Article

Not peer-reviewed version

A Constructive Proof That P ≠ NP

Guanggian Zhang *

Posted Date: 10 June 2025

doi: 10.20944/preprints202506.0747.v1

Keywords: combinatorial problems; the intersection of two set; computational complexity; 0-1 integer programming; NP-complete (NPC); P≠NP

 doi:10.20944/preprints202506.0747.v1

Article

A Constructive Proof That P ≠ NP (Constructive Proof of P ≠ NP)

Guangqian Zhang

Institute of System Engineering, Dalian University of Technology, China; zhgq@dlut.edu.cn

Abstract: Informed by logical principles, this study initiates its inquiry from a combinatorial problem. First, it demonstrates that a linear equation with 0-1 variables inherently represents a set. Building on this foundation, a 0-1 integer programming model composed of two such linear equations is constructed. From the perspective of set intersection, it is proven that this specific 0-1 integer programming admits no polynomial-time solution. Consequently, the general form of 0-1 integer programming defined by $\boldsymbol{CX=d}$ is not solvable in polynomial time. Given that the 0-1 integer programming of the form $\boldsymbol{CX=d}$ is a recognized NP-complete problem, this result implies that all NP-complete problems lack polynomial-time solutions, thereby establishing P≠NP. This work offers novel insights into the P vs. NP problem, advancing the theoretical understanding of computational complexity.

Keywords: combinatorial problems; the intersection of two set; computational complexity; 0-1 integer programming; NP-complete (NPC); P≠NP

1. Introduction

The question of whether P equals NP is a fundamental problem in theoretical computer science, with profound connections to numerous fields, including mathematics, cryptography, and artificial intelligence. Its resolution would enable precise gauge the difficulty of problems. It holds great significance for enhancing our understanding of the nature of computation and driving the advancement of computational complexity theory and algorithm design. Additionally, it is one of the Millennium Prize Problems [1].

As one of the major open problems, since Steve Cook first posed this question in 1971 [2], it has garnered extensive attention due to its crucial theoretical significance and practical value.

To date, the problem remains open [3]. Some scholars hold the view that P = NP, while others believe P ≠ NP. The prevailing opinion leans towards the latter, yet no significant breakthrough has been achieved to confirm either position definitively.

The current state of research on the P vs. NP problem can be summarized as "one scarcity and one abundance". One scarcity refers to there is a scarcity of formally published research specifically dedicated to the P vs. NP problem. One abundance refers to there is an abundance of attention and discussions about the P vs. NP problem on the Internet.

For example [3], research endeavors to prove P≠NP include analyzing the satisfiability problem via diagonalization techniques, attempting to resolve specific NP-complete (NPC) problems using logic circuits, and demonstrating that tautologies lack short proofs in any proof system. Conversely, studies aiming to prove P=NP involve showing that the maximum independent set problem for any graph admits a polynomial-time solution by transforming arbitrary graphs into perfect graphs [4]. Additionally, geometric complexity theory has been employed to investigate the P vs. NP problem [5].

However, these previous attempts have yet to resolve whether P equals NP. This problem is not only a fundamental issue in theoretical computer science but also pertains to nearly all areas across

 doi:10.20944/preprints202506.0747.v1

other disciplines involving computation. While not much research on the P vs. NP problem has seen in journals, this should only be the tip of the iceberg. There are many efforts that have yet to be published. This is evident on the Internet, where numerous discussions about this problem. These discussions encompass diverse levels and perspectives, engaging a broader audience.

The primary challenge in investigating the P vs. NP problem lies in the inherent difficulty of formalizing it into rigorous mathematical formulations. As a result, researchers universally opt to study the P vs. NP problem through the lens of specific NPC problems. The rationale for this approach is as follows.

Two key results regarding the P vs. NP problem have been established in the literature [2,6,7]. First, numerous classical problems—including the satisfiability problem, 0-1 integer programming, and set packing—have been identified as NPC problems. Second, all NPC problems are polynomial-time reducible to one another.

