Bifurcating Power Sets to Show *P* Does Not Equal *NP*

Dharmarajan R * and Ramachandran D

Posted Date: 8 October 2025

doi: 10.20944/preprints202510.0617.v1

Keywords: Algorithm; polynomial time; certi cate; power sets; class P; class NP

 doi:10.20944/preprints202510.0617.v1

Article

Bifurcating Power Sets to Show *P* Does Not Equal *NP*

R. Dharmarajan * and D. Ramachandran

Niels Abel Foundation, Palakkad 678011, Kerala, India
* Correspondence: mathnafrd@gmail.com

Abstract

The ***P*** versus ***NP*** problem, a conjecture formulated by Stephen Cook in 1971, is one of the deepest and most challenging problems in contemporary mathematics and theoretical computer science. A concise mathematical formulation of the problem reads: **is *P* = *NP*?**. In longer phrasing, this asks: given a problem instance, if some additional data can be recognized fast enough as logically implying the existence of a solution (to the instance), then can a solution be computed fast enough without the aid of any such additional data? In this article explain why $\boldsymbol{P \neq NP}$ using a problem centered on bifurcating power sets.

Keywords: Algorithm; polynomial time; certificate; power sets; class ***P***; class ***NP***

MSC: 11Y16; 68W40; 03C70

---

1. Introduction

More on terminologies, symbols and notations used in this article can be found in [3] or [6]. Throughout this article, $\mathbb{N}$ will denote the set of positive integers and $\mathbb{W}$ the set of non-negative integers (i.e., $\mathbb{W} = \mathbb{N} \cup \{0\}$).

The main discussions and the findings in this article appear in six sections. Section 1 deals with some salient points on algorithms, the problem classes ***P*** and ***NP***, attempted solutions and feasible solutions. Section 2 contains essential set-theoretical concepts.

Section 3 contains an optimization version and a decision version of the problem that we formulate to show $\boldsymbol{P \neq NP}$. In Section 4 we present an algorithm that contributes to proving the problem is in ***NP***. In Section 5, we show this problem in not in ***P***. The concluding remarks are in Section 6.

1.1. Algorithms

An *algorithm* is any well-defined computational procedure that takes finitely many quantities as input and produces finitely many quantities as output. An algorithm is thus a sequence of well-defined computational steps transforming the input into the output [3].

An *instance* (also called *input instance* or *problem instance*) of a problem is an input satisfying all the constraints in the problem statement needed to compute a solution to the problem [3]. For example, suppose the problem is to arrange a given finite sequence of integers in non-descending order of their magnitudes. Then any list consisting of finitely many integers is an instance of this problem. One instance is: 300, 10, -18, 231, -117, 10, -117, 7, 10, 0.

How the *size* (or *length*) of the instance is defined depends on the problem $\mathcal{Q}$ being studied [3]. For example, if $\mathcal{Q}$ is the problem seen in the preceding paragraph, then the size of the instance given there is $n = 10$ (since all the repetitions have to be counted in). More examples of instances and input sizes are in Appendix A of [4].

A "step" in an algorithm is a primitive operation executed by the algorithm. In dealing with any algorithm, there is a presupposition that every primitive operation to be executed by the algorithm is unambiguously defined. Since we furnish algorithms only in lines of pseudocodes, we adopt the

 doi:10.20944/preprints202510.0617.v1

following point of view for the notion of number of steps [3]: we assume that each execution of the j^{th} line of the pseudocode takes t_j steps (meaning, t_j primitive operations).

From a computational point of view, given a problem $\mathcal{Q}$, it is natural to ask if there exists, or can be developed, an algorithm that can process any given instance of $\mathcal{Q}$ to a required extent in such a way that the number of steps taken by the algorithm is bounded by a fixed polynomial in the size of the given instance [2]. Such an algorithm is said to process the input instance in *polynomial time* (or run in *polynomial time*; or take *polynomial time* to process) and is therefore called a *polynomial-time algorithm*. Note that an algorithm is deemed to run in polynomial time only vis-*à*-vis a specified problem $\mathcal{Q}$; in other words, the phrase 'the algorithm ALG runs in polynomial time' really means that there is a specified problem $\mathcal{Q}$ such that ALG processes each instance of $\mathcal{Q}$ in polynomial time.

