NP-Completeness

Nishtha Jindal

January 30, 2024

Abstract

This paper provides an introductory overview of complexity computational theory. The paper provides detailed explanations of polynomial and non-deterministic polynomial time complexities before investigating the class of NP-Complete problems. The paper finishes with a discussion of the applications and consequences of NP-Completeness in modern technology.

1 Introduction

Computational theory plays an unknowingly large role in modern day society. By understanding the fundamental principles of analyzing computational algorithms, algorithms can be improved along with the computer systems performing such computations. Advancements in theory, such as improved integer factorization algorithms, are ongoing and have the potential to impact all computing technology.

Given the significant role computational theory plays in modern technology, this paper provides an introductory overview of time complexity and delves into the topic of NP-Completeness. NP-Completeness is a study of algorithms that bridge the gap between problems that can be solved in polynomial time and related solutions that can be verified in polynomial time.

To analyze the topic of NP-Completeness, this paper will first explore time complexity before providing technical definitions of polynomial time vs. non-deterministic polynomial time. Then, the paper will provide an analysis of NP-Complete algorithms and intractability before performing a reduction within the class of NP-Complete problems. The paper will close with a discussion of the applications of NP-Completeness in modern day computing.

2 Time Complexity

Time complexity measures the time it takes to solve a given problem. An important aspect of assessing the feasibility and suitability of an algorithm is analyzing its time complexity. In practice, if a problem has too large of a time complexity, it is deemed unsolvable. Time complexities are measured via the use of Turing Machines and mathematical analysis.

2.1 Turing Machines

Turing machines, proposed by Alan Turing, are simple abstract devices that are used to evaluate the limitations of computation. The Church-Turing Thesis proposes that there is a Turing Machine for every solvable computational algorithm.

Turing machines are composed of three distinct components, including a tape, a tape head, and the finite control. The tape is infinitely long and is divided into a sequence of cells, containing either a

1, 0, or an empty space. The tape head can read the contents of a cell and move horizontally between the cells. Lastly, the last component is the finite control, which issues the commands that operate the machine. These commands include telling the tape head where to go and performing calculations based on the current state. The model of a simple Turing machine can be referenced in Figure 1 below.

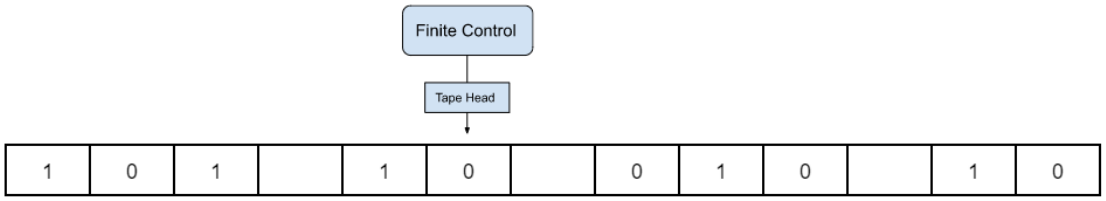

Figure 1: Simplified Turing Machine Model

The Turing machine tape is configured with the problem via the placement of 1's, 0's, and empty spaces in the tape. Once the current state and tape head location are set, the machine executes by moving the tape head and updating the contents of the tape based on the state. Once the machine halts, the contents of the tape contain the solution to the algorithm.

Definition 2.1.1. The time complexity of an algorithm is given by f(n), the number of steps performed by a corresponding Turing machine for input size n.

This is a simplified representation of Turing machines and their operation. The study of Turing machines is complex and has many implications in computational theory. The simplified explanation provided above denotes the overarching use of Turing machines.

2.2 Big-O Notation

Oftentimes in computational theory, time complexities are represented via big-O notation rather than a function. Big-O notation only considers the highest order term of the function and ignores all related coefficients. Due to the nuances of computation, using big-O notation greatly simplifies related calculations.

There are two distinct classifications for the running time of algorithms. The first is algorithms that run in polynomial time, and the second is algorithms that take exponential time.

Definition 2.2.1. An algorithm has polynomial time complexity when it takes the form O($n^k log(n)^k$), where real number k $\geq$ 0.