Therefore, if a polynomial-time solution is proven for any NPC problem, it would establish that P = NP. Conversely, demonstrating the nonexistence of polynomial-time solutions for any NPC problem would confirm P≠NP. To date, over a thousand NPC problems have been identified, offering a rich landscape of candidates for investigating the P vs. NP problem.

In this discussion, it is contended that P≠NP from the perspective of 0-1 integer programming. A central challenge in proving P≠NP lies in the fact that the proposition cannot be established merely by demonstrating that a specific algorithm fails to be a polynomial-time solution.

In accordance with the fundamental principles of logic, a truth must hold universally. If P≠NP were true, contradictions would necessarily emerge in some common problems.

Thus, a constructive proof approach is adopted: if a problem can be identified for which no polynomial-time solution exists and which can be reduced to a specific NPC problem, this would formally establish that P ≠ NP.

2. An Instance of Combination

Define an event B: choose any m items from a set of n items that are each distinct ($1 \le m \le n$).

There are $C_{n,m}$ ways to choose m items from n items. In other words, the number of desired combinations is given by the binomial coefficient $C_{n,m}$, where

$$C_{n,m} = \binom{n}{m} = \frac{n!}{m!\,(n-m)!}$$

All desired combinations form a set denoted as E_1. The j^{th} element of E_1 is denoted as e_j, and the cardinality of E_1 is $C_{n,m}$. The assignment of which combination corresponds to the j^{th} element is purely for convenience and can be arbitrarily specified. If the n items are treated as a set E, then E_1 is a subset of the power set of E.

The set E can comprise almost any n distinct items, with no inherent binary relationships—such as magnitude or inclusion—defined among them. The elements of set E_1 are formed by the desired combinations of distinct elements from E. Analogously, no intrinsic binary relationships (e.g., size or subset relations) exist between these combinations, akin to how words formed by letter combinations carry no inherent connections. Consequently, both E and E_1 are finite, discrete, and unordered sets. Unless otherwise specified, the term "set" in this paper refers to a discrete, finite, and non-partial ordered set, exemplified by E and E_1.

3. Mathematical Modeling of Event B and Related Issues

3.1. The Mathematical Model of B and Its Expansions

If the n items are numbered from 1 to n and each item is assigned a 0-1 variable x_i, such that when the i^{th} item is selected, $x_i = 1$; otherwise, $x_i = 0$, then the corresponding mathematical model for event B is as follows.

$$\sum_{i=1}^{n} x_i = m \quad (1)$$

 doi:10.20944/preprints202506.0747.v1

Equation (1) formalizes the selection of exactly m items from n items, representing event B, with its solution set being E_1. Structurally, Equation (1) is a 0-1 linear equation, yet its essence is a set. Here, the elements of E_1 are n-dimensional 0-1 vectors, where each vector uniquely corresponds to a distinct combination, establishing a bijection between combinations and vectors.

Similarly, equation (2) characterizes a set when x_i is a 0-1 variable. Suppose all n items in event B possess a specific attribute, such as weight or volume, with a_i denoting the attribute value of the i^{th} item. When the i^{th} item is selected it takes up a_i units out of a total of b units (by analogy, Equation (1) corresponds to the scenario where the i^{th} item occupies 1 unit of a total capacity m.). All $a_i (i = 1, \ldots, n)$ and b are assumed to be integers. Equation (2) defines a set based on a partitioning criterion distinct from that of Equation (1).

$$\sum_{i=1}^{n} a_i x_i = b \ (2)$$

The set corresponding to Equation (2) is denoted as E_2. Analogous to E_1, E_2 is also a subset of the power set of E.

3.2. Properties of Sets E1 and E2

When the n items are numbered and their selection status is encoded by 1 and 0. As a result, the elements of E_1 and E_2 are naturally represented as binary vectors. Each vector corresponds to a unique combination of selected items.

