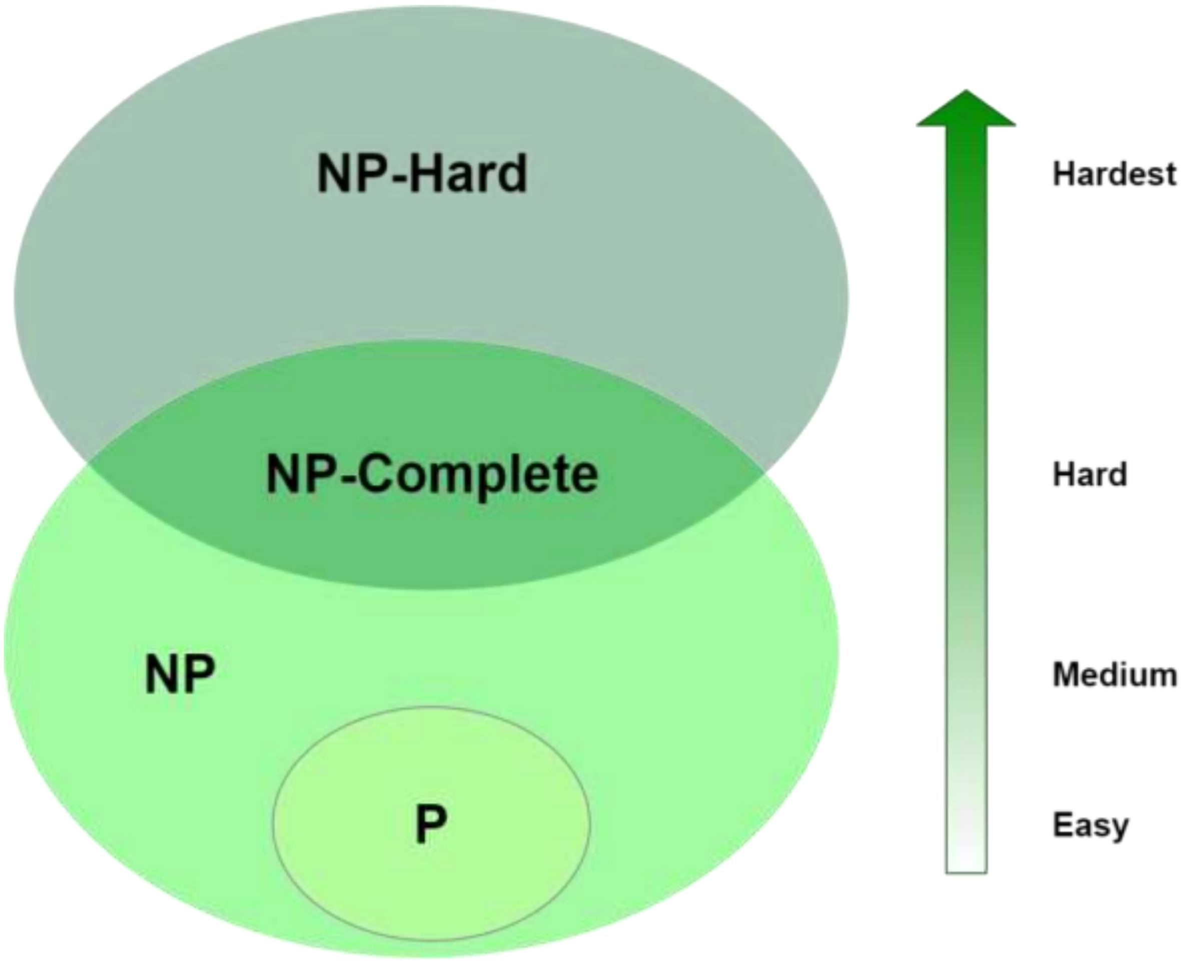
- Knapsack
- Graph Coloring

NP-Hard Problems

The 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. 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.

Simply, A problem is classified as NP-Hard when an algorithm for solving it can be reduced to solve any NP problem. Hence NP-Hard problems are at least as hard as NP problem, but could be much harder or more complex (Yun, 2019).

