NP $\not\subset$ P/poly Proof

André Luiz Barbosa
http://www.andrebarbosa.eti.br
Non-commercial projects: SimuPLC – PLC Simulator & LCE – Electric Commands Language

Abstract. *This paper demonstrates that NP $\not\subset$ P/poly. The way was to generalize the traditional definitions of the classes P, P/poly and NP, to construct an artificial problem (a generalization to SAT: The XG-Poly-SAT, much more difficult than the former) and then to demonstrate that it is in NP but not in P/poly (where the classes P, P/poly and NP are generalized and called too simply P, P/poly and NP in this paper, and then it is explained why the traditional classes P, P/poly and NP should be fixed and replaced by these generalized ones into Theory of Computer Science). The demonstration consists of:*

1. *Definition of Restricted Type X Program*
2. *Definition of the Extended General Poly Problem of Satisfiability of a Boolean Formula – XG-Poly-SAT*
3. *Generalization to classes P, P/poly and NP*
4. *Demonstration that the XG-Poly-SAT is in NP*
5. *Demonstration that the XG-Poly-SAT is not in P/poly*
6. *Demonstration that the Baker-Gill-Solovay Theorem does not refute the proof*
7. *Demonstration that the Razborov-Rudich Theorem does not refute the proof*
8. *Demonstration that the Aaronson-Wigderson Theorem does not refute the proof*

Mathematics Subject Classification (2010). Primary 68Q15; Secondary 68Q17.

Keywords. P, P/poly, NP, Computational Complexity, Formal Languages, Automata Theory.

Contents

1. Introduction

We, following *The Barbosa's Program* and the ideas proposed in [19], could utilize its generalized concepts in order to settle the NP versus P/poly question, which is done here. (About this *Program*, see yet [13, 15].)

Accordingly, in Sections 2 and 3 the *restricted type X programs* and the *XG-Poly-SAT* problem are formally defined, and some notes are included to avoid the traps in these definitions. In order to define the *XG-Poly-SAT*, computational decision problem and poly-time DTM are redefined in more general form, and then the Cook-Levin Theorem is disproved. So, it is proved that the *XG-Poly-SAT* is in NP, with concepts of poly-time *verifier* and *certificate of membership*. In Section 4, this demonstration is repeated with the old kind, using decider poly-time NTM. Then, in Section 5 it is proved that this problem is not in P/poly (therefore, **NP ⊄ P/poly**, naturally, leading to *NEXP ⊄ P/poly*, *NEXP ≠ MA*, *NEXP ≠ BPP*, and other great related results), by demonstrating that it is impossible that any poly-time deterministic computation, even provided at no cost at all with poly-bounded advice function that depends only on the length of input, solves the *XG-Poly-SAT* problem.

In this proof, nothing is assumed about type, structure, form, code, nature, shape or kind of computation, neither structure (or lack thereof) of data, eventually used into any DTM that tries to decide the problem within polynomial time. Otherwise, my proof exploits properties of computation that are specific to real world computers (without *oracles, infinite TMs* and other supernatural devices). In Sections 6, 7 and 8, it is demonstrated that the theoretical barriers against possible attempts to solve the NP versus P/poly question (since NP ⊄ P/poly leads to P ≠ NP, for P ⊂ P/poly) are not applicable to refute my proof. Finally, in Sections 8 and 9 there are some comments about related work (or lack thereof) to really solve this question, and references, respectively.

Shortly, in order for this NP ⊄ P/poly proof be accepted, it is sufficient that the fact if there is an L_z-language (promise problem) separating complexity classes, then they are truly distinct, and the Def. 3.8 are both accepted. On scientific revolution/paradigm shifts, see [17], and, on other amazing insights, see also [26, 27, 28].

2. Definition of Restricted Type X Program

Definition 2.1. Let **S** be a deterministic computer program, let **n** be a finite positive integer and let *time P(n)* be a poly(n) upper bounded number of deterministic computational steps (where time P(n) is the same for all inputs of the same length, and is not previously fixed for all possible programs **S**, but it is fixed for every one). **S** is a *restricted type X program* if and only if the following three conditions are satisfied:

1. **S** allows as input any **n**-bit word (member of arbitrary length **n** from $\{0, 1\}^+$).

2. The **S** behavior must be for each input one of the following:

i. **S** returns in time P(n) **0**;
ii. **S** returns in time P(n) **1**; or
iii. **S** does not halt (never returns any value).

3. The total **S** behavior must be for each **n** one of the following:

i. **S** returns in time P(n) **0** for all the 2^n possible inputs of length **n**; or
ii. **S** returns in time P(n) **1** for at least one possible input of length **n**.

Note 1: The presence of **S** is not to be decided – see Section 3.3.1. Testing whether a computer program is a restricted type X program will not be necessary to the proof. **S** will be given as an absolute assumption: It IS a restricted type X program, and this fact will NOT be under consideration: This is not a contradiction, definitely, since we can easily create innumerous programs of this type and – without need deciding about their types – produce a myriad of instances of the XG-Poly-SAT problem with them – see Sections 5.1 and 5.2, for details.

Note 2: There is no need that the polynomial running times involved in a proof must be previously fixed in order to be defined: For example, what is the fixed polynomial that upper bounds the running time of the reducer concerned in the Cook-Levin Theorem? There is no such fixed polynomial, since this running time depends on the NP problem whose instance is to be reduced to a Boolean formula, but the running time of this reducer is (and must be) polynomial, it is not undefined, of course, otherwise there would be no NP-Completeness. (This insight is formalized in the Def. 3.8.)

Notice that it does not matter at all that we have a different time bound for each NP problem, but the same time bound for each instance of a fixed one, since for this reducer any instance from every NP problem is like just a mere *input* to a deterministic computer program: which is important herein, in fact, is that that polynomial time bound is NOT *uniform*, whereas it is – without any contestation – considered very well defined. (Anyway, see [24] for a surprisingly claim about this issue.)

Note 3: The running time of a fixed program (or machine) **S** on those inputs for which it halts is bounded by a polynomial P(n) (which is a time-constructible function (for each fixed **S**), evidently [22]), hence there must be an equivalent machine (to each fixed **S**) which always halts, and still runs in deterministic polynomial time, of course.

This, however, is not the main point: It is unimportant really whether there must be such an equivalent machine: What matters for my proof, after all, is that this equivalent machine (or program) cannot in general be constructed within deterministic polynomial time, at all, even though we have at free computational expenses a polynomial upper bounded advice function (or set of strings) dependent only on input size, since that polynomial P(n) is *a priori* unknown or not given and – by Proposition 2.1 in [24] – it cannot be computed within deterministic polynomial time (see the detailed proofs in [24] and in Section 5).

Note 4: Into the old traditional definitions of the classes P and NP, a polynomial P(n) must be fixed for whichever program **S** (in order to the XG-Poly-SAT problem (Def. 3.1) is in traditional NP), and it is only over the class of all poly-time machines that such a polynomial is not fixed. However, into the new definitions of the classes P and NP (Defs. 3.5, 3.6 and 3.8), there is no need that there is a fixed polynomial P(n) for all possible **S** in order to the XG-Poly-SAT problem is in the new class NP (Def. 3.5) (see Proposition 3.1). Thus, the comparison with the Cook-Levin Theorem is herein quite well placed (in the note 2 above).

3. Definition of the Extended General Poly Problem of Satisfiability of a Boolean Formula – XG-Poly-SAT

Definition 3.1. Let **S** be a restricted type X program and let **n** be a finite positive integer. The *Extended General Poly Problem of Satisfiability of a Boolean Formula* (*XG-Poly-SAT*) is the question: "– Does **S** return a value **1** for at least one input of length equal to $n\lfloor \log_n(P(n))\rfloor$, where P(n) = (running time of **S** for some input of length **n**)?" Thus, in the XG-Poly-SAT, the input is the pair $\langle \mathbf{S},\mathbf{1^n}\rangle$, clearly, where $\mathbf{1^n}$ is just **n** in unary form. Become aware of that the specific and fixed time P(n) related to **S** is NOT given at all. Verify yet that $\lim_{n\to\infty} \log_n(P(n)) \leq$ degree of any poly(n) that upper bounds the time P(n) related to **S**. (**Note**: $\log_1(P(1))$ is herein defined as equal to **1**.)