In sets E_1 and E_2, elements exhibit no ordinal relationships (e.g., magnitude). For example, consider the solutions $(1,0,0)^T$ and $(0,1,1)^T$ to the equation $2x_1 + x_2 + x_3 = 2$. Neither solution is inherently "greater" or "smaller" than the other. From the practical implications of the elements in E_1 and E_2, and even when restricted to 0-1 linear equations alone, no universally accepted natural ordering exists to sequence the solutions within E_1 or E_2. Additionally, as (1) and (2) are equations, no inclusion relationships exist among the elements within E_1 or E_2. In essence, there are no inherent binary relations—such as magnitude or inclusion—between the elements of E_1 and E_2. Consequently, after 0-1 encoding, E_1 and E_2 remain finite, discrete, and unordered sets.

If a convex set S is defined by $0 \leq x_i \leq 1 \ (i = 1, \ldots, n)$, then encoding the n items is equivalent to mapping the elements in E_1 and E_2 to the corresponding vertices of S. This mapping will introduce a certain ordering aspect, yet it does not fundamentally alter the intrinsic relationships among the elements.

The relationship between E, E_1 and S is akin to the relationship between words, books, and bookshelves. Words combine to create books. For efficient book management, books are often coded and placed in corresponding positions on the bookshelf for easy retrieval. The code tells us where a book is located, but without reading the book itself, the code alone does not reveal its content. The arrangement of books does not alter their inherent relationships. While this encoding introduces a specific order, it does not fundamentally transform the connections between them.

3.3. An Estimation of the Cardinality of Set E2

The cardinality of E_1 is well-established, whereas that of E_2 remains unknown. Thus, it is necessary to make a reasonable estimation of the number of elements in E_2.

In general, for sets, greater cardinality implies more complex corresponding operations. Notice that the geometric interpretation of equation (2) is a hyperplane intersecting the convex set S, with integer-valued intersection points comprising the elements of E_2. When the cardinality of E_2 is maximized, it approximates the middle term of the n^{th} row in Pascal's Triangle. That is, the maximum number of elements of E_2 is on the order of $C_{n,[n/2]}$, where $[n/2]$ denotes the greatest integer not exceeding $n/2$.

The cardinality of E_1 is also maximized when $m = [n/2]$. Therefore, in the context of computational complexity analysis, it is reasonable to assume that the cardinality of E_2 is on the order of $C_{n,m}$.

 doi:10.20944/preprints202506.0747.v1

4. The 0-1 Integer Programming

Combining Equation (1)and Equation (2)yields the following 0-1 integer programming.

$$(\mathrm{I}_1)\begin{cases}\sum_{i=1}^{n} x_i = m\ (1)\\ \sum_{i=1}^{n} a_i x_i = b\ (2)\end{cases}$$

(I_1) is a 0-1 integer programming that seeks to determine all x_i satisfying both Equations (1) and (2) simultaneously. In other words, solving (I_1) is equivalent to computing the intersection of E_1 and E_2. For clarity, the expressions for E_1 and E_2 are presented below.

$$E_1: \{(x_1, x_2, \cdots, x_n)^T \mid \sum_{i=1}^{n} x_i = m, x_i = 0 \text{ or } 1, i = 1,2,\cdots,n\}$$
$$E_2: \{(x_1, x_2, \cdots, x_n)^T \mid \sum_{i=1}^{n} a_i x_i = b, x_i = 0 \text{ or } 1, i = 1,2,\cdots,n\}$$

The above description of E_1 and E_2 follows a standard set-theoretic representation. Within this framework, the following proposition is put forward.

4.1. Proposition on the Intersection of Two Sets

Proposition 1: *When computing the intersection of two nonempty sets, each element must be subjected to at least one basic operation.*

Proof: Let D_1 and D_2 denote any two nonempty sets. If an element $d_i \in D_1$ is left out, then for any algorithm computing the intersection of the two sets, the resulting intersection cannot contain d_i. If D_2 contains an element identical to d_i, the computed result will be incorrect. Indeed, failing to consider d_i makes it impossible to determine whether D_2 contains a matching element, thereby rendering the correctness of the intersection unverifiable. Consequently, any algorithm designed to compute the intersection of D_1 and D_2 must account for d_i. When an algorithm considers an element, this inherently requires performing at least one basic operation on that element.

By symmetry, since d_i is an arbitrary element of D_1, the same logic applies to all elements in both sets. Thus, regardless of the algorithm used to compute the intersection, every element in D_1 and D_2 must undergo at least one basic operation.

