CYBERNETICS AND PHYSICS, VOL. 1, NO. 4, 2012 , 274–278

A SIMPLE PHYSICS-MOTIVATED EQUIVALENT REFORMULATION OF P=NP THAT MAKES THIS EQUALITY (SLIGHTLY) MORE PLAUSIBLE

Jaime Nava and Vladik Kreinovich
Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968, USA
contact email vladik@utep.edu

Abstract

In our opinion, one of the reasons why the problem P$\stackrel{?}{=}$NP is so difficult is that while there are good intuitive arguments in favor of P$\neq$NP, there is a lack of intuitive arguments in favor of P=NP. In this paper, we provide such an argument — based on the fact that in physics, many dependencies are scale-invariant, their expression does not change if we simply change the unit in which we measure the corresponding input quantity (e. g., replace meters by centimeters). It is reasonable to imagine similar behavior for time complexity $t_A(n)$ of algorithms A: that the form of this dependence does not change if we change the unit in which we measure the input length (e. g., from bits to bytes). One can then easily prove that the existence of such scale-invariant algorithms for solving, e. g., propositional satisfiability is equivalent to P=NP. This equivalent reformulation of the formula P=NP is, in our opinion, much more intuitively reasonable than the original formulation — at least to those who are familiar with the importance of scale-invariance in physics.

Key words

P$\stackrel{?}{=}$NP, symmetry, physical arguments.

1 P$\stackrel{?}{=}$NP Problem: A Brief Reminder

1.1 Some Algorithms Are Feasible, Some Are Not: On the Example of Propositional Satisfiability

Until the 1960s, the main emphasis in computer science was on deciding which problems are algorithmically solvable and which problems are not. Some of the resulting algorithms turned to be feasible and practically useful, other algorithms required too many computational steps to be implemented on the existing computers — but since the computers became faster and faster, there was a hope that these algorithms would become feasible in a few decades.

In the last 1960s, it became clear that some algorithms are not feasible — because even for reasonable size data, these algorithms require computation time which exceeds the lifetime of the Universe. Probably the most well-known example of such an algorithm is an exhaustive search approach to solving the *propositional satisfiability problem* (SAT). Let us briefly recall this problem and the corresponding algorithm.

The propositional satisfiability problem is related to the fact that programs are ubiquitous, they control many important aspects of our lives: program control planes in flight, programs control nuclear power plants, programs control medical devices supporting live. To avoid disasters, it is very important to make sure that these programs work correctly. Ideally, it is desirable to *prove* the program correctness, but for many complex programs, such a proof is still not possible. In such situations, to make sure that the program works correctly, we need at least to *test* it on different inputs. In particular, if a program involves branching, i. e., if it follows different paths depending on some condition, then we need to make that all the branches work correctly.

We can have simple conditions: e.g., relations like $a = b$ or $a < b$, or the value of some propositional (Boolean) variable v, i. e., a variable which can take only values "true" or "false". We can also combine different simple conditions by using propositional connectives "and" ($\&$), "or" ($\vee$), and "not" ($\neg$). So, in general, a condition can be described as a *propositional formula*, i. e., an expression obtained from propositional variables $v_1, v_2, \ldots$ by using $\&$, $\vee$, and $\neg$. For example, we can have a formula

$$(v_1 \vee v_2 \vee \neg v_3)\,\&\,(\neg v_1 \vee \neg v_2).$$

To design a test case in which this condition is satisfied, we need to be able, given a propositional formula, find values of the variables v_i which make this formula

true. This problem is called *propositional satisfiability problem*.

For each formula with n variables, each of these variables v_i has exactly two possible values ("true" and "false"), so there are 2^n possible combinations of truth values $(v_1, v_2, \ldots)$. In principle, we can thus algorithmically solve the propositional satisfiability problem by testing all 2^n possible combinations. The problem with this approach is that already for a reasonable size inputs $n \approx 300$, we need $2^n \approx 10^{100}$ computational steps — while if we divide the lifetime of the Universe ($\approx 10^{10}$ years) by the smallest possible time (during which light passes through an elementary particle), we will get only $\approx 10^{40}$ moments of time. This exhaustive search algorithm is clearly not feasible.