The following examples showcase polynomial time complexities and how the highest order term is identified.

Example 2.2.1. Suppose the time complexity of an algorithm is represented by f(n), where $f(n) = 62n^6 + n^2log(n) + 122n^5 + 42n^3 + n + 6$. Then, the algorithm has polynomial time complexity of O(n^6).

Example 2.2.2. Suppose the time complexity of an algorithm is represented by h(n), where $h(n) = n^2log(n) + 20n^2 + +nlog(n) + 23234n$. Then, the algorithm has polynomial time complexity of O($n^2log(n)$).

Algorithms with polynomial time complexities are considered to be computable for large input sizes of n. In contrast, exponential algorithms are considered unsolvable.

Definition 2.2.2. An algorithm has exponential time complexity when it takes the form $O(n^k log(n)^k c^n)$, where real number k $\geq$ 0 and real number c $\geq$ 2.

The following example showcases a function that has exponential time complexity.

Example 2.2.3. Suppose the time complexity of an algorithm is represented by g(n), where $g(n) = 2^n + 17n^40 + 2n^33 + 64n^2.7 + 4n + 1$ Then, the algorithm has complexity $O(2^n)$.

Algorithms with exponential time complexities are theoretically solvable but are unsolvable in practice. While the algorithm may appear solvable for small input sizes of n, as n increases the run time increases exponentially. For instance, when n = 10000, 2^n is approximately 10^3000; this value exceeds the total number of atoms in the universe. For large data sets, where input sizes greatly exceed 10000, algorithms with exponential time complexities are unusable.

Figure 2 provides a visual representation of how different complexities compare. The graph also helps visualize how computation time increases depending on increasing input sizes.

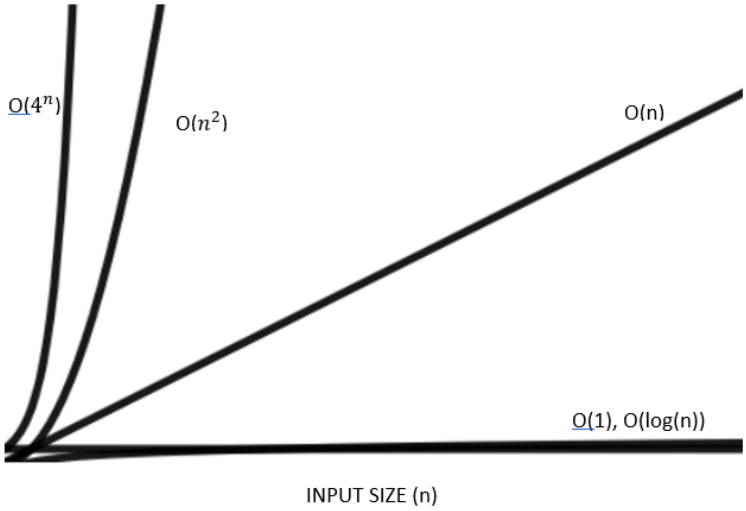

Figure 2: Time Complexity Graph

3 Polynomial Time

Problems solved in polynomial time are said to be tractable and make up a subclass of all problems.

Definition 3.0.1. The class of P consists of problems that can be solved in polynomial time.

For instance, finding the maximum element in an array of size n is a problem that can be solved in polynomial time.

Theorem 3.0.1. MAX-ELEM $\in$ P

Proof. To solve this problem, we can compare every element in the array to the last known maximum.
Step 1: Set maximum variable equal to first element in array.
Step 2: Set variable i equal to 1.
Step 3: If i equals n, skip to step 7.
Step 4: If element i of the array is greater than the value stored in the maximum variable, set the maximum variable equal to element i.

Step 5: Increment i.
Step 6: Jump to step 3.
Step 7: The value stored in the maximum variable is the maximum.

This algorithm completes a fixed number of steps for each element of the array, meaning the algorithm completes in $f(n) = cn$, where c is some coefficient and n is the number of elements in the array. Therefore, the run time of this algorithm in big-O notation is $O(n)$, meaning the algorithm completes in polynomial time. □

While identifying new prime numbers is famously computationally difficult, identifying if two integers x and y are coprime can be computed in polynomial time. In other words, an algorithm with polynomial running time can identify if gcd(x, y) = 1.

