Article

Not peer-reviewed version

A Problem in Power Sets Shows P Does Not Equal NP

Dharmarajan R * and Ramachandran D

Posted Date: 18 March 2025

doi: 10.20944/preprints202503.1321.v1

Keywords: Algorithm; polynomial time; certi cate; power set

Article

A Problem in Power Sets Shows P Does Not Equal NP

R. Dharmarajan * and D. Ramachandran

Niels Abel Foundation, Palakkad 678011, Kerala, India

* Correspondence: mathnafrd@gmail.com

Citing this article:

R. Dharmarajan and D. Ramachandran, *A problem in power sets shows P does not equal NP*, Research Report 2 of Niels Abel Foundation, March 2025, NAF Publications, Palakkad, India.

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. In this article we show **P ≠ NP** using power sets.

Keywords: algorithm; polynomial time; certificate; power set

Mathematics Subject Classification 2020: 03C70; 11Y16; 68W40

---

1. Introduction

More on terminologies, symbols and notations used in this article can be found in [3,5,7]. Throughout this article, $\mathbb{N}$ will denote the set of positive integers, $\mathbb{W}$ the set of non-negative integers (i.e., $\mathbb{W} = \mathbb{N} \cup 0$), $\mathbb{Z}$ the set of integers and $\mathbb{R}$ the set of real numbers.

This article is organised as follows: Section 1 deals with some salient points on algorithms, the problem classes **P** and **NP,** proposed solutions and feasible solutions. Section 2 presents the power set problem and a decision version of it. Section 3 deals with an algorithm that is instrumental in proving **P ≠ NP**. Section 4 outlines proof that the power set problem is in **NP** but not in **P**. The concluding remarks are in Section 5.

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 $\boldsymbol{Q}$ being studied [3]. For example, if $\boldsymbol{Q}$ 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/preprints202503.1321.v1

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

From a computational point of view, given a problem $\boldsymbol{Q}$, it is natural to ask if there exists, or can be developed, an algorithm that can process any given instance of $\boldsymbol{Q}$ 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-á-vis a specified problem $\boldsymbol{Q}$; in other words, the phrase 'the algorithm ALG runs in polynomial time' really means that there is a specified problem $\boldsymbol{Q}$ such that ALG processes each instance of $\boldsymbol{Q}$ 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 $\boldsymbol{Q}$ (meaning, in turn, that the instance X of $\boldsymbol{Q}$ can vary but $f(n)$ does not, so long as the problem $\boldsymbol{Q}$ does not) such that given two instances (of $\boldsymbol{Q}$) 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 that might confirm the existence of a solution to the input instance. This additional data accompanying the instance can be thought of as an "attempt at a solution," and it may 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 also an input component. Also, a certificate only begins as an attempted solution, and might or might not turn out to be a feasible solution [6]. 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 $\boldsymbol{Q}$ 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 attempt at a solution actually turns out to be a feasible solution); in this case $C(X)$ is a *certificate of a solution*.

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 attempt at a solution, though not a

 doi:10.20944/preprints202503.1321.v1

feasible solution by itself, confirms the existence of a feasible solution); in this case *C(X)* is a *certificate for a solution*.

1.2(iii) Neither is *C(X)* a solution nor does it logically imply the existence of a solution; then *C(X)* is a neither a certificate of a solution nor a certificate for 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 call *C(X)* a *certificate*. 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 attempt at a 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 are special cases of certificate candidates.

As noted in the penultimate paragraph of Subsection 1.1, an algorithm is said to solve an instance *X* of a problem $\boldsymbol{Q}$ 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 $\boldsymbol{Q}$ then the algorithm is said to solve $\boldsymbol{Q}$. 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 both 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 $\boldsymbol{Q}$, what are needed to decide that $\boldsymbol{Q}$ can be included in **NP**? We need: one, a check-type algorithm - call it AL($\boldsymbol{Q}$) - and two, to each instance *X* (of $\boldsymbol{Q}$), a certificate candidate *C(X)* that is verified by AL($\boldsymbol{Q}$) 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, for some certificate candidate, the algorithm verifies that the certificate candidate is a certificate. In the context of **NP**, such recognition needs to be done in polynomial time.

 doi:10.20944/preprints202503.1321.v1

Let X be an instance of a problem $\boldsymbol{Q}$ and ALG be the algorithm designed to check if an input certificate candidate is a certificate. Suppose C_1 and C_2 are two distinct certificate candidates. Also suppose that both C_1 and C_2 are feasible solutions to X.