Q.E.D.

Proposition 1 formulates the principle of two-set intersection at the element level, independent of specific set representations or intersection-finding algorithms.

In mathematics, numerous fundamental theorems and properties govern set operations, including the commutative, associative, and distributive laws for intersection, union, and complement operations. These theorems primarily focus on the relationships and rules that underpin set operations.

By contrast, Proposition 1 examines the properties of the intersection computation process at the algorithmic operation level. While Proposition 1 may appear intuitive, it has no direct counterpart in classical set theory theorems. In computer science, this property is frequently invoked in analyses of algorithmic complexity and correctness, often tacitly assumed as a foundational axiom in such contexts.

As Proposition 1 establishes a lower bound on the computational complexity of set intersection and forms the cornerstone of the entire proof, it is explicitly stated here.

There are already several polynomial time algorithms for finding the intersection of two sets [8,9]. These include the naive set intersection algorithm, the hash table based set intersection algorithm, and the sorting and two pointer method based set intersection algorithm. The latter two algorithms are regarded as having linear complexity, yet both rely on specific preconditions. For instance, a suitable hash table is needed for the hash table based algorithm, and the elements must be sortable for the sorting and two pointer method based algorithm.

Note that some algorithms for computing the intersection of two sets can be decomposed into two stages: Preprocessing and Querying. During Preprocessing, one set is transformed into a query-

 doi:10.20944/preprints202506.0747.v1

optimized data structure (e.g., a hash table). In the subsequent Query phase, the intersection algorithm iterates through each element of the second set to derive the intersection.

Notably, the Preprocessing phase processes all elements of one set, while the Query phase processes those of the other. Consequently, every element in both sets is processed at least once. When the algorithm focuses exclusively on the Query phase (assuming Preprocessing is already complete), the computational complexity of solving the intersection is determined solely by this phase. In this context, Proposition 1 can be articulated as follows:

Any correct algorithm designed to compute the intersection of two sets must examine every element in at least one of the sets.

This formulation may be termed Version 2 of Proposition 1, with the original proposition designated as Version 1. The two versions share identical core semantics, differing only in scope: Version 1 encompasses the entire computational process, whereas Version 2 focuses exclusively on the Query phase.

Preprocessing typically encompasses operations like data presorting or hash table construction. However, specific implementations of Preprocessing and the rationale for partitioning algorithms into Preprocessing and Query phases are not elaborated here. In cases where the appropriate division of an algorithm into these two phases is uncertain, Version 1 shall take precedence.

4.2. The Lower Bound of Computational Complexity for the Intersection of Two Sets

Let the cardinality of set D_1 be n_1 and that of set D_2 be n_2. If $d_1 \in D_1$ and $d_2 \in D_2$, when comparing the identity of d_1 and d_2, both d_1 and d_2 have undergone at least one basic operation from an individual perspective. Thus, by Version 1 of Proposition 1, the computational complexity of finding the intersection of two sets is at least on the order of $(n_1 + n_2)/2$, even if comparing two vectors is treated as a single elementary operation. Under Version 2 of Proposition 1, the computational complexity of finding the intersection of two sets is at least $min(n_1, n_2)$, when preprocessing the set with the larger cardinality and regarding the query of an element in the preprocessed set as a basic operation. Since $(n_1 + n_2)/2 \geq min(n_1, n_2)$, the computational complexity of finding the intersection of the two sets D_1 and D_2 is fundamentally bounded by $min(n_1, n_2)$, i.e., $\Omega(min(n_1, n_2))$, which indicates that the most efficient algorithm for finding the intersection of two sets may be linear, but never $O(1)$. This result can be formalized as the following corollary.