1.2 Feasible vs. Non-Feasible Algorithms: Towards a Formal Definition

How can we formally separate feasible and non-feasible algorithms? In most cases, algorithms A whose computation time $t_A(n)$ is bounded by a polynomial $P(n)$ of the (bit) length n of the input are feasible, while algorithms whose computation time grows faster than any polynomial are not feasible. As a result, in theoretical computer science, an algorithm A is called *feasible* if its computation time is bounded from above by some polynomial.

This definition is not perfect: e.g., computation time $t_A(n) = 10^{100} \cdot n$ is polynomial but clearly not feasible, while the computation time $\exp(10^{-6} \cdot n)$ is clearly not polynomial but feasible for all reasonable lengths n. However, this is the best available definition of a feasible algorithm; see, e. g., [Kreinovich *et al.*, 1998; Papadimitriou, 1994].

1.3 From Abstract Computational Devices to Real-Life Computers

In theoretical computer science, computation time is usually defined as the number of elementary computation steps on an abstract computational device — such as Turing machine (a favorite of theoreticians), Random Access Machine (RAM, a model which is closer to real physical computers), or Kolmogorov-Uspensky machine (which also has robotic abilities). The number of steps often depends on what computation model is used: for example, for most algorithms, the number of steps on a RAM (where we can access each memory cell in a single step) is much smaller than on a Turing machine (where we have to pass through each intermediate memory cell).

It turns out, however, that while the computation time changes, polynomial time remains polynomial, and non-polynomial time remains non-polynomial. In other words, whether an algorithm is feasible or not (in the above formal sense) does not depend on the specific computational device: we can use simple Turing machines, we can use sophisticated computers, the class of feasible algorithms remains the same.

1.4 What Is a "Problem"?

A natural next question is: which problems can be solved by feasible algorithms? To formalize this question, we need to formalize what is a *problem*.

For example, in *mathematics*, a typical problem is: given a statement x, find a proof y of either this statement or of its negation $\neg x$. Once a candidate proof is given in all the formal details, checking whether y is indeed a proof is easy: it is sufficient to check, step by step, whether all derivations are legitimate. Computer algorithms for checking the corresponding property $C(x, y)$ (that y is a formal proof of x) were known already in the 1960s. However, as every mathematician knows, coming us with such a proof y is often not easy.

An additional requirement is that the proof should be of reasonable length, so that it will be possible to check its correctness. Similarly to computation time, a reasonable formalization of "reasonable length" is the length $\mathrm{len}(y)$ which is bounded by a polynomial P_l of the length of the input: $\mathrm{len}(y) \le P_l(\mathrm{len}(x))$. In these terms, a typical problem of mathematics take the following form:

- we have a feasible algorithm $C(x, y)$ and a polynomial $P_l(n)$;
- given a sequence of symbols x, we must find a sequence y for which $C(x, y)$ is true and for which $\mathrm{len}(y) \le P_l(\mathrm{len}(x))$.

In *physics*, given data x, we need to find formulas y that explain all this data. For example, we have pairs (I, V) consisting of current I and voltage V, and we would like to recover Ohm's law. Once a formula y is given, checking whether this formula is indeed consistent with all the observations is feasible: we just check that each of the observations is consistent with this formula. In other words, the property $C(x, y)$ — that the formula y indeed explains all the observations — is feasible. What is often not easy is coming up with a simple formula that would explain all the observations.

The length of the desired formula y cannot exceed the length of the inputs, since otherwise, we can simply list all observations x and call it an explanation. Thus, we must have $\mathrm{len}(y) < \mathrm{len}(x)$, i. e., $\mathrm{len}(y) \le P_l(\mathrm{len}(x))$ for the simple polynomial $P_l(n) = n$. So, we also have the problem of the above type.

In *engineering*, given specifications x (e. g., specifications for a bridge), we must find a design y — a collection of symbols and pictures — that satisfies these specifications, e. g., that enables the bridge to withstand given winds, given loads, and be within the given budget. Here also, once a design y is given, known feasible algorithms can usually check whether the design meets the specifications, but coming up with such a design is often not easy. In this example, the length of the (computer representation of the) design should also be reasonable, otherwise we will not be able to implement this design. So, if we interpret "reasonable length" as bounded by a polynomial $P_l(\mathrm{len}(x))$, we also end up with the problem of the above type.