When we say the number of steps taken by an algorithm is bounded by a fixed polynomial in the variable n, what we mean is: there exists a polynomial $f(n)$ that is fixed for $\mathcal{Q}$ (meaning, in turn, that the instance X of $\mathcal{Q}$ can vary but $f(n)$ does not, so long as the problem $\mathcal{Q}$ does not) such that given two instances (of $\mathcal{Q}$) of sizes n_1 and n_2, the number of steps taken by the algorithm is at most $f(n_1)$ to process the n_1-sized instance and at most $f(n_2)$ for the n_2-sized instance.

For the purposes of this article, we consider two types of algorithms: *solve-type* algorithms (that find a solution, or correctly establish the absence of any solution, to each input instance) and *check-type* algorithms (that do not find solutions but only check out whether any additional input that is claimed to be a solution to an instance is really so). One crucial difference between these two types is: a solve-type algorithm needs only the instance of the problem as input whereas a check-type algorithm needs the instance as well as an additional input in the form of an "attempted solution" or a "proposed solution"(called so since it might be a solution or not). A solution to an instance is also called a *feasible solution*. A solve-type algorithm is also called an *exact algorithm*.

Every algorithm considered in this article is assumed, or if necessary shown to be, *correct* in the sense that given an input instance, the algorithm halts with an output that is correct [3].

1.2. Certificate Candidates and Certificates

In this article, we consider two classes of problems that are important in the context of algorithms: ***P*** and ***NP***. Informally, the class ***P*** consists of those problems that are solvable in polynomial time [2,3]. This means to each problem in this class there exists a polynomial-time algorithm that solves each instance of the problem.

The class ***NP*** consists of those problems that are "verifiable" in polynomial time [3]. What is meant by calling a problem "verifiable" is that if, in addition to an instance of the problem, we are somehow given a "certificate" of a solution, then we could verify, using a check-type algorithm running in polynomial time, that the "certificate" is indeed a solution, or logically implies (i.e., confirms) the existence of a solution, to the given instance.

It is clear that a certificate is an input component. Also, a certificate only begins as an attempted solution, and might or might not turn out to be a feasible solution [5]. We therefore wish to distinguish between these two situations.

To this end we define a certificate candidate. A *certificate candidate* is a component that is input along with the problem instance under consideration, the intention (of the user) being to test whether or not the certificate candidate confirms the existence of a solution to the instance. So a certificate candidate is just an attempted solution in the first place.

Let $\mathcal{Q}$ be the problem under consideration, X a problem instance of size n and $C(X)$ the certificate candidate that accompanies X. The result of running an appropriate algorithm on X and $C(X)$ will be exactly one of the following three:

1.2(i) $C(X)$ is per se a solution to X (i.e., the attempted solution actually turns out to be a feasible solution).

 doi:10.20944/preprints202510.0617.v1

1.2(ii) $C(X)$ per se is not a solution but it logically implies the existence of a solution, whether or not the implied solution is subsequently made explicit (i.e., the attempted solution, though not a feasible solution by itself, implies the existence of a feasible solution).
1.2(iii) Neither is $C(X)$ a solution nor does it logically imply the existence of a solution.

Henceforth, if any certificate candidate $C(X)$ obeys 1.2(i) or 1.2(ii), then we will say '$C(X)$ yields a solution (to X)'. And by saying '$C(X)$ does not yield a solution (to X)' we will mean that $C(X)$ obeys 1.2(iii). Examples to distinguish a certificate candidate obeying 1.2(i) from one obeying 1.2(ii) are in Appendix B of [4].

Next, "checking out" a certificate candidate is different from "verifying" it. To *verify* a certificate candidate $C(X)$ is to have a check-type algorithm establish that $C(X)$ yields a solution to the instance X. If a certificate candidate is thus verified then it is a *certificate*; else it is just a certificate candidate. If a certificate candidate becomes a certificate, then we will say either 'the certificate candidate is verified' or 'the certificate is verified'. Such a verification is tantamount to saying "YES" to the question whether the instance X has a solution.