Be careful with a possible confusion made about the XG-Poly-SAT and the Bounded Halting problem (BH), defined over triples $\mathbf{w} = \langle \boldsymbol{M},\mathbf{x},\mathbf{1^k}\rangle$, where $\boldsymbol{M}$ is a nondeterministic machine, **x** is a binary string, **k** is an integer, and $\mathbf{w} \in \mathbf{BH}$ if and only if there exists a computation of $\boldsymbol{M}$ on input **x** that halts within **k** steps [12]: The XG-Poly-SAT is a very different problem, since the time P(n) is not given, and the program **S** into the pair $\langle \mathbf{S},\mathbf{1^n}\rangle$ *always* halts for at least one input of length = $n\lfloor \log_n(P(n))\rfloor$, but maybe **S** does not halt for all the other ones. Furthermore, the XG-Poly-SAT cannot be reduced within polynomial time to BH (– See Section 3.4 and [24]). In order to understand why, verify that my XG-Poly-SAT problem is in the new [generalized] class NP (Def. 3.5), by Proposition 3.1, but it is not in that old traditional one.

3.1 Definition of well-formed string

Definition 3.2. Let **w** be a string from $\{\mathbf{0}, \mathbf{1}\}^+$. **w** is a *well-formed string* if and only if **w** has the form $\mathbf{1^+0s}$ – where $\mathbf{1^+}$ is a finite positive integer **n** encoded in unary form and **s** is the binary representation of the DTM (deterministic Turing Machine) that simulates a restricted type X program **S**. For **n** = 13, a well-formed string **w** would be, for instance, **111111111111101001000101001110010010101100100101011001001011110010010110...1**.

3.2 Definition of the XG-Poly-SAT as well-formed string acceptance testing to a language *L*

Definition 3.3. Let $\boldsymbol{L}$ be a formal language over the alphabet $\Sigma = \{\mathbf{0}, \mathbf{1}\}$. A well-formed string $\mathbf{w} \in \boldsymbol{L}$ if and only if the DTM encoded into **w** returns **1** for at least one input of length $n\lfloor \log_n(P(n))\rfloor$, where P(n) = (running time of **S** for some input of length **n**). The XG-Poly-SAT is the well-formed string acceptance testing to $\boldsymbol{L}$.

Note that as the size of a restricted type X program **S** is constant on **n** ($|\mathbf{S}|(n) = \mathbf{c}$), the length of the DTM that simulates **S** is constant too on **n** ($|\mathbf{s}|(n) = \mathbf{k}$), and then $|\mathbf{w}| = \mathbf{n} + \mathbf{1} + \mathbf{k}$. Thus, time P(n) is the same as time P(|**w**|) and time exp(n) is the same as time exp(|**w**|).

3.3 Class of the language *L* and Class of the L_z-language *L*

$\boldsymbol{L}$ is a nonrecursively enumerable (non-RE or non-Turing-recognizable) language [1], since it is undecidable whether or not an eventual result **1** from a computer program occurs within polynomial time [18], besides the undecidability even whether just it halts for some input [4].

(**Note**: The undecidability of the language $\boldsymbol{L}$ does NOT contradict the proof. The XG-

Poly-SAT is not the undecidable decision problem **w** $\in$? ***L***, but just the decidable one **well-formed string w** $\in$? ***L***, as explained in Section 3.3.1, since a well-formed string **w** is *given* as an absolute assumption: **w** IS well-formed string, and this fact is NOT under consideration. See that exactly the same kind of statement holds to traditional formal languages, where the absolute assumption is that the strings to be tested are members from Σ^* [1].)

Language Incompleteness – The computer theorists generally make a big mistake on definition of *computational decision problem*: They think that ones is the same thing that *languages*, as if all decision problems could be modeled as string acceptance testing to formal languages, like in [1, 5, 6]; however, there exist computational decision problems that can only be modeled as string acceptance testing to L_Z*-languages* (as defined in Section 3.3.1), not to languages, like the XG-Poly-SAT. (See Def. 3.9.)

Thus, all computer theorists generally say '*problem*' to mean '*language*' and vice versa. See below an excerpt of text of a preeminent Professor in the area, in [10]:

"*By Savage's theorem, any PROBLEM in P has a polynomial size family of circuits. Thus, to show that a PROBLEM is outside of P it would suffice to show that its circuit complexity is superpolynomial.*" [The words *PROBLEM* are lowercased in the original]

However, the set of all languages is a mere proper subset of the stronger and more powerful set of all L_Z*-languages* (all the computational decision problems), as established below.

3.3.1 More general definitions for NP, P, P/poly and definition for L_Z-language

Definition 3.4. Let L_Z be a language over a finite alphabet, Σ, and let $L \subseteq L_Z$. We will call L an L_Z*-language*. If $L_Z = \Sigma^*$, then L is a Σ^*-language, a *trivial* L_Z*-language*, which is the same as language (Σ^*-language = language). The complement of an L_Z-language A is another L_Z-language $\bar{A} = L_Z - A$. Thus, L_Z*-language* is simply a generalization to *language* and a string acceptance testing to L is a *computational decision problem* where the string to be tested is *necessarily* member from L_Z. If a *language* can be characterized as a *set*, an L_Z*-language* can be characterized as a *subset*, that is to say a *set into another*.

Observe that a string acceptance testing to L is a *computational decision problem*, but L, rigorously, is not only a *language*, because $L \subseteq L_Z$, which is more restrict than simply $L \subseteq \Sigma^*$, which should hold if L was only a language [6]. Thus, all the computational decision problems can be modeled as string acceptance testing to L_Z-languages, for to accept a string from any determined subset of Σ^* is much more general than do it just from Σ^*, of course.

The main point herein is that the central relevance of the languages is originated in the fact that they model problems, not the inverse. Hence, great part of the Theory of Computation is about languages because of the mistake referred to in Section 3.3. When this mistake – that it is said as *mistake* because it leaves legitimate problems out of that old traditional definition – is fixed, the Theory of Computation will certainly study the generalization to language: The richer and stronger concept of L_Z-language.

A language over Σ is a subset of Σ^*, and an L_Z-language is a subset of the language L_Z over Σ. However, as $L \subseteq L_Z$ and $L_Z \subseteq \Sigma^*$, then $L \subseteq \Sigma^*$, which implies that all L_Z-languages are Σ^*-languages, or simply languages, too, naturally. Any language L is also an L-language, and any L_Z-language L is also a language L. In fact, if $L_Y \supseteq L_Z$ then any L_Z-language L is an L_Y-language L, too. But the great advantage of the L_Z-languages is that string acceptance testing to ones can be much easier than to languages, because the strings **x** to be tested are in special form: $\mathbf{x} \in L_Z$ (this is an absolute assumption). Hence, if we know that all the strings to be tested are from a fixed language L_Z, then it is worth to model this problem as an L_Z-language; but if we do not know it, we must model it as a simple language, of course.

Consequently, the concept of L_Z-language allows the insertion of previous knowledge about the form of the strings to be tested – when they were already constructed in special form or previously accepted by another machine – into traditional concept of language.

(**Note**: If the machine M that decides an L_Z*-language* L is fed a string **x** that is in L_Z, then M *must* decide whether or not **x** is in L, anyway returning correct answer to $\mathbf{x} \in? L$; on the other hand, if M is fed any string that is not in L_Z, it may do whatever, returning anything, even *incorrect* answer to $\mathbf{x} \in? L$ [Σ^**-language* L, in this case], or even not halting at all.)

For instance, the language $\{0^n1^n \mid n > 0\}$ over $\{0, 1\}$ is not regular, but verify that if $L_Z = \{0^n1^n \mid n > 0\} \cup \{1^n0^n \mid n > 0\}$, for example, then the L_Z-language $L_1 = \{0^n1^n \mid n > 0\}$ is regular and can be decided by the NFA $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$, where $\delta(q_0, 0) = \{q_2\}$, $\delta(q_0, 1) = \{q_1\}$, $\delta(q_1, 0) = \varnothing$, $\delta(q_1, 1) = \varnothing$, $\delta(q_2, 0) = \{q_2\}$, $\delta(q_2, 1) = \{q_2\}$, and there are not ε-moves.

Verify that this NFA M recognizes the language $L = 0\{0, 1\}^*$ and $\{0^n1^n \mid n > 0\} = 0\{0, 1\}^* \cap (\{0^n1^n \mid n > 0\} \cup \{1^n0^n \mid n > 0\})$. In fact, this is not coincidence:

Theorem 3.1. If a machine M (DFA, NFA, PDA, DTM, NTM, etc.) recognizes a language L, then M recognizes any L_Z-language $L_1 = L \cap L_Z$.