The graphical representation of the complexity class can be given as follows:

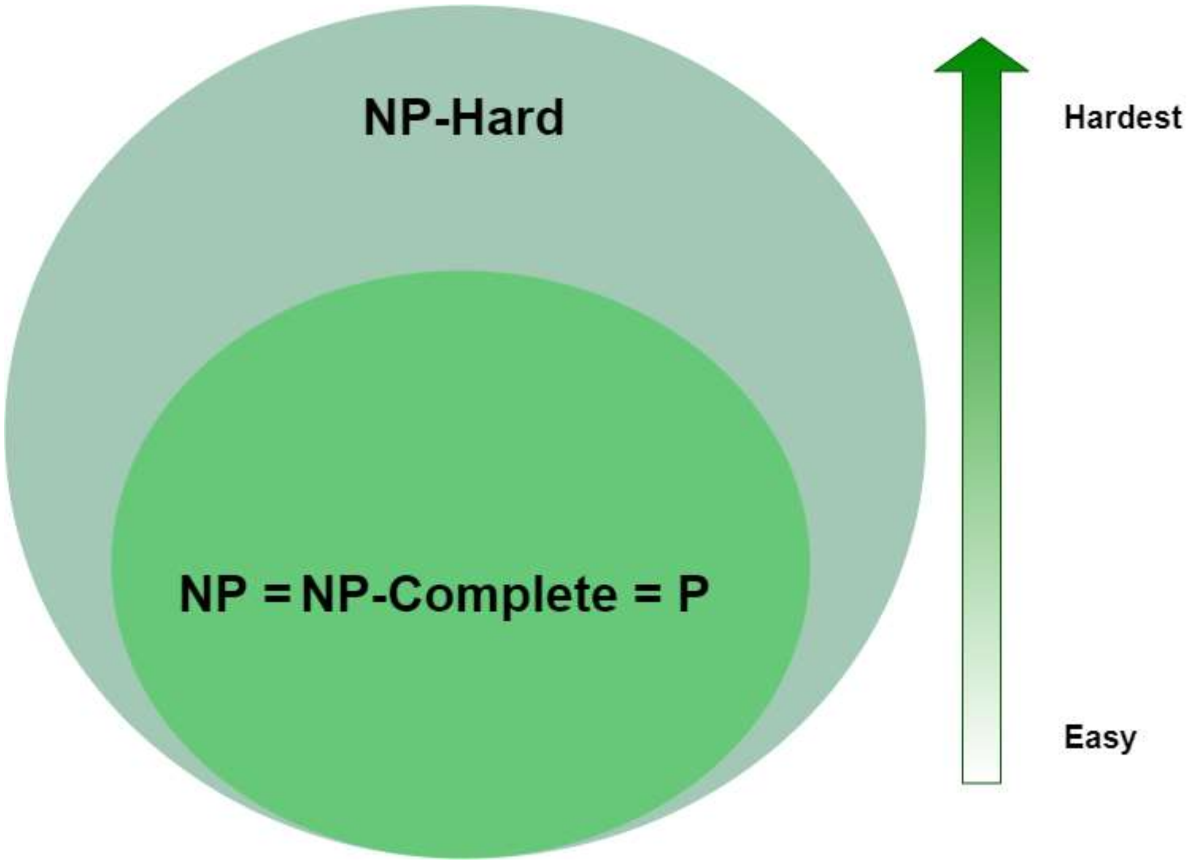


Finally, Does P =NP?

A question that fascinates computer scientists is weather or not all algorithms in NP belong to P. It is an interesting problem because it would mean, that any NP or NP-Complete problem can be solved in polynomial time. So far, proving that P != NP is difficult to find. Because of this nature of the problem it is one of the Millennium Prize Problem for which there is a $1,000,000 prize.

We assume that P != NP, however, P = NP also may be possible. In this case, 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 to the P = NP= NP-Complete class.

We can conclude that P = NP means a radical change in computer science and even in real world scenarios. Currently, some security algorithms, many encryption schemes and algorithms in cryptography are based on the number factorization which is the best-known algorithm with the exponential complexity. If we find a polynomial time algorithm to solve these problems, these algorithms become vulnerable to attacks. This can be graphically illustrated as,



3. Conclusion

We can generalize the findings 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 but slow to solve and can be reduced ro any other NP-Complete problem.
- NP-Hard problems are slow to verify, slow to solve and can be reduced to any other NP problem.

Finally, if P = NP is proved 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.

References

Baeldung. (2020, May 09). *P,NP,NP-Complete and NP-Hard Problems in Computer Science*. Retrieved from Baeldung: https://www.baeldung.com/cs/p-np-np-complete-np-hard

Sharma, P. (2020, June 19). *Computable and Non-Computable problems in TOC*. Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/computable-and-non-computable-problems-in-toc/

Yun, P. (2019, August 27). *P,NP, NP-Hard and NP-Complete Problems*. Retrieved from Medium: https://medium.com/@p.yun1994/p-np-np-hard-and-np-complete-problems-fe679bd1cf9c