On the other hand, to *check out* $C(X)$ is to have a check-type algorithm decide whether or not $C(X)$ yields a solution to X. If the algorithm checking out $C(X)$ decides that $C(X)$ does not yield a solution then this decision is tantamount to saying "NO" to the question whether X can be solved with the aid of $C(X)$. This "NO" is not a conclusive response to whether X has a solution, but only rules that the certificate candidate $C(X)$ is not a certificate. Thus:

1.2(iv) "verifying" requires a certificate candidate that yields a solution to the given instance whereas "checking out" calls for only a certificate candidate that is not required to have any additional properties / features; and
1.2(v) an algorithm is deemed to have verified a certificate candidate $C(X)$ (meaning, $C(X)$ is deemed to have become a certificate) only when $C(X)$ is shown (by the algorithm) to yield a solution to X.

So, while a certificate candidate is only an attempted (or, proposed) solution, a certificate is a feasible solution. In other words, every certificate is a certificate candidate in the first place but not every certificate candidate becomes a certificate. Further, a verification of a certificate candidate necessarily begins as an exercise of a check-type algorithm checking out the candidate but every exercise of checking out a candidate need not culminate in verification.

Thus, certificates (feasible solutions) are special cases of certificate candidates (attempted solutions).

As noted in the penultimate paragraph of Subsection 1.1, an algorithm is said to solve an instance X of a problem $\mathcal{Q}$ if the algorithm produces a solution (to X) if one exists or correctly establishes the absence of a solution otherwise, in either case without the need for any certificate candidate. If an algorithm solves each instance of $\mathcal{Q}$ then the algorithm is said to solve $\mathcal{Q}$. An algorithm that solves a problem instance X and an algorithm that only checks out a certificate candidate $C(X)$ differ primarily in the following aspect: in the former, the input is (X, n) and the output is a conclusive response to the question of a solution to X while in the latter, the input is $(X, n, C(X))$ and the output is an affirmative or a negative response to the question whether $C(X)$ yields a solution to X. Note that in each of these cases, finitely many additional input components are allowed.

Now rises the question whether the algorithm that checks out a given certificate candidate does so in polynomial time, for this is crucial in finding out if the concerned problem is in ***NP***. This underlines the importance of the certificate candidate in this context. Given a problem $\mathcal{Q}$, what are needed to decide that $\mathcal{Q}$ can be included in ***NP***? We need: one, a check-type algorithm - call it $AL(\mathcal{Q})$ - and two, to each instance X (of $\mathcal{Q}$), a certificate candidate $C(X)$ that is verified by $AL(\mathcal{Q})$ in polynomial time (in n, the size of X).

1.3. Remarks on Algorithms Recognizing Feasible Solutions

An algorithm is said to *recognize* a feasible solution if the algorithm verifies that the input certificate candidate is a certificate. In the context of ***NP***, such recognition needs to be done in polynomial time.

 doi:10.20944/preprints202510.0617.v1

Let X be an instance of a problem $\mathcal{Q}$ and ALG be the algorithm designed to check if an input certificate candidate is a certificate. Let C_1 and C_2 be two distinct certificate candidates.

Suppose that both C_1 and C_2 are feasible solutions to X. Then it is desirable that ALG recognizes C_1 or C_2, whichever is input with X. However, owing to the design of ALG the following might ensue:
1.3(i) ALG recognizes the feasible solution C_1 in polynomial time and
1.3(ii) ALG fails, or takes worse than polynomial time (perhaps, exponential or factorial time [3,8]), to recognize the feasible solution C_2.

Then C_2, despite being a feasible solution, is not useful in deciding if $\mathcal{Q}$ is in ***NP***. So only candidates that can be checked out by the algorithm in polynomial time should be used. To aver that $\mathcal{Q}$ is in ***NP***, for each instance of $\mathcal{Q}$ we only need one candidate that ALG recognizes as a feasible solution in polynomial time, even if there exist other feasible solutions that ALG does not recognize at all or does not recognize in polynomial time.

1.4. Looking for Certificates

If a problem $\mathcal{Q}$ is given, then it is not difficult to come up with an instance X of a desired size n and a certificate candidate. But the same cannot be said for a certificate, even if an algorithm is available. Then how can a certificate, if at all one exists, be obtained?