Proof. Suppose that a string $\mathbf{x} \in L_Z$-language L_1 was accepted by a machine M: Then, $\mathbf{x} \in L_Z$ (this is an absolute assumption: All the strings to be tested must be member from L_Z) and $\mathbf{x} \in L$ (the language that M recognizes, regardless of the special form of **x**), which implies that $\mathbf{x} \in L \cap L_Z$; on the other hand, if $\mathbf{x} \in L \cap L_Z$, then **x** will be accepted by M, because $\mathbf{x} \in L_Z$ (**x** can be tested) and $\mathbf{x} \in L$ (**x** will be accepted by definition of string acceptance testing to languages), which implies that the L_Z-language recognized by M $L_1 = L \cap L_Z$. □

See, thus, the proposed fix and generalization to the traditional formal definition for the class **NP** (Nondeterministic Polynomial Time) [14]:

Definition 3.5. Let L be an L_Z-language. $L \in$ **NP** if and only if there is a binary relation $R \subseteq L_Z \times \Sigma^*$ and a known and given finite fixed positive integer $\boldsymbol{p}$ such that the following two conditions are satisfied:

1. For all $x \in L_Z$, $x \in L \Leftrightarrow \exists y \in \Sigma^*$ such that $(x, y) \in R$ and $|y| \in O(|x|^p)$; and
2. The language $L_r = \{x\#y : (x, y) \in R\}$ over $\Sigma \cup \{\#\}$ is decidable by a poly-time DTM.

A DTM that decides L_r is called a *verifier* for L and a **y** such that $(x, y) \in R$ is called a *certificate of membership* or *witness* of **x** in L. Note that – as $\mathbf{x} \in L_Z$ (this is an absolute assumption, by Def. 3.4) – we do not need to describe what language L_z is allowed here. Hence, condition 2 does NOT require any knowledge about how to decide L_z in order to decide whether x#y is in L_r, plainly.

Verify that when $L_z = \Sigma^*$ and the traditional definition for poly-time DTM there is utilized, the formal definition above is equivalent to the traditional one for the class **NP** – a set of mere languages –, which implies that this one is just a particular case of the proposed fixed definition. Consequently, we can name the traditional class **NP** as class **NP-SAT** (or, shortly, **SNP** or **NP$_t$**), where the Cook-Levin Theorem (with the hidden assumption referred in [24]) and all the other mathematical truths on the traditional class **NP** continue holding in (replacing "*P ≠ NP*" and "*P = NP*" by "*SAT is not in P*" and "*SAT is in P*", respectively, etc.). Alternatively, we could call the true class **NP** defined above – an actual set of computational decision problems, or L_z-languages – as class **NP-XG-SAT** (or, shortly, **XNP**), for example, but this naming method would be a mistake: A subset would have the name of the set and the set would have a derived name of the subset, which is hard to explain, confuse and damages the clearness of the notation. The same happens with the classes **P** and **P/poly** in Defs. 3.6 and 3.7.

Proposition 3.1. XG-Poly-SAT is in class NP.

Proof. Into the Def. 3.5, for $\boldsymbol{L}$ modeling the XG-Poly-SAT, $\boldsymbol{\Sigma} = \{\mathbf{0}, \mathbf{1}\}$, $\boldsymbol{L_z}$ is the set of all well-formed strings (as defined in Section 3.1), **p** = 1, and **y** is a word that encodes an input of length **n** (**i$_1$**) and another one of length $\mathbf{n}\,\lfloor\mathbf{log_n(P(n))}\rfloor$ (**i$_2$**), where (x, y) ∈ R if and only if the program **S** encoded into the well-formed string **x** halts in time P(n) for the input **i$_1$** and returns **1** for the input **i$_2$** (a poly-time DTM that on ⟨**x#y**⟩ decodes and simulates **S** running having **i$_1$** as input, then counts the running time until it halts (therefore computing the time P(n)) and verifies whether the length of **i$_2$** is equal to $\mathbf{n}\,\lfloor\mathbf{log_n(P(n))}\rfloor$, and after simulates **S** running having **i$_2$** as input, finally confirming whether it returns **1**, is, in fact, the apt *verifier* for $\boldsymbol{L}$). Hence, $\boldsymbol{L}$ (XG-Poly-SAT) is in NP. □

Note that although the deterministic polynomial time $T(n) = O(n^{\mathbf{i}})$ that the witness predicate is decided is a different polynomial for each input **x**, XG-Poly-SAT is a single problem (it is false that any recursive decision problem is poly-time reducible to it, since T(n) is not previously fixed for all **S**, but it is fixed for every one, by Def. 2.1 – See the note 1 below, for details), where **i** does not depend on **n**, even though it does on **x**. Consequently, (x, y) ∈? R is really decidable in deterministic polynomial time, by Def. 3.8, and the proof above is wholly correct: the XG-Poly-SAT is in NP, undoubtedly.

See that XG-Poly-SAT has strings of the form $\mathbf{1^n0s}$, where **s** is a DTM simulating a restricted X program **S** that accepts within polynomial time some string of length **n** (returning **1** for some **n**-bit input). Notice that we do NOT need to check whether **S** is a restricted X program, by Def. 3.4.

(**Note 1**: Suppose that someone says that the XG-Poly-SAT is not in NP, since its complexity class is really undefined, and it can be, for example, EEXP-Hard (for double exponential time), reasoning as below:

"Let L be an EEXP problem, M be the deterministic Turing Machine that solves L in time $t(n) = 2^{2\wedge\{\text{poly}(n)\}}$. Then we can reduce L to XG-Poly-SAT as follows: Given an input **x** for the problem L, we construct a program **S** that ignores its input and simulates M on input **x**. The promise is satisfied by the constant polynomial p(n') = t(|x|), and clearly (**S**,1) is an instance of XG-Poly-SAT if and only if *M* accepts **x**."

Fortunately, constructions like above cannot disprove that XG-Poly-SAT is in NP, since they do not take into account that time P(n) is not previously fixed for all possible programs **S**, but it is fixed for every one, as stated in Def. 2.1 – hence, as $2^{2^\wedge\{\text{poly}(|x|)\}}$ is not upper bounded by any fixed poly(n), that program **S** is not a restricted type X program, and clearly (**S**,1) is NOT an instance of the XG-Poly-SAT.

Finally, see also that the function $2^{2^\wedge\{\text{poly}(|x|)\}}$ = t(|x|) is not constant, but depends on |x|. However, if **x** is fixed into that TM *M* simulated by **S**, then this function is a constant (and then *M* halts on **x** within only *O*(1) steps, since *M* and **x** are fixed independent of **n**); nonetheless, in this case, *M* does not solve *L*, of course, and then the disproof above fails.)

(**Note 2**: Suppose, yet, that anyone else says that the XG-Poly-SAT is not in NP, since Proposition 3.1 is wrong, as long as either no poly-time TM can simulate a universal TM, or it – about the *verifier* for ***L*** that on ⟨**x#y**⟩ simulates **S** running having inputs encoded into **y** – does not consider the running time of this simulations, which could be non-polynomial.

Fortunately yet, these refutations of Proposition 3.1 are equivocated, since a program **S** is always restricted (hence, it is NOT a universal TM), and the running time of the simulations of the program **S** (encoded into **x**) running having inputs $\mathbf{i}_1$ and then $\mathbf{i}_2$ encoded into **y** ARE necessarily (must be) polynomial, since *time P(n)* is a time-constructible function, by Def. 2.1.

See, however, this interesting review:

"– The author proposes that XG-Poly-SAT is in (promise-)NP but not in (promise-)P. He is right about the second part, but incorrect about the first part: XG-Poly-SAT is unconditionally not in promise-NP. He gives a simple but fallacious argument that XG-Poly-SAT is in promise-NP on p. 8. In note 2 on p. 8 he anticipates but rejects a counterargument, but he is wrong and this counterargument is essentially correct.

The reason is as follows: for any Turing machine *M* and positive integer *t*, we can form a machine M_t that outputs **0** on all inputs except those of length *t*, on which it behaves like *M*. If *M* always halts and *M*'s behavior depends solely on its input length (call this latter restriction *semi-blindness*), then M_t is always a restricted type-X machine.

It is known there exists a unary language *L* that is decidable, yet it is not in EXPTIME, hence not in NP. There is a *semi-blind* machine *M* that decides *L* correctly on each input having the form *1^t*. But if XG-Poly-SAT were in promise-NP, then we could solve *L* in NP: given input *x* of form *x* = *1^t*, we decide whether *x* is in *L* by running the presumed NP verifier on the input (M_t, *1^t*), which obeys the promise. (If *x* is not of form *1^t*, then we can reject *x*.)"

Verify that that conclusion is not true: In order to try to decide that language *L* in NP, as proposed above, we must run the NP verifier on the inputs of form (*M, 1^t*), not (M_t, *1^t*), since to solve whether *M* accepts *1^t* is quite different from do it about M_t, for *M* is not the same thing neither has the same running time complexity as all the machines M_1, M_2, M_3, ... taken into account as a [countably infinite] set. The running time of all those M_t is only *O*(1), since *t* is a fixed constant into M_t, independent of **n** (|input|), while *L(M)* is not even in EXPTIME (hence, *M* is not a restricted type-X machine, at all), which implies, fortunately, that the input (*M, 1^t*) does not obey the promise in Def. 2.1, and then *L* cannot be decided in NP as proposed by that smart reviewer, and then the disproof above fails too.