It is desirable that ALG checks out 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 [3,9]), to recognize the feasible solution C_2.

Then C_2, despite being a feasible solution, is not useful in deciding if $\boldsymbol{Q}$ is in **NP**. So only candidates that can be checked out by the algorithm in polynomial time should be used. To aver that $\boldsymbol{Q}$ is in **NP**, for each instance of $\boldsymbol{Q}$ 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. The Exercise of Looking for Certificates

If a problem $\boldsymbol{Q}$ 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 [6]. 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 $C(X)$ [9].

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 **P** = **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 **P** = **NP**. An opposite approach is finding or formulating a problem that falls in **NP** but cannot be in **P** (thereby establishing **P** ≠ **NP**), even if such a problem is artificially formulated [8].

Any problem in **NP** can be solved by exhaustive search (also called *perebor*, meaning "brute-force search" [8]). But steps in exhaustive search grow to forbiddingly large numbers 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 that **P** ≠ **NP** [10].

1.6. Atomic Sub-Outputs

 doi:10.20944/preprints202503.1321.v1

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.

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 $\boldsymbol{Q}$ such that:

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

1.6(ii) each solution to a given instance (of size n) of $\boldsymbol{Q}$ 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 $\boldsymbol{Q}$ 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 Proposition 2.4 in Section 2). Consequently, if $\boldsymbol{Q}$ is in **NP** then it follows immediately that **P** ≠ **NP**. To confirm that $\boldsymbol{Q}$ is in **NP**, we need a check-type algorithm - call it AL($\boldsymbol{Q}$) - such that to each instance X of $\boldsymbol{Q}$ there must be obtainable at least one certificate candidate $C(X)$ that is verified by AL($\boldsymbol{Q}$), 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($\boldsymbol{Q}$) should need to do only a polynomial amount of computations to verify $C(X)$.

We discuss such a problem and the necessary verifications (using polynomial-time algorithms) in Section 2 through Section 4. The problem is centred on the power set of a nonempty finite set.

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 have dealt 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. The Power Set Problem for Nonempty Finite Sets

A *set* is a collection of definite and distinguishable objects [3,7]. 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*.

The set of all the subsets of a set X, including the empty set ϕ and the set X itself, is denoted by $\mathcal{P}(X)$ and is the *power set* of X. If X is finite of cardinality n then $\mathcal{P}(X)$ is finite [3,7] of cardinality 2^n.

A *variant* of a problem Q is a formulation of Q that seeks a desired type of solution without altering the import of Q. Types of variants that are widely studied and used are: *optimization*, *computation* and *decision*.

An *optimization variant* of $\boldsymbol{Q}$ is a formulation of $\boldsymbol{Q}$ 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 $\boldsymbol{Q}$ [1].

 doi:10.20944/preprints202503.1321.v1

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.

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

The power set problem for nonempty finite sets is: *given a nonempty finite set X, find* $\mathcal{P}(X)$. This statement also represents an optimization variant of the problem. Here the measure considered is cardinality of sets, and the optimum measure is the maximum cardinality.

We name this problem $\mathcal{P}$(FIN). Any instance of $\mathcal{P}$(FIN) is a nonempty finite set X, and the input size of this instance is $|X|$. If some subset $\mathcal{L}$ of $\mathcal{P}(X)$ has been output, then each member of $\mathcal{L}$ is an atomic sub-output of $\mathcal{L}$.

In the context of the **P** versus **NP** problem, we shall be concerned with optimization and decision variants only. It is common to formulate an optimization problem $\mathcal{Q}$ as a decision problem to find out if $\mathcal{Q}$ is in **NP**.

2.1. Decision Variant of $\mathcal{P}$(FIN)

The following is a decision variant of $\mathcal{P}$(FIN).

Inputs: 2.1(i) A nonempty finite set X (the problem instance),

2.1(ii) $n = |X|$ (the size of the instance) and

2.1(iii) $k \in \mathbb{Z}$.