Recall the part of the informal explanation of the class ***NP*** that goes '*...if, in addition to an instance of the problem, we are somehow given a "certificate" of a solution, then we could verify...*' (in the second paragraph of Subsection 1.2). The operative word there is 'somehow.' This clearly indicates there are no hard and fast rules as to the source of a certificate candidate or its form or the process of obtaining it; only every certificate candidate must rest on irrefutable theory that is relevant to the problem. Indeed, given an arbitrary problem instance, it is not known how to identify a certificate [5]. So it is not known how to identify a certificate candidate that will turn out to be a certificate. A certificate candidate $C(X)$ could be readily available (i.e., prepared beforehand by someone) or could be compiled as and when the need arises. Preparing $C(X)$ could take any amount of time - polynomial or worse. But the time to prepare $C(X)$ will not be included in the time required for the algorithm to check out $C(X)$. This is because whoever prepares $C(X)$ is allowed to do any immense amount of calculation beforehand, and only the results of that calculation need be written on $C(X)$ [8].

In any quest for finding out whether a problem is in ***NP***, one assumption (on the quester's part) is that given an instance of the problem there is a non-vacuous class of certificate candidates corresponding to this instance.

*1.5. Remarks on **P** and **NP***

The class ***P*** is contained in the class ***NP*** [2,3]. But it is not known whether these two classes are the same - and this is the crux of the famous problem "Is $\boldsymbol{P} = \boldsymbol{NP}$?" (also called the "***P*** versus ***NP***" problem).

One approach to this problem is trying to prove ***NP*** is contained in ***P***; this, if successful, gives $\boldsymbol{P} = \boldsymbol{NP}$. An opposite approach is finding or formulating a problem that falls in ***NP*** but cannot be in ***P*** (thereby establishing $\boldsymbol{P} \neq \boldsymbol{NP}$), even if such a problem is artificially formulated [7].

Any problem in ***NP*** can be solved by exhaustive search (also called *perebor*, meaning "brute-force search" [7]). But steps in exhaustive search grow to forbiddingly large numbers [9] even for a moderate growth in the size of the instance. Though decades of extensive efforts to settle the ***P*** versus ***NP*** question have produced algorithms that take significantly fewer steps than exhaustive search for problems in ***NP***, an exact polynomial-time algorithm for any of these problems is yet to be. As a consequence, it is now commonly believed [9] that $\boldsymbol{P} \neq \boldsymbol{NP}$.

1.6. Atomic Sub-Outputs

Let S be a solution to a problem instance X. Suppose S consists of finitely many definite and distinguishable objects $Y_1, \ldots, Y_k$ (for some $k \in \mathbb{N}$) that are to be computed in a sequence. Then each of these k objects is an *atomic sub-output* of S.

 doi:10.20944/preprints202510.0617.v1

For example, if the problem is to compute the set of all positive integers q less than a given positive integer m such that q is a perfect cube, then the set $S = \{1, 8\}$ is a solution to the instance $m = 20$ of this problem. Each of the two elements of S is an atomic sub-output of S. S is the only solution to the instance $m = 20$. The instance $m = 1$ has no solution.

Now consider a problem $\mathcal{Q}$ such that:

1.6(i) each instance of $\mathcal{Q}$ has a solution and

1.6(ii) each solution to a given instance of $\mathcal{Q}$ (with n being the size of the instance) consists of m^n atomic sub-outputs (to be computed in a sequence) for some $m \in \mathbb{R}$ with $m > 1$.

Any algorithm that outputs a solution to a given instance of $\mathcal{Q}$ has to compute all the m^n atomic sub-outputs that the solution comprises. Then as the instance size n increases, the number of steps taken by an algorithm cannot be bounded above by any polynomial in n (see, for instance, Proposition 2.4 in [4]). Consequently, if $\mathcal{Q}$ is in ***NP*** then it follows immediately that $\boldsymbol{P} \neq \boldsymbol{NP}$. To confirm that $\mathcal{Q}$ is in ***NP***, we need a check-type algorithm - call it $AL(\mathcal{Q})$ - such that to each instance X of $\mathcal{Q}$ there must be obtainable at least one certificate candidate $C(X)$ that is verified by $AL(\mathcal{Q})$, in polynomial time, to yield a solution to X. Such a $C(X)$ can be obtained from anywhere or prepared anyhow but once it is input along with X and n then from that point $AL(\mathcal{Q})$ should need to do only a polynomial amount of computations to verify $C(X)$ [8].

We discuss one such problem and the necessary verifications (using polynomial-time algorithms) in Section 3 through Section 5.