See, also, another interesting and similar review:

"– Let *L* be any computable language, encoded in unary, and *M* a deterministic TM that solves *L*. The program $S=S_x$ takes its input **y**, and compare its length to **x**. If |y| = |x|, then S(y) simulates *M* on input **x**, and, if *M*(x) accepts, S(y) accepts. Otherwise, S(y) rejects. If |y| is any other value, S(y) rejects.

Clearly, this S runs in linear time, since all it has to do is count the length of y, **except** when |y| = |x|, but this is only finitely many exceptions, and hence doesn't change the asymptotic running time of S. To reduce *L* to XG-Poly-SAT: map **x** to the pair (S_x, 1^{|x|})."

Verify that that conclusion is not true too: By means of the same reasoning above, we could prove that that language *L* would be in NP, since S_x and its input may be mapped to a Boolean expression in deterministic polynomial time (for the running time of S_x is really only a fixed constant), and then this contradiction shows that this other disproof fails too.)

Note that even the language L_r in item 2 above is, in fact, an L_z-language, where L_z is the set of all strings of the form x#y. In fact, all complexity classes can be generalized with the concept of *L_z-language*, like these new definitions proposed for the classes **P** and **P/poly**:

Definition 3.6. Let L be an L_z-language. $L \in$ **P** if and only if for all $x \in L_z$, $x \in ?\ L$ is decidable by a poly-time DTM. Be careful with the traps: For example, all **L_z-languages** L_z are trivially in **P** (where L_z can be *any* language, even non-Turing-recognizable ones), which does NOT mean that all **languages** L_z (Σ^*-languages) are in **P**, noticeably.

Definition 3.7. Let L be an L_z-language. $L \in$ **P/poly** if there is a language A in **P** and a set of advice strings $\{a_0, a_1, \ldots\}$ such that $|a_n| \leq \mathbf{n}^{O(1)}$ and **x** is in L if and only if $(\mathbf{x}, \mathbf{a}_{|x|})$ is in A. [20] Observe that it suffices the existence of such a set of advice strings, where an eventual cost on computing them does not matter at all, neither even whether they are computable.

Notice that the proper definition of *deterministic polynomial-time computation* is more general herein, without losing its more important characteristic: To be understood loosely as "feasible in practice", where the critique in [29] is not applicable:

Definition 3.8: Poly-time DTM. A DTM is said to be polynomial-time if its running time $T(n) = O(n^k)$, where $k = O(1)$, even that **k** depends some way on input. (n = |input|.)

Into the old traditional definition, **k** must be a fixed constant (that does not depend on **n**, obviously), but this stronger restriction is not essential to the vital matter: To maintain the character of vaguely practicable for deterministic polynomial-time computations. In XG-Poly-SAT, the T(n) of its *verifier* is in $O(n^k)$, where **k** depends on the **S** encoded into **w**, but it is in $O(1)$, since $O(\lfloor \log_n(P(n)) \rfloor) = O(1)$, which does not depend on **n**, but only on degree of the minimum poly(n) that upper bounds the time P(n) related to **S** – and **k** cannot be computed [1] neither is given, but it is a fixed constant for each fixed **S**, by Def. 2.1. Furthermore, the traditional definition of poly-time DTM asserts a hidden assumption: **k** must be *a priori* a *known* and *given* fixed constant, as revealed in [24].

See that if $T(n) = O(2^{\text{poly}(n)})$, for example, then $T(n) = O(n^k)$, where **k** (poly(n) $\log_n 2$) is not in $O(1)$, evidently, and is upper unbounded (for non-constant poly(n), of course): hence, in this case T(n) is not polynomial at all. The same happens with $T(n) = O(n^{\log n})$. If $T(n) = O(n^k)$, where **k** is, for example, the [arbitrary] position of the first 1 in **w** (or 1, if $\mathbf{w} = 0^n$), then **k** is not in $O(1)$ too, for those possible positions can be from 1 to |**w**| = **n**, hence in the extreme case T(n) = $O(n^n)$. On the other hand, if T(n) = $O(n^{g(n)})$, but now g(n) is upper bounded by a finite positive constant **k**, that is $\lim_{n\to\infty} g(n) < k$, then T(n) = $O(n^k)$, whence it is polynomial.

Some experts are asserting: "– The XG-Poly-SAT is not in NP (in the author's terms): the polynomial n^k CANNOT depend on the input." However, this assertion is false, being true only for the old traditional definition of polynomial-time DTM, since in the new definition (Def. 3.8), the polynomial CAN definitely depend on the input – as long as that **k** is in $O(1)$. Think: This is just a matter of Math object definition, not of mathematical error or correctness, at all. We are not obligated to follow obsolete definitions only because they are established, unless the Science is finished (or dead). See Section 9.

Very important: Verify that these new definitions of the classes P, P/poly and NP are simply good generalizations of the old traditional ones: Any traditional P, P/poly or NP problem IS too, respectively, in the new class P, P/poly or NP defined above (even though the converse is in general false, since these new generalized classes are strictly larger than the traditional ones), and any superpolynomial deterministic, deterministic/poly or nondeterministic problem is NOT in the new class P, P/poly or NP, respectively, which proves that these generalizations are consistent and smooth.

3.3.2 A new NP genealogy

In fact, the traditional class NP (that we call herein NP_t) can be divided into two new disjoint classes: NP_g (when that *p(n)* is known and given) and NP_u (when *p(n)* is unknown or not given), where $NP_t = NP_g \cup NP_u$ and $NP_k \cap NP_u = \emptyset$. Into traditional beliefs, NP_t is considerate equal to NP_g, and NP_u is considerate equal to $\emptyset$, but these considerations take not account that the class NP_u can be a genuine, useful and very important complexity class into the development of the Computational Complexity Theory, with great powerful applications in mathematically proven unbreakable within polynomial time public-key cryptography, for instance. By the way, see [28].

Into more formal terms, lets see the definitions for the two new disjoint classes that build the traditional class $\mathbf{NP_t}$: $\mathbf{NP_g}$ and $\mathbf{NP_u}$ (traditional Nondeterministic Polynomial Time when the involved polynomial time is or not given, respectively):

Definition 3.9. $\mathbf{NP_g}$. Let L be a language over Σ. $L \in \mathbf{NP_g}$ if and only if there is a binary relation $R \subseteq \Sigma^* \times \Sigma^*$ and a known and given finite fixed positive integer $\boldsymbol{p}$ such that the following two conditions are satisfied:

1. For all $x \in \Sigma^*$, $x \in L \Leftrightarrow \exists y \in \Sigma^*$ such that $(x, y) \in R$ and $|y| \in O(|x|^p)$; and
2. The language $L_r = \{x\#y : (x, y) \in R\}$ over $\Sigma \cup \{\#\}$ is decidable by a polynomial-time DTM whose polynomial is fixed, known and given.

Definition 3.10. $\mathbf{NP_u}$. Let L be a language over Σ. $L \in \mathbf{NP_u}$ if and only if there is a binary relation $R \subseteq \Sigma^* \times \Sigma^*$ and a known and given finite fixed positive integer $\boldsymbol{p}$ such that the following two conditions are satisfied:

1. For all $x \in \Sigma^*$, $x \in L \Leftrightarrow \exists y \in \Sigma^*$ such that $(x, y) \in R$ and $|y| \in O(|x|^p)$; and
2. The language $L_r = \{x\#y : (x, y) \in R\}$ over $\Sigma \cup \{\#\}$ is decidable by a polynomial-time DTM whose polynomial is fixed, but unknown or not given.

Definition 3.11. $\mathbf{NP_t}$. $NP_t = NP_g \cup NP_u$.

Let's see now the definitions for the class $\mathbf{NP_n}$: Non-uniform Nondeterministic Polynomial Time, as $\mathrm{NP_t}$ but when the involved polynomial time is NOT fixed:

Definition 3.11. $\mathbf{NP_n}$. Let L be a language over Σ. $L \in \mathbf{NP_n}$ if and only if there is a binary relation $\mathrm{R} \subseteq \Sigma^* \times \Sigma^*$ and a known and given finite fixed positive integer ***p*** such that the following two conditions are satisfied:

1. For all $\mathrm{x} \in \Sigma^*$, $\mathrm{x} \in L \Leftrightarrow \exists \mathrm{y} \in \Sigma^*$ such that $(\mathrm{x}, \mathrm{y}) \in \mathrm{R}$ and $|\mathrm{y}| \in O(|\mathrm{x}|^{\mathrm{p}})$; and
2. The language $L_{\mathrm{r}} = \{\mathrm{x}\#\mathrm{y} : (\mathrm{x}, \mathrm{y}) \in \mathrm{R}\}$ over $\Sigma \cup \{\#\}$ is decidable by a poly-time DTM, as defined in Def. 3.8.