Theorem 3.0.2. COPRIME ∈ P

Proof. A potential algorithm to solve this problem applies the Euclidean algorithm to integers x and y, where x ¿ y.

Step 1: If y = 0, skip to step 6.
Step 2: Store x modulo y in a temp variable.
Step 3: Set x to equal the value of y.
Step 4: Set y to equal the value of the temp variable.
Step 5: Jump to step 1.
Step 6: If x = 1, x and y are coprime. Otherwise, x and y are not coprime.

Let's evaluate the algorithm using integers x = 2323 and y = 1102. 2323 modulo 1102 is 119, so we set the temp variable to equal 119. x equals 1102, and y equals 119. 1102 modulo 119 is 31, so we set the temp variable to equal 31. x now equals 119, and y equals 31. 119 modulo 31 is 26, so we set the temp variable to equal 26. x equals 31, and y equals 26. 31 modulo 26 is 5, so we set the temp variable to equal 5. x now equals 26, and y equals 5. 26 modulo 5 equals 1, so we set the temp variable to equal 5. x equals 5, and y equals 1. 5 modulo 1 is 0, so we set the temp variable to equal 0. x equals 1, and y equals 0. Since y = 0, we can now check the value of x. Since x = 1, we know that x and y are coprime. Thus, gcd(2323, 1102) = 1.

The time complexity of this algorithm is determined by the number of steps the algorithm must complete until y equals 0. Based on this algorithm, the value of y is x mod y from the previous iteration. This can be written as $y_{i+2} = x_{i+1} \bmod y_{i+1}$. However, we can show that this quantity is bounded by 0 and x/2.

Lemma 3.0.1. x mod y < x/2

Proof. There are two distinct cases.

Case 1: y ≤ x/2
x mod y < y by definition of modulo y. However, y ≤ x/2, so x mod y < x/2.

Case 2: y > x/2
x mod y ≡ x - y. However, y > x/2, so x - y < x/2.

□

Additionally, based on the algorithm, the value of x is the value of y from the previous iteration. Thus, $x_{i+1} = y_i$. Given $y_{i+2} = x_{i+1} \bmod y_{i+1}$, we can write

$$y_{i+2} = y_i \bmod y_{i+1}$$

However, we know x mod y < x/2, which means

$$y_{i+2} < y_i/2$$

This then means that the value of y is at least reduced by half every two iterations. This means the algorithm completes in $f(n) = 2log(n)$, where n is the value of y. The coefficient 2 is used to represent that the value is halved for every two steps. Therefore, the time complexity of this algorithm is represented by $O(log(n))$, meaning it runs in polynomial time. □

4 Non-Deterministic Polynomial Time

Based on the current state of computing and assumptions made about P not being equal to NP, not all problems can be solved in polynomial time. These problems form a separate subclass of all problems.

Definition 4.0.1. The class of NP, or non-deterministic polynomial time problems, consists of problems that can be verified in polynomial time.

For this class of problems, if given a potential solution, called a certificate, the validity of the solution can be determined in polynomial time. This further means that P ⊂ NP as all problems that can be solved in polynomial time can be verified in polynomial time.

For instance, the subset sum problem falls within the class of NP problems. This problem involves identifying if there exists a subset of a given set that adds to a specific number T. For example, for set A = 1, 3, -4, 12, 6 and T = -3, there exists a subset A' = 1, -4 such that elements of A' add up to -3.

While it is seemingly easy to identify a solution for small sets, the problem is known to have exponential run time and becomes impractical as n increases. The most common approach to solve this problem is to cycle through all subsets of the set and then add up each subset to check if it equals T. For such a solution, there would be 2^n number of subsets to check and at most n numbers to add together in each subset. Thus, this solution would be in $O(2^n \times n)$, meaning it would have exponential complexity. Computer scientists have worked on creating an optimized algorithm for years, and the fastest solution yet, proposed by Howgrave-Graham and Joux, still runs in exponential time. The proof below shows, however, that this problem can be verified in polynomial time.