Question: Is there a set $\mathcal{L}$ such that $\mathcal{L} \subseteq \mathcal{P}(X)$ with $|\mathcal{L}| = 2^k$?

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

Output: YES (meaning, there a set $\mathcal{L}$ such that $\mathcal{L} \subseteq \mathcal{P}(X)$ with $|\mathcal{L}| = 2^k$) or NO (meaning, there is no such set $\mathcal{L}$) as appropriate.

Note. It is mandatory that the inputs 2.1(i) through 2.1(iii) and $C(X)$ be free from any error, as also that they be logically consistent with one another.

In Section 3, we outline an algorithm (in pseudocodes) that we name CHECKSUBSET and prove what are required to verify that $\mathcal{P}$(FIN) is in **NP**.

3. Algorithm CHECKSUBSET

The following algorithm will be referred to as CHECKSUBSET. The input is $(X, n, k, C(X))$. X, n, k and $C(X)$, as well as the decision question and the required output, are given in Subsection 2.1.

Decision problem question: Is there a set $\mathcal{L}$ such that $\mathcal{L} \subseteq \mathcal{P}(X)$ with $|\mathcal{L}| = 2^k$? (Here $k = C(X)$.)

Algorithm CHECKSUBSET

BEGIN

1. **if** $0 \le k \le n$
2. **then** print "YES, exists $\mathcal{L} \subseteq \mathcal{P}(X)$ with $|\mathcal{L}| = 2^k$ since $0 \le k \le n$" and STOP
3. **else** print "NO. No set $\mathcal{L} \subseteq \mathcal{P}(X)$ can have $|\mathcal{L}| = 2^k$ since $k < 0$ or $k > n$" and STOP
4. **endif**

STOP

Proposition 3.1. Let $C(X) = k \in \mathbb{Z}$ be the certificate candidate corresponding to the input instance X in the algorithm CHECKSUBSET.

(i) If $0 \le k \le n$ then $|\mathcal{L}| = 2^k$ for some $\mathcal{L} \subseteq \mathcal{P}(X)$.

(ii) If $k < 0$ or $k > n$ then there is no set $\mathcal{L} \subseteq \mathcal{P}(X)$ such that $|\mathcal{L}| = 2^k$.

Proof. (i) Write $X = \{x_1, \ldots, x_n\}$. Let $A = \{x_1, \ldots, x_k\}$ and $\mathcal{L} = \mathcal{P}(A)$. Then $\mathcal{L} \subseteq \mathcal{P}(X)$ and $|\mathcal{L}| = 2^k$.

 doi:10.20944/preprints202503.1321.v1

(ii) Suppose $k < 0$. Then no set $\mathcal{L}$ can satisfy $|\mathcal{L}| = 2^k$. If $k > n$ then $|\mathcal{L}| \leq 2^n < 2^k$ for every set $\mathcal{L}$ such that $\mathcal{L} \subseteq \mathcal{P}(X)$, from which the conclusion follows. ■

Proposition 3.2. The algorithm CHECKSUBSET is feasible and correct.

Proof. The certificate candidate is $C(X) = k \in \mathbb{Z}$. The algorithm makes decisions based on whether or not $0 \leq k \leq n$ (line 1). This check comprises the checks $0 \leq k$ and $k \leq n$, 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, CHECKSUBSET is feasible. Next, we prove its correctness.

If $0 \leq k \leq n$ is true then the algorithm decides YES (line 2), which is the correct output by Proposition 3.1(i).

If $0 \leq k \leq n$ is false then the algorithm decides NO (line 3). This is the correct output by Proposition 3.1(ii). ■

Proposition 3.3. Given an input $(X, n, k, C(X))$, the algorithm CHECKSUBSET runs in polynomial time in n.

Proof. The total number (say, T) of steps executed by the algorithm CHECKSUBSET is the sum of the numbers of steps for all the lines executed.

Suppose one execution of the line j of the algorithm CHECKSUBSET requires s_j steps and that this line is executed exactly q_j times. Then $s_j q_j$ is the number of steps taken by the line j in the execution of the algorithm.

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

We suppose that each **endif** line takes one step, independent of n.