Now, the old traditional class NP ($\mathrm{NP_t}$) is clearly seen simply as a proper subset of our new and legitimate extended class NP: $(\mathrm{NP_t} \cup \mathrm{NP_n}) \subset \mathrm{NP}$ (as defined in Def. 3.5).

As always, in all the definitions above a DTM that decides L_{r} is called a *verifier* for L and a **y** such that $(\mathrm{x}, \mathrm{y}) \in \mathrm{R}$ is called a *certificate of membership* or *witness* of **x** in L.

3.3.3 L_z-languages and Promise Problems

An *L_z-language L* can be considered as a *promise problem* $\prod$, as introduced by Alan L. Selman [Information and Computation, Vol. 78, Issue 2, (1988), pp. 87-98] and defined in [9], where the *promise* $(\prod_{\mathrm{YES}} \cup \prod_{\mathrm{NO}}) = L_z$, $\prod_{\mathrm{YES}} = L$, $\prod_{\mathrm{NO}} = L_z - L$, and its restricted alphabet {0, 1} is generalized to any finite alphabet Σ. Nonetheless, notice that the concepts, notation, generality, power and applicability of the *L_z-languages* are clearer, richer, simpler, conciser, more elegant, aesthetic and stronger than ones of the *promise problems*.

3.4 More general definition for Computational Decision Problem

Note yet that the definition of *computational decision problem* used herein is also more general, without losing its more essential attribute: To model *all* real computer-based questions – not only a small part of them – having one and only one answer from two alternatives [16]:

Definition 3.9: A *computational decision problem* is any arbitrary **Yes**-or-**No** (**True**-or-**False**) question on a finite or infinite set of inputs (strings of any finite length over a finite alphabet Σ), where these ones are necessarily member from another determined set (or consistently the set of inputs *of obligatory specified form* for which the problem returns **Yes** (**True**)). Equivalently, decision problems are completely isomorphic to L_z-languages of strings, and can always be modeled as string acceptance testing to L_z-languages.

Into the traditional definition for computational decision problem [1, 16], using plain languages, the inputs for a problem are simply from Σ^*, whereas for this more general definition they are from any arbitrary subset of Σ^*. So, we can consider a *traditional* problem (*language*) as a *set*, and a *more general* one (*L_z-language*) as a *subset* of a *set*. Hence, the set of all languages ($L_z = \Sigma^*$) is just a little proper subset of the set of all L_z-languages (L_z = any subset of Σ^*). So, only *one* set characterizes language, but we need *two* sets for L_z-language.

See yet that, for this generalization, the strings to be tested (into a string acceptance testing to an L_z-language) are necessarily (must be) member from L_z (whatever L_z is), where this fact IS an absolute assumption and IS NOT under consideration. Verify that exactly the same kind of statement holds to traditional formal languages, where the absolute assumption is that the strings to be tested (into a string acceptance testing to a language) are always necessarily (must be) members from Σ^*.

3.4.1 The falseness of the Cook-Levin Theorem

Theorem 3.2. The Cook-Levin Theorem (CLT) is false.

Proof. See it in [24]. There are some comments on it in [25].

3.4.1.1 How can a Theorem be false?

Since a *theorem* is an absolute mathematical truth, how can the Cook-Levin Theorem be false? – See it in [24].

4. Old demonstration that the XG-Poly-SAT is in NP

Given **n** and a restricted type X program **S**, the question “Does **S** return a value **1** for at least one input of length = n $\lfloor \log_n(P(n)) \rfloor$, where P(n) = (running time of **S** for some input of length **n**)?” can be decided in nondeterministic poly-time (time NP(n): similar to time P(n), using nondeterminism), since can be constructed a universal NTM (nondeterministic TM) that simply simulates the running of **S** and tests it for all $2^{n \lfloor \log n(P(n)) \rfloor}$ possible inputs of that length at the same time (“on parallel”) and verifies in time NP(n) the returns: If they are **0** for all the inputs, then the NTM will answer “No” after the conclusion of the last computation path (branch); on the other hand, if at least one return is **1** then the NTM will answer “Yes” at the end of the first path that returns **1**, regardless of whether or not the other paths are yet running. One and only one of these two events must happen in time NP(n), by Def. 2.1.

Verify that the NTM, in order to compute the time P(n), counts initially this time simulating the running of **S** – and counting the number of its steps – for all the 2^n possible ones (like above, “on parallel”), and waiting anyone to halt. Note that, by Def. 2.1, **S** must halt in time P(n) for at least one input of any length.

5. Demonstration that the XG-Poly-SAT is not in P/poly

Theorem 5.1. NEXP $\not\subset$ P/poly.

Proof. As demonstrated in Sections 3.3.1 and 4, any instance **w** of the EXP-SAT can be recognized in nondeterministic polynomial time. However, can it be recognized by a poly-time DTM provided at no cost at all with a set of poly-bounded advice strings (or function) **h** that depend only on |w| (or, equivalently, with a [*non-uniform*] family of polynomial-size Boolean circuits [20])?

By hypothesis, consider that it can: In this case, must exist a DTM **Q** that – given a poly-bounded advice function **h** (only dependent on |w|), a positive integer **n** and a restricted type X program **S** into **w** – answers correctly within polynomial time the question “Does **S** return a value **1** for at least one input of length = n $\lfloor \log_n(P(n)) \rfloor$, where P(n) = (running time of **S** for some input of length **n**)?” (If **w** is in XG-Poly-SAT, then **Q(w, h(**|w|**))** = “Yes”, else **Q(w, h(**|w|**))** = “No”).

Proposition 5.1. While the function **h** is like a supernatural device, that computes anything that depends only on the size of input and can be stored within poly-space, even answers to undecidable problems or superpoly-time or incomputable functions, at absolutely no cost at all –, the DTM **Q** is, in fact, a real computer program. Although it may work entirely in a different way from someone would expect from the method that the XG-Poly-SAT was defined, **Q** cannot be a magical or dream machine, since it must be an actual machine.

So, let **W**: $\Sigma^* \times L_Z \rightarrow \mathbf{N}$ be a function with a DTM and a well-formed input for it as arguments, where if **W**(**Q**, **w**) = **m** (**Q** can simulate the running of **S** into **w** and test some inputs for **S** in such a simulation – considered herein a step-by-step process running **S** into **Q**), then **m** is the number of inputs for **S** simulated by **Q** in this process: $0 \leq \mathbf{m} \leq 2^n$.

Note: It does not matter for this proof whether **W** is a computable function or not; and if **X** is not a DTM or is not interested in the XG-Poly-SAT problem, then **W**(**X**, **w**) is defined as 0.

Thus, in order to answer the question, there are no miracles: **Q** must found – even though with the help from **h** on |w| – the value of the time P(n), in order to evaluate $n\lfloor \log_n(P(n)) \rfloor$, since if **Q** does not know this information, then it is completely lost about even what to search or compute in order to answer the main question on XG-Poly-SAT, where **Q** can act into only four possible ways (where **m** = **W**(**Q**, **w**)):

1. **Q** simulates the running of **S** for:

   i. All the possible inputs (**m** = 2^n);
   ii. All the inputs from an arbitrary nonempty proper subset of all them ($0 < \mathbf{m} < 2^n$); or
   iii. Only one input (or all from a nonempty proper subset of all them) previously computed, where **S** halts on this input (or where **S** halts on anyone of these inputs) ($\mathbf{m} = \mathbf{d} < 2^n$).

2. **Q** does not simulate the running of **S** at all (**m** = 0).

Note: all the inputs in this Section are **n**-bit, unless another is explicitly indicated.

Proof. These ways are exhaustive: Either **Q** simulates the running of **S** or not; and, if **Q** simulates the running of **S**, then it can test on it all the possible inputs (1.i); arbitrarily less than all ones (1.ii); or just one (or all from a nonempty proper subset of all them) that was anyway previously computed whose return decides the question (1.iii). Unfortunately, there are no more alternatives besides that ones. (Note: Into ways (1.ii) and (1.iii), **m** must be polynomial in **n** in order to **Q** can decide the XG-Poly-SAT in deterministic polynomial time.)

As well, the running time of a universal NTM that decides in time NP(n) the XG-Poly-SAT – as in Section 4 – cannot be upper bounded by any fixed poly(n). Moreover, a program **S** does not necessarily halt for all its possible inputs. Furthermore, the time P(n) in Def. 2.1 cannot be upper bounded by any fixed poly(n), too. Thus, in general, cannot exist any fixed poly(n) number of TM configurations that represents the entire processing of **S**.

