Ilkogretim Online - Elementary Education Online, 2021; Vol 20 (Issue 3): pp. 4751-4760
http://ilkogretim-online.org
doi: 10.17051/ilkonline.2021.03.486

Analysis Of Computational Complexity Theory For Theoretical Computer Science Applications

Anil Negi Asst. Professor, Department of Mathematics, Graphic Era Hill University, Dehradun Uttarakhand India.

Abstract

Computational Complexity Theory is a fundamental field in Theoretical Computer Science that analyzes the resources required to solve computational problems. It provides a framework for understanding the efficiency and limitations of algorithms, enabling us to classify problems into different complexity classes based on their inherent difficulty. This paper presents an analysis of Computational Complexity Theory and its applications in Theoretical Computer Science. The class co-NP and the idea of NP-completeness are two additional complexity classes beyond P and NP that are covered in the paper. It examines why it is thought that NP-complete problems are computationally challenging and investigates their significance in defining the limits of intractability. In order to understand the links between different complexity classes, the concepts of reduction and completeness are explained. In a variety of fields, the uses of computational complexity theory are investigated. The relevance of complexity assumptions in cryptography, where secure communication and encryption algorithms are built on them, is discussed in the study. It also emphasises the significance of understanding problem complexity in algorithm design, optimisation, machine learning, and artificial intelligence, where this knowledge helps direct effective solution approaches.

Keywords: Complexity, Turing machine, notation, optimization.

I. Introduction

The exploration of computational complexity theory across several domains is explored in the paper. It focuses on the significance of complexity assumptions in cryptography, where they form the basis of encryption techniques and secure communication. The importance of understanding problem complexity in algorithm design, optimisation, machine learning, and artificial intelligence is also highlighted. The development of effective solution techniques in various domains is greatly influenced by this understanding. The study emphasises the value of computational complexity theory in diverse fields of research and practise by exploring its applications. We can generalise algorithm run outcomes across various issue instances, machines, and implementations thanks to computational models.

Lacking such models, it would be difficult to create a theory based on the intricate details of these actual things' characteristics. Furthermore, even if we were able to develop such a theory, it would probably be of little use in practise because it would need to be adjusted for every different hardware combination. Instead, we are able to describe execution time as a function of problem size, regardless of particular processors or machines, thanks to computational models. Time is assessed in this context by the number of steps needed to solve a certain issue instance, offering a more generalised and abstract viewpoint.

The demand for effective algorithms that can tackle challenging problems across a variety of fields is rising as a result of technology's quick development. However, not all problems can be resolved fast, and it is important to grasp their underlying computational complexity in order to assess if it is possible to come up with effective solutions. A useful approach for problem analysis and classification based on computational needs is computational complexity theory.

This essay's main goal is to examine the core ideas of computational complexity theory and how they apply to practical situations. We will examine the main complexity classes, including P, NP, and co-NP, and talk about how important it is for defining the limits of computing tractability.

The Computational Complexity Theory has numerous and varied applications. We shall look at its application to cryptography, where the creation of safe communication and encryption techniques is supported by complexity assumptions. We will also look at its significance in the design of algorithms, optimisation, machine learning, and artificial intelligence, where a thorough comprehension of problem complexity directs the creation of effective solution approaches.

This paper intends to shed light on the underlying principles that regulate the effectiveness and complexity of computing problems by studying computing Complexity Theory and its real-world applications. Making wise decisions about the feasibility of solving problems and allocating resources is made easier with a better understanding of problem complexity.

II. **Model of Computation**

We define A as the collection of strings that machine M will accept in the context of machine M and its language A. The connection L(M) = A states that the language A is the one that machine M can understand. It is important to remember that even though a computer can take several strings, it can only ever accept one language. This distinction enables us to distinguish between the language that the machine generally understands and the collection of accepted strings.

It is typical to concentrate on strings made up of the binary letters "0," "1," to make the analysis more straightforward. The application of the theory is not considerably constrained by this decision. The fact that we can easily come up with a strategy for converting strings from any finite alphabet into strings over the binary alphabet "0, 1" should not be overlooked. We can explore the theory effectively using this encoding method without any major limits on the kinds of alphabets used.