1.7. Remarks on Capabilities of Algorithms

There are computational capabilities algorithms are known to possess, and literature abounds with discussions on such capabilities. And there are capabilities that supposedly cannot be possessed by any algorithm, at least as of now. For instance, to date there is no known algorithm with the capability to carry out exhaustive search in polynomial time for an arbitrary input. Discussing these capabilities is not in the scope of this article. Suffice it to say that facts / suppositions about capabilities of algorithms rest on established theories. These theories may advance and then there may come along algorithms with new / enhanced capabilities. We deal with the ***P*** versus ***NP*** problem in the framework of capabilities of algorithms at present.

In the preceding discussions, only informal definitions of algorithm, ***P*** and ***NP*** have been used. Their formal definitions are in [2,3].

2. Essential Theory

A *set* is a collection of definite and distinguishable objects [3,6]. Each object in a set is an *element* or a *member* of the set. It is taken for granted that if X is a given set then there is a well-formed definition that decides conclusively whether or not two given elements of X are distinct. Also, such a definition is made explicit when necessary. There is a unique set that contains no members, and this is the *empty set*, denoted by ϕ.

The *cardinality* (or, *size*) of a set X is the number of elements in X, and is denoted by $|X|$. Obviously $|X| \geq 0$. If $|X| \in \mathbb{W}$ then X is a *finite set*; else X is an *infinite set*.

A set A is a *subset* of a set B (written $A \subseteq B$) if each element of A is also an element of B. Every set is a subset of itself, and the set ϕ is a subset of every set,

For set union ($\cup$) and set intersection ($\cap$), see [6]. The sets $A_1, \ldots, A_k$ are said to be *pairwise disjoint* if no two among these k sets have a common element. These k sets are said to *partition* the set A if: (i) A_i is a nonempty subset of A for each $i = 1, \ldots, k$, (ii) $A = A_1 \cup \cdots \cup A_k$ and (iii) the sets A_1 through A_k are pairwise disjoint.

For one-one function and onto function, see [6]. Two sets A and B are said to be in *bijective correspondence* if there exists a function [6] $f : A \to B$ such that f is both one-one and onto. If A and B are finite sets that are in bijective correspondence, then $|A| = |B|$.

 doi:10.20944/preprints202510.0617.v1

The set of all the subsets of X, including ϕ and X itself, is the *power set* of X, and is denoted by 2^X. If X is finite with $|X| = n$ then 2^X is finite [6] with $|2^X| = 2^n$. Each of the notations $Y \in 2^X$ and $Y \subseteq X$ will mean Y is a subset of X.

Let X be a nonempty finite set with $|X| = n$. If $\mathcal{C} \subseteq 2^X$ such that $|Y|$ is an even integer for each $Y \in \mathcal{C}$, then $\mathcal{C}$ is an *even-sizes coterie* in 2^X. If $\mathcal{C} \subseteq 2^X$ such that $|Y|$ is an odd integer for each $Y \in \mathcal{C}$, then $\mathcal{C}$ is an *odd-sizes coterie* in 2^X.

The set of all $Y \in 2^X$ such that $|Y|$ is an even integer will be denoted by $\mathcal{C}_e(X)$ and is the *maximum even-sizes coterie* in 2^X. The set of all $Y \in 2^X$ such that $|Y|$ is an odd integer will be denoted by $\mathcal{C}_o(X)$ and is the *maximum odd-sizes coterie* in 2^X.

From these definitions, the following are immediate:
(i) $\mathcal{C}_e(X)$ is the only even-sizes coterie that is maximal in 2^X with respect to set inclusion - i.e., $\mathcal{C}_e(X)$ is not properly contained in any even-sizes coterie in 2^X.
(ii) $\mathcal{C}_o(X)$ is the only odd-sizes coterie that is maximal in 2^X with respect to set inclusion.
(iii) Any even-sizes coterie in 2^X is a subset of $\mathcal{C}_e(X)$.
(iv) Any odd-sizes coterie in 2^X is a subset of $\mathcal{C}_o(X)$.

Proposition 2.1. For any nonempty finite set X:
(i) $\mathcal{C}_e(X) \cap \mathcal{C}_o(X) = \phi$ and
(ii) $2^X = \mathcal{C}_e(X) \cup \mathcal{C}_o(X)$.

Proof. Straightforward. •

Proposition 2.2. Let X be a nonempty finite set with $|X| = n$. Then $|\mathcal{C}_e(X)| = |\mathcal{C}_o(X)| = 2^{n-1}$.