In general, in many application areas, we have problems for which, once we have a candidate for a solution, it is feasible to check whether this candidate is indeed a solution – but coming up with a solution may not be easy. All these problems have the above type:

- we have a feasible algorithm $C(x, y)$ and a polynomial $P_l(n)$;
- given a sequence of symbols x, we must find a sequence y for which $C(x, y)$ is true and for which $\text{len}(y) \le P_l(\text{len}(x))$.

The class of all such problems is known as the class *NP* [Kreinovich *et al.*, 1998; Papadimitriou, 1994].

1.5 P$\stackrel{?}{=}$NP

For some problems from the class NP, there exist known feasible algorithms that solve all instances of this problem, i.e., algorithms that, given x, generate y for which $C(x, y)$ is true and $\text{len}(y) \le P_l(\text{len}(x))$ (or return a message that such y is impossible). The class of all such feasibly solvable problem is denoted by P (for *P*olynomial time). A natural question is: are all problems from the class NP feasibly solvable? In other words, is the class NP equal to P?

If NP is equal to P, then in mathematics, we would have a feasible algorithm that, given a formula x, generates a proof of either this formula or its negation (or a message that such a proof is impossible). In physics, we would have a feasible algorithm that, given data x, generates a simple formula that explains this data. In engineering, we would have a feasible algorithm that, given bridge specifications (wind, load, cost, etc.), would generate a design for this bridge – or a message that such a design is not possible within given constraints.

This may sound impossible at first glance, but there exist non-trivial examples of polynomial-time algorithms that solve complex problems for which such algorithms were originally thought to be impossible. For example, there is a polynomial-time algorithm for solving linear programming problems, i. e., for checking whether a given system of linear inequalities $\sum_{j=1}^{n} a_{ij} \cdot x_i \ge b_i$, $i = 1, \ldots, m$, is consistent (see, e. g., [Papadimitriou, 1994] and references therein).

1.6 NP-Hard Problems (Including Satisfiability)

While it is still not known whether NP is equal to P, what *is* known is that some problems from the class NP are provably hardest, in the sense that the solution to any other problem from this class can be reduced, in polynomial time, to the solution of this problem.

Historically the first of such *NP-hard* problems is the above-described propositional satisfiability problem (SAT): if we can solve this problem is polynomial time, then we can solve *any* problem from the class NP in polynomial time (i. e., then P=NP). So, the problem P$\stackrel{?}{=}$NP is equivalent to the problem of whether we can solve SAT (or any other NP-hard problem) in polynomial time.

2 Need For a Better Intuitive Understanding of the P=NP Option

In history of mathematics, solutions to many long-standing problems came when the consequences of the corresponding statements being true or false became clearer. For example, mathematicians have tried, for many centuries, to deduce the V-th Postulate — that for every point P outside a line ℓ, there is no more than one line ℓ' going through P and parallel to ℓ — from other postulates of geometry. The independence proof appeared only after the results of Gauss, Bolyai, and Lobachevsky made geometry without this postulate more intuitively clear; see, e. g., [Bonola, 2010].

For this viewpoint, maybe one of the difficulties in solving the P$\stackrel{?}{=}$NP problem is that while there are good intuitive arguments in favor of P$\neq$NP, there is a definite lack of intuitively convincing arguments in favor of P=NP.

3 Example of Intuitive Arguments in Favor of P$\neq$NP

Example of arguments in favor of P$\neq$NP are numerous, many of them boil down to the following: if P=NP, we will have feasible algorithms for solving classes of problems which are now considered highly creative — and for which, therefore, such algorithms are intuitively unlikely.

As we have mentioned earlier, one example of a highly creative activity area is mathematics, where one of main objectives is, given a statement S, to prove either this statement or its negation $\neg S$. We are usually interested in proofs which can be checked by human researchers, and are, thus, of reasonable size. In the usual formal systems of mathematics, the correctness of a formal proof can be checked in polynomial time. So, the problem of finding a reasonable-size proof of a given statement S (or of its negation) belongs to the class NP. If P was equal to NP, then we would be able to have a polynomial-time algorithm for proving theorems — a conclusion which most mathematicians consider unlikely.