Line 1 comprises the checks $0 \leq k$ and $k \leq n$. The number of steps required for each of these checks is bounded above by n^2. Likewise for line 2 as well as line 3. Consequently, $T \leq 4n^2 + 1$. ■

Proposition 3.4. To each instance X of $\mathcal{P}$(FIN) there is a certificate that is verified by CHECKSUBSET in polynomial time in the size (n) of X.

Proof. $C(X) = k \in \mathbb{Z}$ where $0 \leq k \leq n$ is such a certificate. ■

4. $\mathcal{P}$(. FIN), the Class NP and the Class P

Proposition 4.1. $\mathcal{P}$(FIN) is in NP.

Proof. Let X be a given instance of $\mathcal{P}$(FIN) with $|X| = n$. The next requirements are a check-type algorithm and a certificate candidate that is verified in polynomial time by this algorithm to confirm the existence of a solution to X.

The required algorithm is CHECKSUBSET (Section 3) and an appropriate certificate candidate is $C(X) = k \in \mathbb{Z}$ with $0 \leq k \leq n$ (Proposition 3.3 and Proposition 3.4). ■

Proposition 4.2. $\mathcal{P}$(FIN) is in not in **P**.

Proof. Let ALG$\mathcal{P}$(FIN) be any algorithm that solves $\mathcal{P}$(FIN). Given an instance X with $|X| = n$, the solution to this instance is $\mathcal{P}(X)$, and each of the 2^n distinct members of $\mathcal{P}(X)$ is an atomic sub-

 doi:10.20944/preprints202503.1321.v1

output of this solution. Name these sub-outputs $S_1, \ldots, S_q$ in the order that ALG$\mathcal{P}$(FIN) follows in the process of solving X. Clearly $q = 2^n$.

For $j = 1, \ldots, q-1$, having taken s_j steps for only the computation of S_j, suppose ALG$\mathcal{P}$(FIN) takes another s_{j+1} steps to compute S_{j+1}; in other words, once ALG$\mathcal{P}$(FIN) executes s_j steps to compute S_j then beginning with the next step ALG$\mathcal{P}$(FIN) executes s_{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 $s_j \geq 1$ and $s_q \geq 1$.

If $T(X)$ is the total number of steps taken by ALG$\mathcal{P}$(FIN) to compute the subsets S_1 through S_q, then $T(X) \geq s_1 + \ldots + s_q \geq q = 2^n$. Consequently, ALG$\mathcal{P}$(FIN) cannot run in polynomial time. ■

5. Conclusion

P ≠ NP follows from Proposition 4.1 and Proposition 4.2.

There is a caveat, though. The optimization variant of the problem $\mathcal{P}$ (FIN) requires computations that take exponential number of steps, as can be gauged from the proof of Proposition 4.2. This leads 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 either problem in polynomial time. But what if the underlying theory gets advanced sufficiently so that algorithms with this capability are designed? Would it imply that exhaustive search could be done in polynomial time? Then would **P = NP** ensue? This is a moot point. However, at present, surveys ([10], for instance) seem to favour the opinion that no algorithm is ever likely to have such a capability. This seems to justify the conclusion above.

Acknowledgement: 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. It is because of these reports that 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. R. Dharmarajan and D. Ramachandran, *A problem in Moon-Moser graphs to show P does not equal NP*, Research Report 1 of Niels Abel Foundation, February 2025, NAF Publications, Palakkad, India.
5. Harris, J. M., Hirst, J. L. and Mossinghoff, M. J., 2008, Combinatorics and Graph Theory, Springer, New York, USA. Doi: 10.1007/978-0-387-79711-3.
6. Savage, J. E., 1998, Models of Computation, Addison Wesley, Reading, USA.
7. Stoll, R. R., 2012, Set Theory and Logic, Courier Corporation, USA.
8. Trakhtenbrot, B. A., 1984, "A Survey of Russian Approaches to Perebor (Brute-Force Search) Algorithms," IEEE Ann. Hist. Comput. **6**(4), pp. 384--400.
9. Wilf, H. S., 1994, Algorithms and Complexity, Internet Edition, Summer.
10. Woeginger, G. J., 2003, "Exact algorithms for NP-hard problems: A survey," Combinatorial Optimization - Proc. 5th Intl. Workshop, Aussois, France, pp. 185--207.