Additionally, as to find the input whose return decides the question and simulate the running of **S** only for this input is impossible (see in Way 1.iii below), the particular fixed running time P(n) of a specific restricted type X program cannot be computed within any fixed poly(n) upper bounded number of deterministic computational steps. Hence, an instance of the XG-Poly-SAT cannot be reduced within polynomial time into another one of another poly-time decidable problem, because the reducer machine must run within polynomial time in this case, but it cannot previously know or compute what upper bounds that time P(n), by Proposition 2.1 in [24]. □

Suppose that someone claims, with the following argument, that the NP $\not\subset$ P/poly proof of mine fails: "– The author assumes that the 4 ways mentioned are the only way to solve the problem. Why can't the DTM **Q** decide the question some other way?"

The answer is not complicated: **Q** cannot decide the question by some other way because there is no another possible way to decide the XG-Poly-SAT besides the four ones mentioned above: These ways do not specify type, structure, form, code, nature, shape or kind of computation, neither structure (or lack thereof) of data – but just the **number (m) and kind of inputs (arbitrary or computed) tested** in eventual simulated running of **S** –, into *any* running of *any* DTM that tries to decide the XG-Poly-SAT: (1) all inputs (**m** = 2^n); (2) arbitrary ones less than all (0 < **m** < 2^n); (3) computed ones less than all (**m** = **d** < 2^n); or (4) none (there is no simulating **S** at all) (**m** = 0).

Can there be some other way? No, by a reasoning similar to *pigeonholes* from *pigeonhole principle*: Either **Q** simulates **S** or not. And simulating **S** for more than all inputs – or for any subset with exponential number of them – leads to exp(n) running time, as explained in the Way 1.i; less than none go to negative number of inputs, which makes no sense in actual computations; and between these limits the number and kind of inputs for eventual simulated running of **S** must be one from the four mentioned above.

Consequently, all the possible deterministic computations to decide the XG-Poly-SAT are really into one from these four ways.

Can we create new ways to decide in deterministic polynomial time the XG-Poly-SAT combining the four ones? Unfortunately, no way: The way 1.i is useless to decide in deterministic polynomial time the question; the way 1.ii is useless to decide in any time the XG-Poly-SAT; and the combination of the ways 1.iii and 2 results simply in the way 2 – when none result from the simulation is used by **Q** in order to answer the question, which is the case treated below in the way 2.

Hence, claims like above do not go to refute this NP $\not\subset$ P/poly proof.

Note, yet, that the method utilized in this proof cannot be adapted to decide whether SAT is in P, because if a program **S** is simulating a Boolean formula with **n** variables, it *must* always halt for all the possible inputs; however, this additional restricted condition cannot be held in general restricted type X programs, like one in the proof of the Propositions 5.2 below. Hence, to decide whether an arbitrary general deterministic computer program halts for determined input (which is undecidable, by the Rice's Theorem [11]) cannot be reduced to SAT as it does to XG-Poly-SAT (as demonstrated in this proof), and then any attempt to adapt my proof to solve whether SAT is in P is condemned to fault.

See that if **S** is simulating a Boolean formula with **n** variables, then the Rice's Theorem cannot be applied to the **S** behavior for any input, since it is restricted for all the possible ones.

Finally, suppose that else one tries to refute the proof saying:

"This proof follows a common theme: Defines an NP problem with a certain structure, argues that any algorithm that solves that problem must work in a certain way and any algorithm that works that way must examine an exponential number of possibilities. But we can't assume anything about how an algorithm works. Algorithms can ignore the underlying semantic meaning of the input and focus on the syntactic part, the bits themselves."

As in the previous "refutation" of my proof, the answer is also not complicated: If the DTM **Q** ignores the underlying semantic meaning of **w** and focus on its syntactic part, the bits themselves, considering **w** just a series of bits, this approach only places **Q** into the Way 2 – where **Q** does not simulate the running of **S** at all (**m** = 0) –, and then the proof continues to hold, naturally.

Shortly, the spirit of the proof is very simple: the XG-Poly-SAT is decidable by brute-force search because whether **S** halts for at least one input from all the 2^n possible ones is decidable, whereas whether **S** halts for at least one from a nonempty proper subset of them is in general undecidable, by Def. 2.1, which does that all the other ways to decide the XG-Poly-SAT (without brute-force searching) be absolutely hopeless.

Consequently, we can say that one of the profoundest questions in Computational Complexity Theory was solved by this plain ingenious characterization, the Def. 2.1!

Be brave and see below that all these four exhaustive ways to decide in deterministic polynomial time the XG-Poly-SAT fail:

Way 1.i Q simulates the running of S for all the possible inputs (m = 2^n):

The obvious way in order to the DTM **Q** to compute the time P(n) is to simulate one step by time the running of **S** for all the 2^n possible ones (in a breadth-first search, to avoid running forever in a computation path that does not halt), counting the number of the steps in each one of these paths of this simulation and waiting anyone to halt. Note that, by Def. 2.1, **S** must halt in time P(n) for at least one input of any length. (After this, it suffices to simulate the running of **S** for all inputs of length = n $\lfloor \log_n(P(n)) \rfloor$ (in a breadth-first search), verifying whether there is a result **1**, in order to decide the XG-Poly-SAT.)

Nevertheless, this brute-force method, on worst case, can compute the time P(n) only at the end of testing all these inputs, in time exp(n).

Notice that the advice function **h(**|w|**)** cannot help enough **Q** herein, in this step-by-step simulation, in order to avoid brute-force searching, since **h** can encode only fixed poly(|w|)-bounded quantity of information, whereas **Q** needs to treat eventually inputs for **S** whose length is upper unbounded by any fixed poly(|w|), searching unavoidably exp(n)-bounded (hence exp(|w|)-bounded) number (also upper unbounded by any fixed exp(n)) of possible results from **S**, unfortunately.

Way 1.ii Q simulates the running of S for all the inputs from an arbitrary nonempty proper subset of all them (0 < m < 2^n):

Note that to simulate the running of **S** only for a polynomial number of arbitrary inputs (or just for a number of them less than all the possible ones – for instance: $\mathbf{n}^{\log n}$) does not work: Even the test of $2^n - 2$ inputs on the simulation cannot help to compute the time P(n) if **S** does not halt for any simulated ones (in fact, this simulation cannot help to decide even whether **S** simply halts for a specified input from these two ones).

Moreover, even the simple question whether **S** halts for at least one input from an arbitrary nonempty proper subset of the set of all the 2^n possible inputs is undecidable, of course, by Def. 2.1. (Obs.: This question is only decidable for the set of all these possible inputs: The answer is always “True”, by Def. 2.1.)

Note again that the advice function **h(**|w|**)** cannot help enough **Q** herein too, by same reasons from Way 1.i.

Way 1.iii Q simulates the running of S only for an input (or inputs) previously computed ($\mathbf{m} = \mathbf{d} < 2^n$):

Proposition 5.2. A DTM **Q** cannot compute, without simulating the running of **S** for all the 2^n possible inputs, a nonempty proper subset of ones, where **S** halts for at least one of them, and then to simulate the running of **S** only for these inputs to compute the time P(n).

Proof. Let a well-formed string **f** be constructed with an arbitrary **n,** and let the restricted type X program **F** be below, where **Q** was, by the Turing-Church Thesis, translated into a computer program where it was included the instruction Simulated_by_Q[e] := True; just before any instruction of this program that starts the simulation of **F** for any input **e** (Simulated_by_Q is a global variable of type dynamic array or vector of Booleans values that was initialized with False in all its positions).

We call **Q’** to this program derived from **Q**. Verify that if **Q** runs in polynomial time, then **Q’** also do it, of course, and the behaviors and results from **Q’** and **Q** are the same.

```
01. F(string input) { // F is a restricted type X program, since it returns 1 for at least one …
02.    if (Simulated_by_Q[input]) do { input := “1”; } while (1 = 1); // infinite loop on d inputs
03.    else return(1); // … input of any length, since Q’ does not simulate all the $2^n$ ones
04. }
```

It is easy see that **Q**, simulating the running of **S** only for inputs previously computed (not all them, of course, since $\mathbf{d} < 2^n$), cannot compute the time P(n) of **F**, since **F** does not halt (line 02) for any input simulated by **Q**. Verify yet that **Q** cannot decide whether **F** halts for any determined input, for this problem is undecidable, by the Acceptance Problem for DTM. □

See once more that the advice function **h(**|**f**|**)** cannot help enough **Q** herein too, by similar reasons from Way 1.i. Note also that it is not possible to put information upon **h(**|**f**|**)** to help enough **Q** in order to it can decide whether **Q(f, h(**|**f**|**))** = **Q’(f, h(**|**f**|)), for example, because this decision is independent of |**f**|, clearly.