Proof. In the light of Proposition 2.1, it suffices to show $|\mathcal{C}_e(X)| = 2^{n-1}$. The proof will be by induction on n. Let P(n) denote the statement to prove.

Let $n = 1$. Write $X = \{a\}$. Then $2^X = \{\phi, X\}$. Immediately we have $\mathcal{C}_e(X) = \{\phi\}$. Then $|\mathcal{C}_e(X)| = 1 = 2^{n-1}$, proving P(1).

Induction hypothesis. Assume P(r).

Next, let $|X| = r + 1$. Write $X = \{a_1, \ldots, a_{r+1}\}$ and let $Y = X - \{a_{r+1}\}$. Let $\mathcal{M} = \{A \cup \{a_{r+1}\} : A \in \mathcal{C}_o(Y)\}$. Note that (i) $|Y| = r$, (ii) $\mathcal{M} \subseteq \mathcal{C}_e(X)$ and (iii) $\mathcal{C}_e(Y) \subset \mathcal{C}_e(X)$.

We now assert that $\mathcal{C}_e(Y)$ and $\mathcal{M}$ partition $\mathcal{C}_e(X)$.

If $B \in \mathcal{C}_e(Y)$ then $a_{r+1} \notin B$, from which $B \notin \mathcal{M}$, leading to $\mathcal{C}_e(Y) \cap \mathcal{M} = \phi$.

Next, if $B \in \mathcal{C}_e(X)$ then either $a_{r+1} \in B$ or not. If $a_{r+1} \in B$ then $B = A \cup \{a_{r+1}\}$ for some $A \in \mathcal{C}_o(Y)$, giving $B \in \mathcal{M}$. If $a_{r+1} \notin B$ then $B \in \mathcal{C}_e(Y)$, whence $\mathcal{C}_e(X) \subseteq \mathcal{C}_e(Y) \cup \mathcal{M}$.

On the other hand, if $B \in \mathcal{C}_e(Y)$ then clearly $B \in \mathcal{C}_e(X)$; whereas if $B \in \mathcal{M}$ then $B = A \cup \{a_{r+1}\}$ for some $A \in \mathcal{C}_o(Y)$, whence $B \in \mathcal{C}_e(X)$. Consequently, $\mathcal{C}_e(X) = \mathcal{C}_e(Y) \cup \mathcal{M}$ is a partitioning of $\mathcal{C}_e(X)$.

Over to $|\mathcal{C}_e(X)|$. By the above partitioning, we have $|\mathcal{C}_e(X)| = |\mathcal{C}_e(Y)| + |\mathcal{M}|$. By the induction hypothesis (applied to Y), we have $|\mathcal{C}_o(Y)| = |\mathcal{C}_e(Y)| = 2^{r-1}$. Further, $\mathcal{M}$ is in bijective correspondence [6] with $\mathcal{C}_o(Y)$, giving $|\mathcal{M}| = 2^{r-1}$. Consequently, $|\mathcal{C}_e(X)| = |\mathcal{C}_e(Y)| + |\mathcal{M}| = 2^r$, completing the induction.•

3. Problem Variants

A *variant* of a problem $\mathcal{Q}$ is a formulation of $\mathcal{Q}$ that seeks a desired type of solution to a given instance of $\mathcal{Q}$. Types of variants that are widely studied and used are: optimization, computation and decision.

An *optimization variant* of $\mathcal{Q}$ is a formulation of Q that asks for a solution of an optimum measure (which is either the maximum or the minimum of the concerned measure) to each instance of $\mathcal{Q}$ [1].

A *computation variant* of $\mathcal{Q}$ is a formulation that asks for a solution (to each instance of $\mathcal{Q}$) subject to finitely many conditions.

 doi:10.20944/preprints202510.0617.v1

A *decision variant* of $\mathcal{Q}$ is a formulation of $\mathcal{Q}$ using a decision question that asks for either a "yes" or a "no" answer to each instance. The basic ingredients [1] of a decision variant are: the set of instances, the set of attempted solutions (i.e., certificate candidates [4**?**]) and the predicate that decides whether an input certificate candidate yields a solution.

We shall be concerned only with optimization and decision variants of the problem we discuss in the coming sections. It is common to formulate an optimization problem $\mathcal{Q}$ as a decision problem to find out if $\mathcal{Q}$ is in *NP*.