Corollary: *Let the cardinalities of two sets D_1 and D_2 be n_1 and n_2 respectively. The lower bound of the computational complexity for computing the intersection of these two sets is $min(n_1, n_2)$, that is, $\Omega(min(n_1, n_2))$.*

5. The Computational Complexity of $(\mathbf{I_1})$

5.1. The Relativity of Computational Complexity

It is necessary to recall the definitions of P-type and NP-type problems[2,6]: "Let P be the class of languages recognizable in polynomial time by one-tape deterministic Turing machines, and let NP be the class of languages recognizable in polynomial time by one-tape nondeterministic Turing machines.". Additionally, a theorem regarding NP class problems states[6]: "$L \in NP$ if and only if L is accepted by a nonderministic Turing machine which operates in polynomial time.".

From these definitions and the theorem, for NP-class problems, the computational complexity is polynomial when evaluated on nondeterministic Turing machines. However, when evaluated on deterministic Turing machines, their complexity is characterized as nondeterministic polynomial time. This demonstrates that the computational complexity of NP-class problems is relative and depends on the underlying computational model(different types of Turing machines have different computational capabilities).

Although NP-type problems exhibit divergent computational complexity characteristics under different measurement criteria, such disparities are not always evident for a specific problem. Fortunately, defining two distinct benchmarks for measuring the intersection of E_1 and E_2 is relatively straightforward.

 doi:10.20944/preprints202506.0747.v1

Existing research confirms that computing the intersection of two sets is unequivocally a P-class problem. However, it is critical to emphasize the relativity of computational complexity in the context of set intersection computation.

For instance, polynomial-time algorithms exist to determine the intersection of $E_1$and E_2 based on their cardinalities. However, if a different benchmark for measuring algorithmic complexity is employed, then by the Corollary, when the number of elements in both E_1 and E_2 is on the order of $C_{n,m}$, the computational complexity of finding their intersection is $\Omega(C_{n,m})$ with respect to n and m.

5.2. The Role of $\boldsymbol{C_{n,m}}$ in Complexity Analysis

If $m = 1$ or $n - 1$, then

$$C_{n,m} = n \ (3)$$

Analyze $C_{n,m}$ using Stirling's formula:

$$C_{n,m} \approx \frac{1}{\sqrt{2\pi}} \sqrt{\frac{n}{m(n-m)}} \times \frac{n^n}{m^m (n-m)^{n-m}}$$

If $m = [n/2]$, then

$$C_{n,m} \approx \sqrt{\frac{2}{\pi}} \cdot \frac{2^n}{\sqrt{n}} \ (4)$$

It can be observed that for the computational complexity $C_{n,m}$, when m is either extremely small or extremely large, $C_{n,m}$exhibits approximately linear behavior, as demonstrated in Equation (3). Conversely, when m approaches the median of n, $C_{n,m}$ grows exponentially, as shown in Equation (4). As m varies, the complexity $C_{n,m}$ transitions from polynomial to non-polynomial.

In discussions of algorithmic complexity, the focus is typically on the worst-case scenario. Although both cases (3) and (4) are valid, computational complexity $C_{n,m}$ should be assessed based on case (4) unless explicit constraints on m are imposed—such as $m \ll n$ or m being a constant.

Furthermore, 2^n typically denotes the cardinality of the power set of a set with n elements, or the number of all vertices of S in an n-dimensional space. Its practical significance often relates to complete enumeration. Although Equation (4) includes a $\sqrt{n}$ term in the denominator, implying an incomplete enumeration. However, this expression closely approximates complete enumeration for large n.

5.3. (I_1) Admits No Polynomial-Time Solution

The scale of (I_1) is generally characterized by the number of its variables and constraints, implying that the existence of polynomial time solutions for (I_1) hinges on n and m, rather than $C_{n,m}$. The core focus is whether (I_1) admits a polynomial-time solution. Hence, unless explicitly stated otherwise, all computational complexity discussions hereinafter in this paper refer to the parameters n and m, where m is a parameter that geometrically represents the scale of a linear equation with 0-1 variables.

Equation (1) and Equation(2) represent E_1 and E_2, respectively, and solving (I_1) yields the intersection of E_1 and E_2. In other words, solving (I_1) is equivalent to computing the intersection of E_1 and E_2. The specific representation of the sets E_1 and E_2 does not affect the essence of the fact that E_1 and E_2 are sets. This scenario aligns with Proposition 1 and its Corollary, from which it follows directly that the computational complexity of solving (I_1) is $\Omega(C_{n,m})$. This immediately implies that (I_1) does not have a polynomial-time solution.