Way 2. Q does not simulate the running of S at all ($\mathbf{m} = 0$):

If the running time of **Q** depends on the one of the restricted type X program **S** into **w** for some input (where if **S** does not halt for any input, then **Q** does not halt at all, too), which occurs when **Q** acts reducing **w** into instance of another problem or simulating the running of **S**, then the use of the diagonalization method in order to demonstrate that **Q** cannot decide the problem fails, since **Q** does not have to be as restricted as **S**. But, as these running times are independent ones in the special case treated herein, where **w** is considered just a bit string, or **Q** decides whether **S** returns **1** for some input by engaging in more indirect reasoning about the code of **S**, without simulating it at all, then we can use diagonalization in order to demonstrate that **Q** cannot decide the XG-Poly-SAT. E.g., if **Q** converts a problem in **E** without $2^{n/10}$-size circuits into a PRG which fools n^c-size ones, for any fixed **c**, then **Q** is here.

Note that if the running time of **Q** is anyway always greater than one of the restricted type X program **S** into **w** for some input (where, remember, if **S** does not halt for any input, then **Q** does not halt at all, too), then **Q** is, maybe indirectly, simulating the running of **S** for this same input or reducing the instance of the XG-Poly-SAT constructed with **S** to instance of another problem, of course. In general, to reduce within polynomial time an instance of the XG-Poly-SAT is impossible, by the proof of the Proposition 5.1. We will see below that to compute that time P(n) without simulating the running of **S** – or do it in a running time upper bounded by a fixed (or even non-fixed) integer polynomial function of |**w**| – is impossible, too.

Notice yet that if the program **S** into **w** must been only one specified, fixed and known program, then the input to XG-Poly-SAT would be only **n** and we could encode the value of the time P(n) in the function **h**, where **h(**n**)** = P(n) and **Q** would have this information within polynomial [constant] time. However, **S** can be any program that complies with the Def. 2.1, and the value of the time P(n) is entirely independent of the size of **S**, thus this value is completely independent of |w|, hence is impossible that **h** helps **Q** on this computation, as proved in the Proposition below:

Proposition 5.3. Is impossible to compute within deterministic polynomial time the time P(n) of **S** for a given **n** without simulating the running of **S** at all, even with the help at no cost of a poly-bounded advice function **h**.

Proof. Assume that **Q** can, using **h**, compute and return **k** on any instance **w** of the XG-Poly-SAT, where **k** *must* be the running time of **S** into **w** for some input of length **n** (the time P(n)). Then, let **S** into **w** be the DTM (computer program) below – note that, as **Q** is supposed to be poly-time, **S** is really a *restricted type X program*:

```
01. S(input) {
02.   k := Q(⟨w, h(|w|)⟩); // Q runs on an arbitrary instance w and returns k
03.   execute (k+1000) arbitrary steps; // this diagonalization makes Q always …
04.   if (k > 8000) return(0); else return(1); // … wrong when n = length(input)
05. }
```

Thus, as it is easily seen above, the diagonalization makes that the running time of **S** into **w** is always greater than the result from **Q**, when **n** = length of the input. Hence, cannot exist a poly-time DTM that, even freely using a poly-bounded advice function, returns the running time of **S** into an arbitrary **w** on input of determined size.

Suppose, however, that **Q** could decide whether there is diagonalization into string **Q**. In this case, **Q** could stay running forever, no returning anything at all, which would imply that it would not be incorrect, because, in this case, **S** would not be *restricted type X*.

However, **S** can in general execute any arbitrary deterministic program. Hence, if **Q** can decide whether there is diagonalization into **S**, then **Q** can decide whether a given arbitrary deterministic program has a particular nontrivial behavior, which is undecidable, by the Rice's Theorem; of course, for we can easily change this behavior into a string acceptance testing to a formal language (reflect: replacing **Q** could be *any* computer program in the line 02). Hence, **Q** cannot decide whether there is diagonalization into **S**, lasting then condemned to fault: to answer incorrectly the question, by the diagonalization above (line 03). □

Perceive that to state that **Q** answers whatsoever if and only if **w** is a well-formed string does not work, because, as demonstrated in Section 3.3, ***L*** (the set of all well-formed strings) is a non-RE language, which implies that **Q** cannot decide whether **w** is a well-formed string in order to decide accordingly whether it can answer anything without mistaking. That is, in order to **Q** works in this case, it must assume absolutely that **w** is a well-formed string, and then this assumption implies that it is really true, and that **Q** for the input **w** returns within polynomial time incorrect answer, by the diagonalization above into **S**.

Observe that **h(**|w|**)** cannot do anything herein too, since the dilemma above is independent of |w|, of course.

Observe yet that if the DTM **Q** decides the XG-Poly-SAT simulating the running of **S**, then **Q** cannot run in time upper bounded by any fixed polynomial function of |**w, h**(|w|)| (in fact, none TM – DTM or NTM – that decides the XG-Poly-SAT can do it), undoubtedly, by Def. 2.1.

Conclusion:

As demonstrated above, all the four exhaustive possible ways to decide in deterministic polynomial time, even provided at no cost with poly-bounded advice function that depends only on the length of input, the XG-Poly-SAT fail: Consequently, there exists a computational decision problem that can be decided in nondeterministic polynomial time, but not in deterministic polynomial time, even provided at free computational expenses with polynomial upper bounded advice function (or set of strings) dependent only on input size, which implies **NP** $\not\subset$ **P/poly**, naturally, in our sad computational world. □

For this reason, by union of the Rice's Theorem, the diagonalization method and the complexity classes P/poly and NP, this proof is a beautiful unification and an amazing synthesis between the Computability Theory and the Computational Complexity Theory, like that one in [19].

Lastly, someone can say that if a fixed and known polynomial *p(n)* ≥ time P(n) of the program **S** into **w** is given (see this one is not deterministic poly-time computable, by Proposition 2.1 in [24], neither can be encoded into **h**, since *p(n)* is independent of |w|, clearly), then the instances of the XG-Poly-SAT can be reduced to Boolean formula ones by Cook-Levin Theorem, and then if the SAT is decidable in deterministic poly-time, then the XG-Poly-SAT is too. Big idea!

This conclusion is erroneous, however, since knowing a fixed *p(n)* ≥ that time P(n) is unnecessary to decide the problem (the universal NTM in Section 4 and the universal DTM in Way 1.i decide the XG-Poly-SAT without knowing this information (or without such an input), naturally), proving that nondeterministic computation is fundamentally much more faster than deterministic computation – even though it is freely provided with poly-bounded advice function that depends only on the size of input –, and that the brute-force search is unfortunately unavoidable in the real-world computations (I'm very sorry): To verify a correct answer is definitely very easier than find it, naturally.

5.1 Running time of the functions into programs

About running in time P(n) and time greater than P(n), let the function be:

```
Poly_Function(string input) {
    int i, counter := 0, n := length(input);
    for i := 1 to n^10 { counter := counter + 1; } // poly(n) upper bounded running time
    if (counter > 100) return(1); else return(0);
}
```

The function above evaluated at string input is just a number, naturally. But we can decide that its running time is poly(n) upper bounded, where **n** = |input|. We don't need a TM to decide it. On the other hand, let the function be:

```
SuperPoly_Function(string input)
{
   int i, counter := 0, n := length(input);
   for i := 1 to 2^n { counter := counter + 1; } // exp(n) upper bounded running time
   if (counter > 100) return(1); else return(0);
}
```

Of course, the running time of the function above is exponential in **n**. We know countless functions as the ones above [2] to use them in order to make restricted type X programs. Constructing restricted type X programs using algorithms with known running time is human work, not TM computation [2].

5.2 Example of construction of an instance of the XG-Poly-SAT

Let the restricted type X program **S** be:

```
S(string input)
{
   remainder := mod(integer(input), 2);     // remainder on division of input (converted into
                                            // integer) by 2
   if (remainder = 0) return(Fun2(input)); // returns the value returned by Fun2 and halts
   if (remainder = 1) return(Fun1(input)); // never halts
}

Fun1(string input)
{
   do { input := "1"; } while (1 = 1); // infinite loop
   return(1);
}

Fun2(string input)
{
   int i, counter := 0, n := length(input);
   for i := 1 to n^10 { counter := counter + 1; } // poly(n) upper bounded running time
   if (counter > 0) return(1); else return(0);
}
```

Thus, we can simply convert this program **S** into a DTM ***M***, translate it into a binary form **s**, and then construct the well-formed string **w** = **111111110s**, an instance of the XG-Poly-SAT. Herein, constructing XG-Poly-SAT instances, it stands very clear that the human reasoning is much more powerful than mechanical (TM) computation.

6. Baker-Gill-Solovay Theorem and the Proof

Verify that the proof does not use the diagonalization method (except in the justified special cases in Section 5) and it is based about the difference, on worst cases, between running times from a DTM and an NTM that recognize the L_z-language ***L***, as demonstrated in Way 1.i of Section 5 compared to Section 4.