Theorem 4.0.1. Subset sum ∈ NP

Proof. Suppose we are given set S, representing the initial set, set S', representing the subset, and value T, representing the number the elements of the subset must add up to.

Step 1: Store the number zero in a sum variable.
Step 2: Store the number one in a flag variable.
Step 3: If set S' is empty, skip to step 8.
Step 4: Remove an element from set S'.
Step 5: If the element is not present in set S, set the flag variable to 0.
Step 6: Add the value of the element to the sum variable. Step 7: Jump to step 3.

Step 8: If the flag variable equals 1 and sum variable equals T, then the solution is valid. Otherwise, the solution is invalid.

This algorithm runs a fixed number of steps for each element of set S', meaning the algorithm completes in $f(n) = cn$, where c is some coefficient. Therefore, the run time of this algorithm in big-O notation is $O(n)$, meaning the algorithm checks the solution in polynomial time. □

Another famous NP problem is the boolean satisfiability problem. This problem determines if there exists values for the literals in a clause that make a statement true. Literals are propositional values or negated propositional values, and a clause is a disjunction of multiple literals. A clausal formula is a conjunction of multiple clauses. For instance, the following is a valid clause for three literals: $x \wedge (\neg y \vee z) \wedge (x \vee \neg z)$. One possible solution for this problem is when x = true, y = false, and z = true.

In order to solve this problem, the most common approach involves checking every combination of true and false for all the propositions. For n propositions, there are 2^n different combinations that need to be checked. This means this algorithm would take $O(2^n)$ time, meaning it would have exponential complexity. This complexity may be reasonable for very small input sizes; however, as the value of n increases, this algorithm becomes unusable. However, as shown below, this problem can be verified in polynomial time.

Theorem 4.0.2. SAT ∈ NP

Proof. Suppose we are given a clause with n literals and true or false values for each proposition. The solution can be checked by plugging in the values provided for each proposition and identifying if the final clause is true or false. If the clause is false, the solution is invalid. If the clause is true, the solution is valid.

This algorithm runs in one singular step, meaning the algorithm completes in $f(n) = c$, where c is some constant. Therefore, the run time of this algorithm in big-O notation is $O(1)$, meaning the algorithm checks the solution in constant time, which makes it part of the class of P problems. □

5 NP-Completeness

Before defining NP-Completeness, one must understand the concept of problems being time reducible.

Definition 5.0.1. A problem X is polynomial time reducible to problem Y when problem X can be transformed into problem Y via an algorithm that completes in polynomial time.

NP-Complete represents a subclass of decision problems, problems with 'yes' or 'no' answers. The NP-Complete problems are considered to be the hardest problems in the NP set.

Definition 5.0.2. A problem X is in the NP-Complete set if and only if it satisfies the following conditions.
1. Problem X ∈ NP
2. Every problem in NP is polynomial time reducible to problem X. This condition identifies if a problem is in the class of NP-Hard problems.

Condition two seems impossible to prove as it requires us to reduce every problem in NP to problem X; however, due to the transitivity of reduction, it suffices to reduce a problem in the NP-Complete

to problem X. If problem X $\in$ NP and a known problem in NP-Complete is time reducible to problem X, problem X is NP-Complete.

The first problem in the NP-Complete class was thus found by Stephen Cook and Leonid Levin. The formal proof, called the Cook-Levin Theorem, involves using a deterministic Turing machine to show that every problem in NP can be reduced to the boolean satisfiability problem introduced in the section above. Thus, if the SAT problem can be reduced to some problem X in polynomial time, problem X is NP-Complete.

The following section contains a reduction of the SAT problem to the 3-SAT problem.

6 Reduction

The 3-SAT problem is a variation of the SAT problem introduced in the Non-Deterministic Polynomial Time section above. The 3-SAT problem requires that each clause must contain exactly three literals. For instance, $(\neg x \vee y \vee z) \wedge (x \vee \neg y \vee z)$ is an example of a 3-SAT problem since each clause contains exactly three literals.

In order to prove that 3-SAT is NP-Complete, we must show that 3-SAT is in the class of NP problems and that every problem in NP can be reduced to 3-SAT in polynomial time.