To strengthen the persuasiveness, the computational complexity of problem (I_1) can be further elaborated as follows, as illustrated in Figure 1.

 doi:10.20944/preprints202506.0747.v1

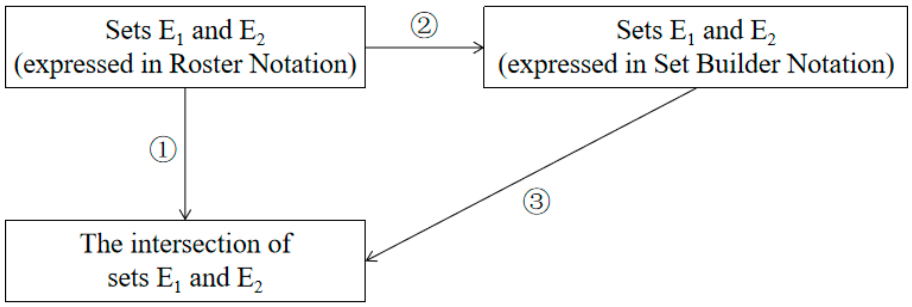

Figure 1. Two Methods for Solving the Intersection of Sets E_1 and E_2.

Sets are typically represented in two canonical forms: enumeration and description, commonly termed Roster Notation and Set Builder Notation, respectively. When discussing algorithms for computing the intersection of two sets earlier, the sets in question are implicitly assumed to be represented in Roster Notation by default.

Given the two canonical set representation methods, there are two distinct approaches to computing the intersection of sets E_1 and E_2 (depicted in Figure 1):

Approach I: Directly compute the intersection of E_1 and E_2 using their Roster Notation representations, as outlined in Procedure ①.

Approach II: First convert the set representations from Roster Notation to Set Builder Notation (corresponding to Procedure ②), then solve the associated 0-1 integer programming (corresponding to Procedure ③) to derive the intersection of E_1 and E_2. This method can be succinctly denoted as "②+③".

Although Approach I and Approach II share the same objective, they differ in their intermediate steps, representing distinct pathways to achieve the same result.

For Approach II, first, not all sets admit both Roster and Set-Builder representations. However, this paper explicitly focuses on a class of sets that can be characterized by 0-1 linear equations. Second, Procedure ② entails constructing corresponding 0-1 linear equations (1) and (2) based on the elements (points in n-dimensional space) of E_1 and E_2, respectively. Procedure ③ involves solving the 0-1 integer programming (I_1).

In n-dimensional space, a hyperplane is uniquely defined by n affinely independent points and can be algebraically represented as a linear equation. The process of determining such a hyperplane from n points is equivalent to solving a system of linear equations. While the cardinalities of sets E_1 and E_2 are on the order of $C_{n,m}$, the affine independence of these points (as they correspond to vertices of hypercube S) ensures that only n points from each set are required to construct the hyperplane. Notably, the hyperplane can also be generated using a constant multiple of n elements (e.g., $2n$ or $3n$), with its computational complexity remaining within polynomial time.

The constraint that variables are 0-1 variables manifests primarily in two aspects: first, the coefficients of the resulting linear equations are all integers; second, when a variable can assume multiple non-zero values, it is consistently set to 1. By appending the 0-1 variable constraints to the two linear equations solved above, the sets E_1 and E_2 are defined in Set Builder Notation, thereby completing Procedure ②.

Approach I involves directly computing the intersection of sets E_1 and E_2 in their Roster Notation representations, as outlined in Procedure ①. By the Corollary, the computational complexity of determining the intersection of E_1 and E_2 via this approach is $\Omega(C_{n,m})$. Since there is no constraint specifying m to be a constant or significantly smaller than n, the computational complexity of basic operations with magnitude $C_{n,m}$ cannot be categorized as polynomial time.

When solving for the intersection of E_1 and E_2 using Approach II, the process also begins with sets in Roster Notation and culminates in deriving their intersection. By the Corollary, the computational complexity of the combined procedures "②+③" is $\Omega(C_{n,m})$.