Similarly, in theoretical physics, one of the main challenged is to find formulas that describe the observed data. The size of such a formula cannot exceed the amount of data, so the size is feasible. Once a formula is proposed, checking whether all the data is consistent with this formula is easy; thus, the problem of searching for such a formula is in the class NP. So, if P was equal to NP, we would have a feasible algorithm for the activity which is now considered one of the most creative ones – judged, e.g., by the fact that Nobel Prizes in Physics get a lot of publicity and bring a lot of prestige.

4 What We Do in This Paper

In this paper, we propose a physics-motivated argument in favor P=NP.

5 Physical Motivations: The Idea of Scale Invariance

The value of a physical quantity can be measured by using different units. For example, length can be measured in meters, in centimeters, in inches, etc. When we replace the original unit by a new unit which is λ times larger, all numerical values x change, from x to $x' = \dfrac{x}{\lambda}$, so that $x = \lambda \cdot x'$; this transformation is known as *re-scaling*.

For many physical processes, there is no preferred value of a physical quantity; see, e. g., [Feynman, Leighton, and Sands 2005]. For such processes, it is reasonable to require that the corresponding dependence have the same form no matter what measuring unit we use. For example, the dependence of the pendulum's period T on its length L has the form

$$T = f(L) = 2\pi \cdot \sqrt{\frac{L}{g}} = c \cdot \sqrt{L}$$

for an appropriate constant c. If we change the unit of length, so that $L = \lambda \cdot L'$, we get a *similar* dependence

$$T = f(\lambda \cdot L') = c \cdot \sqrt{\lambda \cdot L'} = c \cdot \sqrt{\lambda} \cdot \sqrt{L'}.$$

If we now accordingly re-scale time, to new units which are $\sqrt{\lambda}$ times larger, then we get the exact *same* dependence in the new units $T' = c \cdot \sqrt{L'}$. Since we get the same formula for all measuring unit, physicists say that the pendulum formula is *scale-invariant*.

In general, a dependence $y = f(x)$ is called scale-invariant if each re-scaling of x can be compensated by an appropriate re-scaling of y, i. e., if for every λ, there is a value $C(\lambda)$ for which $f(\lambda \cdot x) = C(\lambda) \cdot f(x)$ for all x and λ. For continuous functions, this functional equation leads to the power law $f(x) = c \cdot x^\alpha$; see, e. g., [Aczel, 2006].

Scale-invariance is ubiquitous in physics: e. g., it helps explain most fundamental equations of physics, such as Einstein's equations of General Relativity, Schrödinger'e equations of quantum mechanics, Maxwell's equations, etc. [Finkelstein, Kreinovich, and Zapatrin, 1986]. It is also useful in explaining many semi-empirical computer-related formulas; see, e. g., [Nguyen and Kreinovich, 1997].

6 Maybe Some Algorithms Are Scale-Invariant

One of the main concepts underlying P and NP is the concept of computational complexity $t_A(n)$ of an algorithm A, which is defined as the largest running time of this algorithm on all inputs of length $\le n$. Similar to physics, in principle, we can use different units to measure the input length: we can use bits, bytes, Kilobytes, Megabytes, etc. It is therefore reasonable to conjecture that for some algorithms, the dependence $t_A(n)$ is scale-invariant – i. e., that its form does not change if we simply change a unit for measuring input length.

It should be mentioned that for discrete variables n, scale-invariance cannot be defined in exactly the same way, since the fractional length n/λ does not always make sense. Thus, we require scale-invariance only *asymptotically*, when $n \to \infty$.

Definition.

- *We say that functions $f(n)$ and $g(n)$ are* asymptotically equivalent *(and denote it by $f(n) \sim g(n)$) if $f(n)/g(n) \to 1$ when $n \to \infty$.*
- *We say that a function $f(n)$ from natural numbers to natural numbers is* asymptotically scale-invariant *if for every integer k, there exists an integer $C(k)$ for which $f(k \cdot n) \sim C(k) \cdot f(n)$.*
- *We say that an algorithm A is* scale-invariant *if its computational complexity function $t_A(n)$ is scale-invariant.*

Now, we are ready to present the promised equivalent reformulation of P=NP, a reformulation that — in view of the ubiquity of scale invariance in physics — provides *some* intuitive argument in favor of this equality.