1. (DTM) Deterministic Turing Machine:

A control unit and a memory unit are the components of a deterministic Turing machine (DTM). While the memory unit is represented as an eternally expanding tape divided into tape squares or cells, the control unit contains a limited number of states. One of a limited number of tape symbols is stored in each tape square. A read/write tape head, which scans one tape square at a time, serves as the conduit for communication between the control unit and the tape. A Turing machine's configuration is an extensive record of all data pertinent to computation, including the current state, symbols on the tape, and the tape head's location.

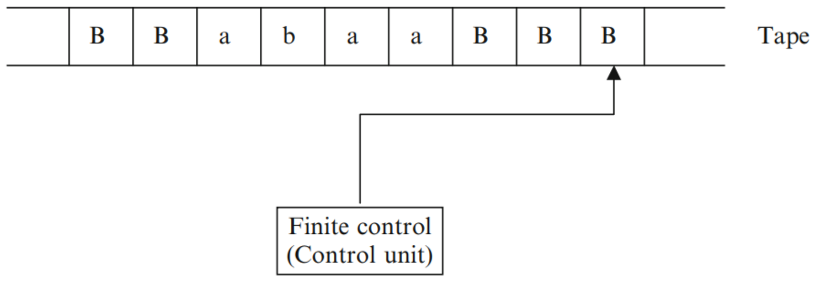

Figure 1: Step wise execution DTM

2. Non-Deterministic Turing Machine (NTM)

Since there is only one move required for each configuration and only one possible next configuration, the Turing machine previously described is deterministic. The machine is referred to as a nondeterministic Turing machine (NTM) if we permit multiple movements for some configurations, creating more than one possible next configuration. Changing a machine's configuration, or moving from one configuration to another, is the general definition of computing. In the end, a finite amount of computations take us from the machine's initial state to a target state, which symbolises the intended outcome of the current problem.

III. Big-O Notation

It is important to pick a specific computing model and examine the resources needed by that model to solve the problems before discussing the complexity of computational challenges. By comparing examples of various sizes, it is possible to compare two issues' fundamental complexity in relevant ways. We can assess how much of the machine's resources, such as time or space, are required to find answers to the issues by fixing a particular computational model, such as a Turing machine or a computational model based on a certain algorithmic paradigm. With this method, we may examine the effectiveness and complexity of problem-solving within a clear computational framework.

If there are positive numbers c and d such that for all n d, the inequality $f(n) \leq c * g(n)$ holds true, we write $f(n) = O(g(n))$ given the two functions f(n) and g(n) mapping positive integers to positive integers.

F(n)< =cg(n)

cF(n)< =g(n)

We compare the growth rates of functions using notations like big-O, big-omega, and big-theta in the context of function analysis.

When we write $f(n) = O(g(n))$, it means that there are positive integers c and d that exist such that for any n d, f(n) is bounded above by c times g(n) for two functions, f(n) and g(n). To put it another way, g(n) acts as a limit for f(n).

Similarly, if we write $f(n) = g(n)$, it means that positive integers c and d exist such that for any n d, f(n) is constrained below by c times g(n). G(n) serves as a lower bound for f(n) in this situation.

We write $f(n) = (g(n))$ when both $f(n) = O(g(n))$ and $f(n) = (g(n))$ are true. This indicates that when n approaches infinity, f(n) and g(n) grow at the same pace up to a fixed factor.

It's crucial to remember that these notations contrast function growth rates rather than their exact values. As a result, even when $f(n) = (g(n))$, f(n) and g(n) can nevertheless have very different values in reality.

For example:

The order of the polynomial f(x) = 6x4 - 2x3 + 5 is referred to as O(g(x)) or O(x4). According to the concept of order, this indicates that there is a constant c such that, for all values of x higher than 1, the absolute value of f(x), denoted by the symbol |f(x)|, is less than or equal to c times the absolute value of g(x), denoted by the symbol |g(x)|. In other words, for every $x > 1$, $|f(x)| \leq c|g(x)|$ holds true.