Given that solving a system of linear equations is a known P-type problem, the computational complexity of Procedure ② is polynomial time. Consequently, the non-polynomial complexity of the

 doi:10.20944/preprints202506.0747.v1

combined procedure "②+③" must originate from Procedure ③. In computational complexity analysis, polynomial terms are negligible compared to non-polynomial terms, meaning the overall complexity of "②+③" is dominated by Procedure ③. This implies that solving (I_1) inherently requires $\Omega(C_{n,m})$ operations, regardless of the specific algorithm employed for solving 0-1 integer programming. Therefore, no polynomial-time algorithm exists for computing the intersection of E_1 and E_2 with respect to n and m via solving the 0-1 integer programming (I_1).

Thus, 0-1 integer programming (I_1) admits no polynomial-time solution.

6. P≠NP

The 0-1 integer programming in the form $\boldsymbol{CX} = \boldsymbol{d}$ is an NPC problem, specifically the second on Karp's list of 21 NP-complete problems proposed in 1972 [6]. This problem involves a system of 0-1 linear equations where both matrix $\boldsymbol{C}$ and vector $\boldsymbol{d}$ consist of integers. Clearly, (I_1) is a special case of 0-1 integer programming in the form of $\boldsymbol{CX} = \boldsymbol{d}$. Hereafter, $\boldsymbol{CX} = \boldsymbol{d}$ is denoted as (I).

The relationship between (I_1) and (I) is clear-cut. If (I) admits a polynomial time solution, then (I_1) bound to have one. Conversely, if (I_1) has no polynomial time solution, then (I) cannot possess one either.

Since it has been established that (I_1) admits no polynomial time solution, it directly follows that (I) has no polynomial time solution.

Given that (I) is a recognized NPC problem, it follows that all NPC problems have no polynomial time solution. Therefore, it can be concluded that P ≠ NP.

Comment. The relationship between P and NP is a fundamental issue in computational complexity theory. Logically, if P = NP, the two would be consistent across all problems. Conversely, if P ≠ NP, contradictions must arise in some common problems.

The problem selected here is a combinatorial problem. The set with $C_{n,m}$ elements can be expressed as a linear equation with 0-1 variables (or a 0-1 integer programming). Given the exponential growth of $C_{n,m}$ with respect to n and m, it is implausible to construct such a set in polynomial time by solving Equation (1). This observation strengthens the hypothesis that P ≠ NP.

0-1 linear equations essentially represent sets. When it comes to computational complexity, operations on sets are gauged by the cardinal numbers of those sets. In contrast, 0-1 integer programming measures complexity by the number of variables and equations involved. This divergence in perspective between these two computational frameworks leads to the conclusion that P ≠ NP. As the proverb goes, "All things have cracks, and that is where the light shines in.". This disparity is where the light shines in in the P vs. NP problem.

Moreover, although the lower bound of the computational complexity for the intersection of two sets is well-established, when the number of elements in the two sets is on the order of $C_{n,m}$, and the computational complexity of the intersection algorithm is measured in terms of n and m, the lower bound becomes less obvious. As m varies, the induced computational complexity transitions from polynomial (P-type) to non-polynomial (NP-type), a phenomenon that likely contributes to the challenge of rigorously distinguishing between P and NP.

7. Conclusions and Future Prospects

This paper presents a novel interpretation of the P vs. NP problem through the interplay between set theory and 0-1 integer programming. Starting from a combinatorial problem, it establishes that 0-1 linear equations fundamentally characterize sets. Leveraging this equivalence, a 0-1 integer programming composed of two 0-1 linear equations is constructed. From the perspective of set intersections, it is proven that this problem does not admit a polynomial-time solution. By synthesizing with established results in computational complexity theory, the conclusion that P ≠ NP is logically derived.