Proposition. P=NP *if and only if there exists a scale-invariant algorithm for solving propositional satisfiability SAT.*

Proof. If P=NP, then there exists a polynomial-time algorithm A for solving SAT, i. e., an algorithm for which $t_A(n) \le C \cdot n^\alpha$ for some C and α. We can modify this algorithm as follows: first, we run A, then wait until the moment $C \cdot n^\alpha$. Thus modified algorithm A' also solves SAT, and its running time $t_{A'}(n) = C \cdot n^\alpha$ is clearly scale-invariant.

Vice versa, let us assume that A is a scale-invariant algorithm for solving SAT. For $k = 2$, this means that for some number $C(2)$, the ratio $\dfrac{t_A(2n)}{C(2) \cdot t_A(n)}$ tends to 1 as $n \to \infty$. By definition of the limit, that there exists an N such that for all $n \ge N$, we have $\dfrac{t_A(2n)}{C(2) \cdot t_A(n)} \le 2$, i. e.,

$$t_A(2n) \le 2 \cdot C(2) \cdot t_A(n).$$

By induction, for values $n = 2^k \cdot N$, we can now prove that

$$t_A(2^k \cdot N) \le (2 \cdot C(2))^k \cdot t_A(N).$$

For every $n \geq N$, the smallest k for which $2^k \cdot N \neq n$ can be found as

$$k = \lceil \log_2(n/N) \rceil \leq \log_2(n/N) + 1.$$

By definition, the function $t_A(n)$ is non-decreasing, hence $t_A(n) \leq t_A(2^k \cdot N)$ and thus,

$$t_A(n) \leq (2 \cdot C(2))^k \cdot t_A(N).$$

Due to the above inequality for k, we get

$$t_A(n) \leq (2 \cdot C(2))^{\log_2(n/N)+1} \cdot t_A(N) =$$

$$(2 \cdot C(2))^{\log_2(n/N)} \cdot 2 \cdot C(2) \cdot f(N).$$

Here,

$$(2 \cdot C(2))^{\log_2(n/N)} = \left(2^{\log_2(2 \cdot C(2))}\right)^{\log_2(n/N)} =$$

$$2^{\log_2(2 \cdot C(2)) \cdot \log_2(n/N)} =$$

$$\left(2^{\log_2(n/N)}\right)^{\log_2(2 \cdot C(2))} = \left(\frac{n}{N}\right)^{\alpha},$$

where $\alpha \stackrel{\text{def}}{=} \log_2(2 \cdot C(2))$, so

$$t_A(n) \leq \left(\frac{n}{N}\right)^{\alpha} \cdot 2 \cdot C(2) \cdot f(N).$$

Thus, the SAT-solving algorithm A is indeed polynomial time, and hence, P=NP. The proposition is proven.

Acknowledgements

This work was supported in part by the National Science Foundation grants HRD-0734825 and DUE-0926721, by Grant 1 T36 GM078000-01 from the National Institutes of Health. The authors are thankful to Yuri Gurevich for inspiring discussions, and to the anonymous referees for valuable suggestions.

References

Aczel, J. (2006) *Lectures on Functional Differential Equations and their Applications*, Dover, New York.

Bonola, R. (2010) *Non-Euclidean Geometry*, Dover, New York.

Feynman, R., Leighton, R., and Sands, M. (2005) *The Feynman Lectures on Physics*, Addison Wesley, Boston, Massachusetts.

Finkelstein, A.M., Kreinovich, V., and Zapatrin, R.R. (1986) Fundamental physical equations are uniquely determined by their symmetry groups, *Springer Lecture Notes on Mathematics*, **1214**, pp. 159–170.

Kreinovich, V., Lakeyev, A., Rohn, J., and Kahl, P. (1998) *Computational Complexity and Feasibility of Data Processing and Interval Computations*, Kluwer, Dordrecht.

Nguyen, H.T., and Kreinovich, V. (1997) *Applications of Continuous Mathematics to Computer Science*, Kluwer, Dirdrecht.

Papadimitriou, C.H. (1994) *Computational Complexity*, Addison Wesley, San Diego.