Proof:

|6x^4 - 2x^3 + 5| ≤ 6x^4 + 2x^3 + 5 where x = 1

Applying the inequality relation, where x3 x4, results in:

6x^4 + 2x^3 + 5 ≤ 6x^4 + 2x^4 + 5x^4

Making the right side simpler:

6x^4 + 2x^4 + 5x^4 = 13x^4

As a result, we can write the phrase 13x4 instead of j6x4 - 2x3 + 5j.

Finally can write:

$$f(x) \text{is } O\big(g(x)\big) \text{as } x \longrightarrow \infty$$

IV. Reduction Analysis

A reduction is a method for turning one issue into another in such a way that, if the second issue can be resolved, the original issue can also be resolved. Consider the scenario where you must find your way around a strange city. If you had a map of the city, it would be rather easy. The situation serves as an example of reducibility. The difficulty in locating a map of the city can be reduced to the difficulty of navigating the city. Mathematical examples of reducibility can be discovered as well. For instance, it is possible to simplify the challenge of solving a system of linear equations to the challenge of inverting a matrix.

a. Linear Reduction

Linear reductions are crucial to the study of complexity theory. The following is a definition of the term "linear reduction" from Brassard and Bratley:

Let A and B represent two solvable issues. If an algorithm for problem B that operates in the time complexity O(t(n)), where t(n) is any arbitrary function, implies the existence of an algorithm for problem A that likewise operates in the time complexity O(t(n)), then A is said to be linearly reducible to B, denoted as A l B. Accordingly, if problem B can be effectively solved, then problem A can be effectively solved as well.

A and B are referred regarded as being linearly equal when both A l B and B l A are valid, represented as A l B. In other words, two problems are deemed to be linearly comparable if the existence of an effective solution for one problem implies the existence of an efficient algorithm for the other problem. In complexity theory, linear reductions are a useful tool for examining the connections between various issues and their computational complexity. They enable us to make connections and gauge problems' relative complexity or simplicity depending on how easily they may be reduced to one another. We can acquire insights into the intrinsic complexity and viability of doing computer jobs effectively by understanding the linear reductions between issues.

b. Polynomial Reduction

Two issues, X and Y, shall exist. If there is an algorithm that can solve issue X in a time complexity that would be polynomial if we ignored the time necessary to solve random instances of problem Y, then problem X is said to be polynomially reducible to problem Y in the sense of Turing, denoted as X T Y. In other words, the algorithm for solving problem X can use a fictitious technique to magically solve problem Y without incurring any additional costs.

In other words, if a good algorithm for solving problem Y already exists, then we can design a good algorithm for solving problem X. In other words, if a good algorithm for solving problem Y already exists, then we can design a good algorithm for solving problem X. Without taking into account the extra time needed to solve instances of issue Y, the polynomial reduction enables us to use the problem-solving abilities of problem Y as a subroutine in solving problem X.

c. Traveling Salesman Optimization Problem:

Operations research and the sciences. In order to return to the starting city after visiting each of a specified set of cities exactly once, the quickest path a salesman can take must be determined.

The TSP will be indicated as follows:

Assume that the undirected graph G = (V, E) contains the set of cities V and the edges E that connect them.

Let d(i, j) represent the travel time or distance between cities i and j.

Finding a Hamiltonian cycle a cycle that stops in every city precisely once is the goal of the TSP in order to reduce the overall distance travelled.

A permutation of the cities, indicated as = (v_1, v_2,..., v_n), can be used to represent a solution to the TSP, where v_i is the i-th city in the permutation. The total distance covered by a given solution (π) is represented by the objective function for the TSP, which we can now define. We'll write this function's name as f(π):

$$f(\pi) = \Sigma[d(v\_i, v\_i + 1)], \text{for } i = 1 \text{ to } n - 1, + d(v\_n, v\_1)$$

Finding the permutation that minimises the objective function f() is the aim of solving the TSP. It is computationally difficult to solve the TSP optimally for big problem cases because the number of permutations increases exponentially with the number of cities.