Theorem 6.0.1. 3-SAT $\in$ NP-Complete

Proof. We must first show that 3-SAT $\in$ NP. However, this directly follows from the proof that SAT $\in$ NP shown in Section 4. Thus, we can check a certificate of a given 3-SAT problem in polynomial time.

Next, we must show that for every problem X in NP, X $\leq$ 3-SAT. However, in order to show this, we can show that one problem in NP is time reducible to 3-SAT.

Therefore, we can find a function f that can be computed in polynomial time and takes a boolean expression and yields an expression with three literals per clause.

Given clausal formula ϕ, we can work on one clause C at a time in order to expand or reduce it to three literals. There are three distinct cases for each clause.

Case 1: C has exactly three literals
Do not change the clause.

Case 2: C has less than three literals
If there is only one literal in the clause, duplicate the literal twice, and if there are two literals, duplicate the first literal once. Add a disjunction between the literals. For instance, $\neg x$ becomes $(\neg x \vee \neg x \vee \neg x)$ and $(x \vee z)$ becomes $(x \vee x \vee z)$.

Case 3: C has more than three literals
Suppose clause C has n literals and takes the form $(l_1 \vee l_2 \vee l_3 \vee ... \vee l_n)$, where n $>$ 3. We can define n-3 propositions $(\lambda_1, \lambda_2, \lambda_3, ..., \lambda_{n-3})$ and break clause C into clauses with three literals. Each clause will contain one or two λ variables. The clauses will take the following form:

$$(l_1 \vee l_2 \vee \lambda_1) \wedge$$

$$(\neg\lambda_1 \vee l_3 \vee \lambda_2) \wedge$$

...

$$(\neg\lambda_{n-4} \vee l_{n-2} \vee \lambda_{n-3}) \wedge$$

$$(\neg\lambda_{n-3} \vee l_{n-1} \vee l_n)$$

For instance, suppose C = $(\neg x \vee y \vee \neg z \vee \neg a \vee b \vee c)$. The corresponding clauses would be $(\neg x \vee y \vee \lambda_1) \wedge (\neg\lambda_1 \vee \neg z \vee \lambda_2) \wedge (\neg\lambda_2 \vee \neg a \vee \lambda_3) \wedge (\neg\lambda_3 \vee b \vee c)$.

We must verify that this reduction preserves the satisfiability of the original clause. To do this, We must show that when C_i is satisfiable the corresponding expanded clauses are satisfiable. We must also show that when C_i is not satisfiable the expanded clauses are also not satisfiable.

1. C_i is satisfiable

When C_i is satisfiable, at least one literal from $(l_1 \vee l_2 \vee l_3 \vee ... \vee l_n)$ must be true.

If l_1 or l_2 are true, set $(\lambda_1, \lambda_2, \lambda_3, ..., \lambda_{n-3})$ to be false. Each clause in the expanded clause, except for the first, contains $\neg\lambda_i$, which is true. Therefore, the expanded clauses maintain the satisfiability of C_i.

If l_{n-1} or l_n are true, set $(\lambda_1, \lambda_2, \lambda_3, ..., \lambda_{n-3})$ to be true. Each clause in the expanded clause, except for the last, contains λ_i, which is true. Therefore, the expanded clauses maintain the satisfiability of C_i.

If l_i is true where $i \notin 1, 2, l_{n-1}, l_n$, set $(\lambda_1 ... \lambda_{i-2})$ to true and set $(\lambda_{i-1} ... \lambda_{n-3})$ to false. Therefore, all the clauses before the clause containing l_i will evaluate to true because the $(\lambda_1 ... \lambda_{i-2})$ literals, the right-most terms, are true. Additionally, all the clauses after the clause containing l_i will evaluate to true because the $(\lambda_{i-1} ... \lambda_{n-3})$ literals, the left-most terms, are true. The clause containing l_i is true because l_i is true. Therefore, the expanded clauses maintain the satisfiability of C_i.