3.1. Optimization Variant of a Problem in Bifurcating Power Sets

The following is a computation variant of the problem we name ES-COT:
If X is a nonempty finite set with $|X| = n$, then find the maximum even-sizes coterie in 2^X.

3.2. A decision Variant of ES-COT

Inputs: (i) A nonempty finite set X, (ii) $n = |X|$ and (iii) $k \in \mathbb{W}$.

Question: Does there exist $\mathcal{C} \subseteq \mathcal{C}_e(X)$ such that $|\mathcal{C}| = 2^k$?

Output: YES (a desired set $\mathcal{C}$ exists) or **NO** (no such set $\mathcal{C}$ exists), as appropriate.

Certificate candidate: $C(X) = k$.

Note. It is mandatory tha the inputs (i) through (iii) and the certificate candidate be free from any error, as also that they be logically consistent with one another.

In Section 4 we outline an algorithm towards showing that the problem ES-COT is in ***NP***.

4. Algorithm ES-COT-in-*NP*

The following algorithm will be referred to as ES-COT-in-***NP***. The input is (X, n, k). X, n and k, as well as the decision question and the required output, are given in Subsection 3.2.

Algorithm ES-COT-in-***NP***

BEGIN
1. **if** $0 \leq k \leq n-1$
2. **then** print "YES, there exists $\mathcal{C} \subseteq \mathcal{C}_e(X)$ such that $|\mathcal{C}| = 2^k$" and STOP
3. **else** print "NO, no such $\mathcal{C}$ exists" and STOP
4. **endif**

STOP

Proposition 4.1. Let X be a nonempty finite set with $|X| = n$. Let $k \in \mathbb{W}$.
(i) If $0 \leq k \leq n-1$ then there exists $\mathcal{C} \subseteq \mathcal{C}_e(X)$ such that $|\mathcal{C}| = 2^k$.
(ii) If $k > n-1$ then there is no set $\mathcal{C} \subseteq \mathcal{C}_e(X)$ such that $|\mathcal{C}| = 2^k$.

Proof. (i) Consequence of Proposition 2.2.
(ii) By Proposition 2.2, $|\mathcal{C}| \leq 2^{n-1}$ for each $\mathcal{C} \subseteq \mathcal{C}_e(X)$, from which the conclusion follows immediately.
•

Proposition 4.2. The algorithm ES-COT-in-***NP*** is feasible (i.e., terminates in a finite number of steps) and correct.

 doi:10.20944/preprints202510.0617.v1

Proof. The certificate candidate is $C(X) = k \in \mathbb{W}$. The algorithm makes decisions based on whether or not $0 \leq k \leq n-1$ (line 1). This check comprises the checks $0 \leq k$ and $k \leq n-1$, each of which clearly terminates in a finite number of steps.

Each output returned by the algorithm is either YES or NO. The possible outputs are all accounted for in two lines of the algorithm - namely, lines 2 and 3. So the algorithm returns only finitely many outputs. Printing each decision clearly terminates in a finite number of steps.

Consequently, ES-COT-in-***NP*** is feasible. Next, we prove its correctness.

If $0 \leq k \leq n-1$ is true then the algorithm decides YES (line 2), which is the correct output by Proposition 4.1(i). If $0 \leq k \leq n-1$ is false then the algorithm decides NO (line 3). This is correct by Proposition 4.1(ii). •

Proposition 4.3. Given an input (X, n, k), ES-COT-in-***NP*** runs in polynomial time in n.

Proof. The total number (say, T) of steps executed by the algorithm ES-COT-in-***NP*** is the sum of the numbers of steps for all the lines executed. Suppose that one execution of the line j requires t_j steps and that this line is executed exactly r_j times when the algorithm is executed once. Then $t_j r_j$ is the number of steps consumed by the line j in one execution of the algorithm.

In one execution of the algorithm, each line is executed once if at all. Hence, for $j = 1, \ldots, 4$, $t_j r_j = t_j$.

We suppose each **endif** line takes constant time, independent of n.

The number of steps required for line 1 is at most n^2. Likewise for lines 2 and 3. Hence $T = 3n^2 + 1$.•

Proposition 4.4. To each instance X of the problem ES-COT, there is a certificate candidate that is verified by the algorithm ES-COT-in-***NP*** in polynomial time in the size ($n = |X|$) of X.