Moreover, notice that the addition into the proof methods of oracles to a PSPACE-Complete language *W* does not imply that false statement $P^W \neq NP^W$ (because the proof cannot be adapted to demonstrate that $P^W \neq NP^W$, since a DTM **Q** with an oracle to *W* could simulate any NTM with the same oracle using only a poly(n)-quantity of space, in an adapted Way 1.i, which would otherwise prove that $P^W = NP^W$).

These facts imply that the Baker-Gill-Solovay Theorem of inseparability of the classes P and NP by oracle-invariant methods (techniques that are conserved under the addition of oracles, like the pure diagonalization method without *algebraic oracle* [8]) does not refute this NP $\not\subset$ P/poly proof. In other words, my proof technique does not *relativize* [4].

7. Razborov-Rudich Theorem and the Proof

SAT's weakness – The proof does not try to prove any lower bounds on the circuit complexity of a Boolean function, because it does not try to solve the still open question whether SAT is in P, since to prove NP $\not\subset$ P/poly it was not necessary to solve the SAT question (for the proof, different from the wrong conclusion in [3, 6], it is irrelevant whether SAT is in P), whereas it was enough to prove that XG-Poly-SAT is in NP but not in P: Thus, the Razborov-Rudich Theorem of the Natural Proofs does not refute this proof. In other words, my proof technique does not *naturalize* [7].

8. Related Work, Aaronson-Wigderson Theorem and the Proof

There is no relevant related work on the goal to really solve the NP versus P/poly question. From important papers upon the matter, there are only some “negative” results, like the ones referred to in Sections 6 and 7 and, more recently, as an extension of the *relativization* in Section 6, the proof that techniques that are conserved under the addition of an oracle and a low-degree extension of it over a finite field or ring cannot work on this question too, by the concept of *algebrization*, explained in [8].

Remember, however, that my proof does not use the pure diagonalization method (as referred to in Section 6), but it exploits properties of computation that are specific to real world computers, and then this new barrier is not valid to refute it, too.

9. Expert Advice & Academic Honesty

A reviewer, referring to the technical report in [15], has said “– It is disconcerting to see how the present author continues to ignore expert advice. His title borders on, and perhaps transgresses, academic honesty. Papers with such grandiose claims should only be considered after an endorsement by an expert.”

The heart of my paper is just challenging some traditional definitions on TCS field, essentially the need of poly-uniformity on the definitions of the classes P/poly and NP. But that technical report says, for instance: “– As ... Definition 3.5 of his paper ... needs to before the universal quantification on *x* fix a polynomial bounding the length of the certificates, we from here on assume that his definition is viewed as being modified to do that ...”

So, as my proposed new definitions are so distorted in that expert advice, it has very low value in order to evaluate my proof, thus ignoring it is not really academic dishonesty at all, but only logical consequence of that challenge upon enhancing those definitions.

10. Freedom & Mathematics

"**– The essence of Mathematics is Freedom.**" (Georg Cantor) [23]

11. References

[1] J. E. Hopcroft, J. D. Ullman, and R. Motwani, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading MA, 2001.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (Second Edition)*, The Mit Press, Cambridge MA, 2001.

[3] K. J. Devlin, *The Millennium Problems: The Seven Greatest Unsolved Mathematical Puzzles of Our Time*, Basic Books, New York NY, 2002.

[4] M. Sipser, *Introduction to the Theory of Computation – Second Edition*, Thomson Course Technology, Boston MA, 2006.

[5] Adapted from the appendix of the paper *Uniformly Hard Sets* by L. Fortnow and R. Downey, unpublished, available: http://weblog.fortnow.com/media/ladner.pdf

[6] Cook S.A., "*P versus NP problem*", unpublished, available: http://www.claymath.org/millennium/P_vs_NP/Official_Problem_Description.pdf

[7] From Wikipedia, the free encyclopedia, "*Natural Proof*", unpublished, available: http://en.wikipedia.org/wiki/Natural_proof

[8] S. Aaronson and A. Wigderson, *Algebrization: A New Barrier in Complexity Theory*, Electronic Colloquium on Computational Complexity, Report No. 5 (2008), available: http://eccc.hpi-web.de/eccc-reports/2008/TR08-005/Paper.pdf

[9] O. Goldreich, *On Promise Problems (in memory of Shimon Even (1935-2004))*, unpublished, available: http://www.wisdom.weizmann.ac.il/~oded/PS/prpr.ps

[10] M. Sipser, Cambridge MA 02139, in *The History and Status of the P Versus NP Question*, p. 606, unpublished, available: http://www.seas.harvard.edu/courses/cs121/handouts/sipser-pvsnp.pdf

[11] From Wikipedia, the free encyclopedia, "*Rice's Theorem*", unpublished, available: http://en.wikipedia.org/wiki/Rice's_theorem

[12] O. Goldreich, *Notes on Levin's Theory of Average-Case Complexity*, unpublished, available: http://www.wisdom.weizmann.ac.il/~oded/COL/lnd.pdf

[13] A. L. Barbosa, *P != RP Proof*, unpublished, available: http://www.andrebarbosa.eti.br/P_different_RP_Proof_Eng.pdf

[14] From Wikipedia, the free encyclopedia, "*P versus NP Problem*", unpublished, available: http://en.wikipedia.org/wiki/P_versus_NP_problem

[15] L. A. Hemaspaandra, K. Murray, and X. Tang, *Barbosa, Uniform Polynomial Time Bounds, and Promises*, Technical Report, unpublished, available:

http://arxiv.org/abs/1106.1150

[16] From Wikipedia, the free encyclopedia, "*Decision Problem*", unpublished, available: http://en.wikipedia.org/wiki/Decision_problem

[17] T. S. Kuhn, *The Structure of Scientific Revolutions*, University of Chicago Press, Chicago IL, 1962.

[18] From StackExchange (cstheory), cc-wiki, "*Are runtime bounds in P decidable? (answer: no)*", unpublished, available: http://cstheory.stackexchange.com/questions/5004/are-runtime-bounds-in-p-decidable-answer-no

[19] A. L. Barbosa, *P != NP Proof*, unpublished, available: http://arxiv.org/ftp/arxiv/papers/0907/0907.3965.pdf

[20] From Computational Complexity, blog, "*P/poly*", posted at September 07, 2005, by L. Fortnow, unpublished, available: http://blog.computationalcomplexity.org/2005/09/ppoly.html

[21] From Wikipedia, the free encyclopedia, "*Zermelo-Fraenkel Set Theory*", unpublished, available: http://en.wikipedia.org/wiki/Zermelo-Fraenkel_set_theory

[22] From Wikipedia, the free encyclopedia, "*Constructible Function*", unpublished, available: http://en.wikipedia.org/wiki/Constructible_function

[23] From The Engines of Our Ingenuity, site, "*Episode nº 1484: GEORG CANTOR*", posted by John H. Lienhard, unpublished, available: http://www.uh.edu/engines/epi1484.htm

[24] A. L. Barbosa, *The Cook-Levin Theorem is False*, unpublished, available: http://www.andrebarbosa.eti.br/The_Cook-Levin_Theorem_is_False.pdf

[25] From Gödel's Lost Letter and P=NP, a personal view of the theory of computation, blog, public comments on "*Facts No One Really Checks*", posted at July 25, 2012, by R. J. Lipton, unpublished, available: http://rjlipton.wordpress.com/2012/07/25/facts-no-one-really-checks/#comment-22187

[26] A. L. Barbosa, *What is the Size of the Hilbert Hotel's Computer?*, unpublished, available: http://www.andrebarbosa.eti.br/The_Size_of_the_Hilbert_Hotel_Computer.pdf

[27] A. L. Barbosa, *The Randomness Delusion*, unpublished, available: http://www.andrebarbosa.eti.br/The_Randomness_Delusion.pdf

[28] A. L. Barbosa, *The Dead Cryptographers Society Problem*, unpublished, available: http://arxiv.org/ftp/arxiv/papers/1501/1501.03872.pdf

[29] J. Abascal and S. Maimon, *Critique of Barbosa's "P != NP Proof"*, unpublished, available: https://arxiv.org/pdf/1711.07132.pdf

André Luiz Barbosa – Goiânia - GO, Brazil – e-Mail: webmaster@andrebarbosa.eti.br – August 2011

Site....….. : www.andrebarbosa.eti.br
Blog....….. : blog.andrebarbosa.eti.br

This Paper : http://www.andrebarbosa.eti.br/NP_is_not_in_P-Poly_Proof_Eng.htm
PDF…..… : http://www.andrebarbosa.eti.br/NP_is_not_in_P-Poly_Proof_Eng.pdf