The TSP is approached and approximate solutions that are near to the optimal path are found using a variety of methods and techniques, including genetic algorithms, ant colony optimisation, greedy heuristics, and dynamic programming. These methods try to balance the effectiveness of computation with the quality of the solutions.

d. Problems in NP-complete

The idea of NP-completeness is significant because it gives a concrete indication of how difficult it is to solve a problem effectively. It is exceedingly improbable to uncover a polynomial-time algorithm for a problem that has been shown to be NP-complete. Theoretically, it is necessary to concentrate on an NP-complete problem in order to show that P is not equal to NP. Showing the difficulty of any NP-complete problem would indicate the existence of problems that cannot be solved effectively until P = NP because it is thought that NP-complete problems take longer than polynomial time to solve.

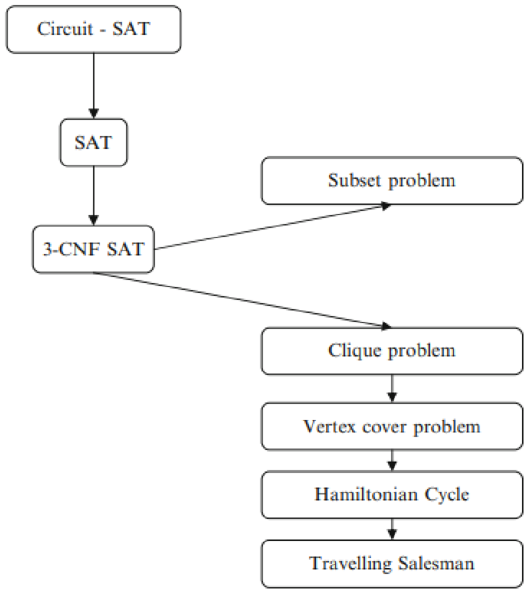

Figure 2: NP-complete problem snapshot

On the other hand, finding a polynomial time technique for any NP-complete issue allows one to demonstrate that P = NP. This would result in a fundamental advancement in the theory of computer complexity by proving that all NP-hard problems can be effectively addressed.

NP-completeness has important ramifications in the real world. Researchers and practitioners can use it as a reference point to avoid wasting time looking for polynomial time solutions to situations that are known to be NP-complete. Although there is no conclusive mathematical solution to the P vs NP dilemma, the notion that P is not equal to NP is widely accepted. As a result, proving a problem to be NP-complete offers compelling proof that it is fundamentally challenging and unlikely to have an effective polynomial-time solution.

Researchers might concentrate their efforts on creating approximation algorithms, heuristics, or specialised strategies to effectively tackle these challenging problems by acknowledging the NP-completeness phenomena. It enables reasonable expectations for computing effectiveness and stimulates investigation of other problem-solving strategies outside the realm of precise algorithms.

V. Conclusion

For theoretical computer science applications, the examination of computational complexity theory is of utmost significance. This field offers a framework for comprehending both the resources needed to solve computational problems effectively and their intrinsic complexity. Researchers and practitioners can choose wisely when inventing algorithms, streamlining procedures, and creating intelligent systems by carefully considering complexity assumptions, problem complexity, and algorithmic efficiency. The division of issues into complexity classes like P, NP, and beyond is one of the major learnings from computational complexity theory. This classification aids in determining a problem's tractability or intractability and directs the creation of efficient algorithms. The discoveries made by computational complexity theory have applications in a variety of fields. Complexity assumptions are the foundation of safe transmission and encryption techniques in cryptography. Complexity analysis directs the creation of effective algorithms for tackling problems in algorithm design. Understanding the complexity of learning and optimisation problems is helpful in selecting the right algorithms and modelling strategies in machine learning and artificial intelligence. The foundation of theoretical computer science is computational complexity theory, which offers vital insights into the intrinsic complexity of issues and the effectiveness of algorithms. It gives academics the ability to evaluate and compare the complexity of different problems, create effective algorithms, and create workable solutions to challenging real-world problems. Computational complexity theory research and applications are crucial for pushing the limits of what is computationally possible and fostering innovation in computer science as technology develops further.