Proof. The required certificate is $k \in \mathbb{W}$ where $0 \leq k \leq n-1$. •

5. *NP*, *P* and the Problem ES-COT

Proposition 5.1. 2^{n-1} is unbounded as n increases over $\mathbb{N}$.

Proof. Straightforward.•

Proposition 5.2. The problem ES-COT is in ***NP***.

Proof. Consequence of Proposition 4.2 through Proposition 4.4. •

Proposition 5.3. The problem ES-COT is not in ***P***.

Proof. Suppose $\mathcal{A}$ is a given feasible algorithm that outputs the maximum even-sizes coterie $\mathcal{C}_e(X)$ of the input instance X (where X is a nonempty finite set of cardinality n). Each of the 2^{n-1} members of $\mathcal{C}_e(X)$ is an atomic sub-output of $\mathcal{C}_e(X)$. Name these sub-outputs $S_1, \ldots, S_q$ in the order that $\mathcal{A}$ follows in computing the members of $\mathcal{C}_e(X)$. By Proposition 2.2, $q = 2^{n-1}$.

For $j = 1, \ldots, q-1$, having taken t_j steps for only the computation of S_j, suppose $\mathcal{A}$ takes another t_{j+1} steps to compute S_{j+1}; in other words, once $\mathcal{A}$ executes t_j steps to compute S_j then beginning with the next step, $\mathcal{A}$ executes t_{j+1} steps to compute S_{j+1}, allowing that any of the already-computed sub-outputs S_1 through S_j may be used anywhere in the computation of S_{j+1}. Obviously, then, each $t_j \geq 1$ and $t_q \geq 1$.

 doi:10.20944/preprints202510.0617.v1

If $T(X)$ is the total number of steps taken by $\mathcal{A}$ to compute and output the members S_1 through S_q of $C_e(X)$, then $T(X) \geq t_1 + \cdots + t_q \geq q = 2^{n-1}$. By Proposition 5.1, $\mathcal{A}$ cannot run in polynomial time (in n), from which the conclusion follows. •

6. Conclusion

$\boldsymbol{P} \neq \boldsymbol{NP}$ follows from Proposition 5.2 and Proposition 5.3.

There is a caveat, though. Finding a solution to a given instance of the optimization variant of ES-COT requires computations that take exponential number of steps, as can be gauged from the proof of Proposition 5.3, leading to a thought that no algorithm is likely to possess the capability to compute exponential (or worse) number of atomic sub-outputs of the solution to any instance of ES-COT in polynomial time. But what if underlying theories get advanced sufficiently so that algorithms with this capability are designed? Would it imply that exhaustive search could be done in polynomial time? Then would $\boldsymbol{P} = \boldsymbol{NP}$ ensue? This is a moot point. However, at present, surveys ([9], for one) seem to favor the opinion that no algorithm is ever likely to have such a capability. This seems to justify the conclusion above.

Acknowledgements

This research was fully funded by Niels Abel Foundation (NAF), Palakkad, Kerala State, India. The authors are especially grateful to everyone of the referees – all anonymous – who returned highly favourable reports about the correctness of the proofs in this article, as a result of which NAF gave the authors substantial financial rewards.

References

1. Bovet, D. P. and Crescenzi, P., 1994, Introduction to the theory of complexity, Prentice Hall, London, UK.
2. Cook, S., 2000, The P versus NP problem, Available online: http://www.claymath.org/millennium/P vs NP/pvsNP.pdf.
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., 2009, Introduction to Algorithms, MIT Press, USA.
4. Dharmarajan, R. and Ramachandran, D., A problem in Moon-Moser graphs to show $\boldsymbol{P}$ does not equal $\boldsymbol{NP}$, *Glob. J. Pure Appl. Math.* **21** (1) (2025) 51-73.
5. Savage, J. E., 1998, Models of Computation, Addison Wesley, Reading, USA.
6. Stoll, R. R., 2012, Set Theory and Logic, Courier Corporation, USA.
7. Trakhtenbrot, B. A., 1984, "A Survey of Russian Approaches to Perebor (Brute-Force Search) Algorithms," IEEE Ann. Hist. Comput. 6(4), pp. 384-400.
8. Wilf, H. S., 1994, Algorithms and Complexity, Internet Edition, Summer.
9. Woeginger, G. J., 2003, "Exact algorithms for NP-hard problems: A survey," Combinatorial Optimization - Proc. 5th Intl. Workshop, Aussois, France, pp. 185–207.