Formally, 0-1 integer programming is analogous to linear programming (LP) and linear systems of equations, having fully incorporated advancements from both fields. Researchers commonly analyze 0-1 integer programming through the frameworks of LP or linear systems of equations.

 doi:10.20944/preprints202506.0747.v1

However, a fundamental distinction exists between 0-1 integer programming and these domains. While 0-1 linear equations are frequently employed as algebraic models for combinatorial problems—though not all combinatorial problems are amenable to such modeling—they typically represent resource allocation schemes, encoding all feasible allocations that satisfy the constraints. Viewing 0-1 integer programming through a set-theoretic lens offers a critical perspective for deciphering the relationship between P and NP.

Anyone familiar with the P vs. NP problem knows that the crux of proving P ≠ NP lies in finding a non polynomial time lower bound for an NPC problem. This paper derives a lower bound for 0-1 integer programming by establishing the existence of a lower bound for the intersection of two sets. While computing the intersection of two sets is a P-class problem, the corresponding lower bound for the associated 0-1 integer programming is non-polynomial. Specifically, due to a distinct method of quantifying problem scale, its complexity exhibits approximately exponential growth. As the foundational pillar of the entire proof, the complexity of computing the intersection of two sets must not be constant but is inherently tied to the sizes of the sets. This result thus serves as critical evidence in support of the P ≠ NP conjecture.

0-1 integer programming is a canonical NPC problem. Specifically, (I_1), a 0-1 integer programming instance with two equality constraints, remains inherently an NPC problem. The computational complexity of (I_1) exhibits approximately exponential growth. Given that all NPC problems are polynomially reducible to one another, this indicates that the fundamental nature of NPC problems is rooted in enumeration. In essence, NPC problems are those that must be solved by exhaustive enumeration in the worst case.

The practical significance of studying the P vs. NP problem resides in assessing the real-world feasibility of algorithms. Specifically, it addresses whether a given problem admits an algorithm capable of providing an exact solution within a reasonable computational time, thereby determining its practical applicability.

The assertion that P ≠ NP implies inherent algorithmic limitations in dealing with NPC problems. However, as algorithms continue to advance, especially ongoing advancements in computational paradigms—such as ternary computers, quantum computers, and parallel processing —may enhance a computer's ability to execute more operations per unit time. This could lead to a scenario envisioned in the definition of NP problems: if machines with non-polynomial computational capabilities emerge, even NPC problems could be solved in polynomial time. Should such a technological breakthrough occur, while P ≠ NP would remain theoretically valid, the practical feasibility of solving NP problems could approximate P = NP in practice. The advent of new computational paradigms is inevitable, it is only a matter of how long it will take.

References

1. Carlson, J.A., Jaffe, A., Wiles, A.. The millennium prize problems. Cambridge, MA, American Mathematical Society, Providence, RI, Clay Mathematics Institute ,2006
2. Cook, S. The complexity of theorem-proving procedures. In Proceedings of the 3rd ACM Symposium on the Theory of Computing, ACM, NY, 1971, 151–158.
3. Fortnow L. The status of the P versus NP problem[J]. Communications of the ACM, 2009, 52(9):78-86. DOI:10.1145/1562164.1562186.
4. Heal, M., Dashtipour, K., Gogate, M.. The P vs. NP Problem and Attempts to Settle It via Perfect Graphs State-of-the-Art Approach. In: Arai, K. (eds) Advances in Information and Communication. FICC 2023. Lecture Notes in Networks and Systems, vol 652. Springer, Cham. https://doi.org/10.1007/978-3-031-28073-3_23
5. Mulmuley, K., Sohoni, M..Geometric complexity theory I: An approach to the P vs. NP and related problems. SIAM Journal on Computing, 2001,31(2): 496-526.
6. Karp, R.M.. Reducibility among combinatorial problems. In: Complexity of Computer Computations. Springer, 1972,pp. 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

 doi:10.20944/preprints202506.0747.v1

7. Michael R. Garey, David S. Johnson.Computers and Intractability： A Guide to the Theory of NP-completeness, W.H.Freeman & Co Ltd., New York,1979.4
8. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C.. Introduction to Algorithms(4th ed.). MIT Press and McGraw-Hill, 2022.
9. Weiss, M. A.. Data Structures and Algorithm Analysis in C (3rd ed.). Addison-Wesley Professional,2014