2. C_i is not satisfiable

When C_i is not satisfiable, none of the literals from $(l_1 \vee l_2 \vee l_3 \vee ... \vee l_n)$ are true. If we set all the values of $(\lambda_1, \lambda_2, \lambda_3, ..., \lambda_{n-3})$ to be true, the last clause will be false, making the conjunction of the clauses false. Therefore, the expanded clauses maintain that C_i is not satisfiable.

Since there are a fixed number of steps that need to take place for each clause, this algorithm takes polynomial time. Therefore, SAT is polynomial time reducible to 3-SAT, making 3-SAT an NP-Complete problem.

□

7 Applications

The infamous P vs. NP problem has been labeled the most important unsolved problem in computer science. It is one of the seven Millennium Prize Problems. The problem asks whether every problem that can be verified in polynomial time can also be solved in polynomial time. If P = NP, then the class of NP problems can be computed in polynomial time. As of right now, modern technology and computing operates on the assumption that P $\neq$ NP.

Computational theory and the implications of NP-Completeness serve several distinct purposes in various fields of computer science. The following subsections provide a more detailed analysis of how branches of computer science are affected.

7.1 Artificial Intelligence

In artificial intelligence, computational theory forms the foundation of determining what can and cannot be learned from data with the use of algorithms. Artificial intelligence algorithms use computational models of perception, control, decision-making, and pattern recognition in order to optimize an AI agent's behavior. Further sub-fields of artificial intelligence, such as neural networks, use computational models to simulate the behavior of the human brain and its neuron activity. If it were proven that P = NP, these algorithms would become increasingly efficient and optimized.

7.2 Cybersecurity

Computational theory is used to design "computationally secure" cryptographic algorithms in cybersecurity. While in theory these algorithms can be successfully cracked, in practice it is deemed unfeasible due to exponential time complexities. Thus, the entire process of authentication and secure transactions relies on these unsolvable algorithms. If it were proven that P = NP, all cryptographic methods currently used would become vulnerable to attackers. Additionally, large amounts of private data would be made public. Public-key cryptography relies on the constraints of integer factorization and the corresponding exponential complexity; however, if P = NP, then there would exist some polynomial time algorithm to factor large integers, breaking public-key cryptography.

7.3 Computer Architecture

Similarly, computer architecture, or the design and organization of modern computer systems, relies on parallel computing and complex memory hierarchy structures. Computational models are used to design algorithms that effectively and simultaneously make use of several processors. Related models are also used to optimize access of memory locations, such as RAM, cache, and disk. If it were proven that P = NP, these models would become increasingly optimized, leading to faster machines and improved hardware.

References

[1] Arnav Attri. *Time complexity of Euclid's Algorithm*. URL: https://stackoverflow.com/questions/3980416/time-complexity-of-euclids-algorithm. (accessed: 05.01.2023).

[2] Liesbeth De Mol. "Turing Machines". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2021. Metaphysics Research Lab, Stanford University, 2021.

[3] GeeksForGeeks. *Introduction to NP-Completeness*. URL: https://www.geeksforgeeks.org/introduction-to-np-completeness/. (accessed: 05.01.2023).

[4] OpenDSA. *Reduction of SAT to 3-SAT*. URL: https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/SAT_to_threeSAT.html. (accessed: 05.01.2023).

[5] Umang Sharma. *Types of Complexity Classes — P, NP, CoNP, NP hard and NP complete*. URL: https://www.geeksforgeeks.org/types-of-complexity-classes-p-np-conp-np-hard-and-np-complete/. (accessed: 05.01.2023).

[6] East Carolina University. *3-SAT*. URL: http://www.cs.ecu.edu/karl/6420/spr16/Notes/NPcomplete/3sat.html. (accessed: 05.01.2023).

[7] Wikipedia. *Boolean satisfiability problem*. URL: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem. (accessed: 05.01.2023).

[8] Wikipedia. *P versus NP Problem*. URL: https://en.wikipedia.org/wiki/P_versus_NP_problem. (accessed: 05.01.2023).

[9] Wikipedia. *Subset sum problem*. URL: https://en.wikipedia.org/wiki/Subset_sum_problem. (accessed: 05.01.2023).

[10] Wikipedia. *Turing machine*. URL: https://en.wikipedia.org/wiki/Turing_machine. (accessed: 05.01.2023).