References:

1. L. Adleman and M.-D. Huang. Recognizing primes in random polynomial time. In Proc. 19th ACM Symposium on Theory of Computing, 1987, 462–469.

2. Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. Annals of Mathematics, 160: 781–793 (2004).
3. Dorit Aharonov and Oded Regev. Lattice Problems in NP ∩ coNP. In Proc. 45th IEEE Symposium on Foundations of Computer Science, 2004, 362–371.
4. Bautista-Valhondo, Joaquín & Pereira, Jordi. (2006). A GRASP algorithm to solve the unicost set covering problem. Computers and Operations Research. 34.
5. Ignatiev, Alexey & Previti, Alessandro & Marques-Silva, Joao. (2015). SAT-Based Formula Simplification. 10.1007/978-3-319-24318-4_21.
6. Mohammadi, Aboozar & Abadi, I.. (2012). Heuristic algorithm for solving the integer programming of the lottery problem. Scientia Iranica. 19. 895–901. 10.1016/j.scient.2012.04.015.
7. Le Berre, Matthieu & Rebai, M. & Hnaien, Faicel & Snoussi, Hichem. (2015). A Specific Heuristic Dedicated to a Coverage/Tracking Bi-objective Problem for Wireless Sensor Deployment. Wireless Personal Communications. 10.1007/s11277-015-2548-2.
8. Haddadi, Salim & Cheraitia, Meryem & Salhi, Abdel. (2018). A Two-Phase Heuristic for Set Covering. International Journal of Mathematics in Operational Research. 13, No. 1. 61-78. 10.1504/IJMOR.2018.10013170.
9. Yin, Minghao. (2015). A novel local search for unicost set covering problem using hyperedge configuration checking and weight diversity. Science in China Series F Information Sciences. 60. 10.1007/s11432-015-5377-8.
10. Kim, Ji-Su & Lee, Dong-Ho. (2013). A restricted dynamic model for refuse collection network design in reverse logistics. Computers and Industrial Engineering. 66. 1131-1137. 10.1016/j.cie.2013.08.001.
11. Sondi, Patrick & Gantsou, Dhavy & Lecomte, Sylvain. (2013). Design guidelines for quality of service support in Optimized Link State Routing-based mobile ad hoc networks. Ad Hoc Networks. 11. 298-323. 10.1016/j.adhoc.2012.06.001.
12. Lin, Tzu-Hua & Woungang, Isaac. (2010). An Enhanced MPR-Based Solution for Flooding of Broadcast Messages in OLSR Wireless ad hoc Networks. Mobile Information Systems. 6. 249-257. 10.1155/2010/820453.
13. Clausen, Thomas Heide & Hansen, Gitte & Christensen, Lars & Behrmann, Gerd. (2002). The optimized link state routing protocol, evaluation through experiments and simulation.
14. Hariprasad S A. (2020). A Conjectural based Framework to Detect & defend/Classify Selfish Nodes and Malicious Nodes in Manets Using AODV. 15. 08-15. 10.21172/ijiet.151.03.
15. Jerlin, Asha & Deverajan, Ganesh & Patan, Rizwan & Gandomi, Amir. (2020). Optimization of Routing-Based Clustering Approaches in Wireless Sensor Network: Review and Open Research Issues. Electronics. 9. 1630. 10.3390/electronics9101630.

16. Lalitha, K. & D, Rajesh Kumar & Gopal, Dhananjay & Gadekallu, Thippa & Aboudaif, Mohamed & Abouel Nasr, Emad. (2020). A Heuristic Angular Clustering Framework for Secured Statistical Data Aggregation in Sensor Networks. Sensors. 20. 4937. 10.3390/s20174937.
17. Konyeha, Susan & John-Otumu, Adetokunbo. (2020). An Improved Token-Based Umpiring Technique for Detecting and Eliminating Selfish Nodes in Mobile Ad-hoc Networks. Journal of Computer Science and Technology. 44. 74-85.