Under consideration for publication in J. Functional Programming

TQBF in P demonstration and other methods

Juan Manuel Dato Ruiz
Ex-consultant of the data protection law in Educamurcia Las Torres (Spain)
(*e-mail:* jumadaru@gmail.com)

Abstract

Scientific language, as well as engineering itself, uses declarative notations to try to reach conclusions about the regression of data. The only way to work with huge amounts of data is to ensure that algorithms do not exceed their complexity beyond the polynomial dimension. This document will constructively show how the quantification of Boolean formulas falls within this class and will open the door to new fields of research in modular operations management. This document will also present strategies and ways of working to achieve objectives that are beyond the possibilities that are also shown to exist.

Contents

Introduction

Society needs to democratize itself. The technology, little by little, due to the preponderance of patents is becoming more expensive and, therefore, more exclusive of corporations that are able to invest in libraries and components. Little by little we see how even scientists need machines that have been designed to be paid for by state funds, at the very least. It also occurs with the publication of works in Journals. Increasingly, it is becoming more and more important to have to rush the entries to accumulate in contributions what may fit within the limitations imposed by the Journal itself. In any case, if you want to offer the work free of charge, it is clear that someone will have to pay for this management; therefore, not only did the researcher dedicate his or her efforts, but he or she will also have to pay for the disclosure of his or her work.

One of the most fundamental problems, to the extent that most people do not know how much it could affect them in their daily lives, is the comparison of classes P and NP. It may always seem presumptuous on the part of the one who studies a field to emphasize a work more than it actually has; however, in this case it is not possible to exaggerate, since the same process of inventiveness of new problems and applications depends on the understanding of the problem in question. That is, to what extent a problem can be useful, frecuently often it has to do with understanding the problem itself.

You can solve a problem that requires thousands of years of maturity in a few minutes, but the only reason why people will have knowledge of this tool is because the most experienced scientists have given it the go-ahead. That is to say, because in advance someone agreed not to have arrived first in the race to solve that; what expert on this subject has not wanted to try it? Well, it wasn't only going to be difficult for a couple of them to agree because they were experts in the field, the main problem might be that the solution could be to rewrite the wording of the problem, which would destroy the work of 99% of the essays that have been written.

Seen in this way, it is very difficult to publish the correct solution in a Journal if necessary. It is in the opinion of this author that, indeed, it happens. A judgement that is made after more than a decade of trying to make public a result that is beneficial for absolutely everyone and which, beyond the gratuitousness that some repository can provide, I was also surprised how the author's personal defects have never been complemented by the professional work of the person reviewing it but, rather, it seemed that it was used as an excuse to back down any attempt at disclosure.

It is like saying, either you are perfect and resolute, or you are imperfect and unsolvable; but don't think about questioning the work of scientific consensus from imperfection. It is unsightly to tell an expert that he or she has been overtaken by a stranger who makes spelling mistakes or does not submit his or her documentation to a spelling check program, that he or she may have significant errors in number theory when performing self-checking steps to find contradictions where there are none or that he or she skips intermediate steps when explaining them.... and no one realizes that all three aspects are exactly the same pathology, which affects language and, at the same time, does not affect understanding. It sounds ironic to think that precisely what I have called logical crystallization is going to be exposed in this document: that it is the foundation behind the capacity of a machine

to justify or explain what it already knows (without making use of Robinson's unification algorithm or derivatives).

Even so, the machines that will be presented in this document do not have to have all the relevance that I intend. I also don't know how many machines it might be interesting or relevant to put here. To what extent the reader will be willing to read and accept, to see for themselves and to tolerate the different ways of doing things well. This is a second irony in this strange story: this document tries to talk about efficiency, something that seems forbidden. In the exercise of doing things right, we will see how even an aphasian is able to offer a healthy perspective, as long as he or she is given the opportunity to objectively evaluate his work beyond its most notorious superfluous deficiencies which, if uncorrected, we will attest that there will be many of them.

Introduction to the methods.

A Turing Machine is a notation that allows us to represent the enunciation of a numberable problem in a way that simplifies its resolution. The notation, as it is so powerful, also admits statements that will not be able to finish simplifying; likewise, it is possible to pose problems that exceed the ability to be codifiable: an example would be the problem of knowing if a certain configuration for the Turing Machine is susceptible to never stop for some input. This last result marks the profile of the limits of symbolic notation: it allows us to solve concrete problems and to have more solutions than certainties.

Faced with the exact notation offered by Turing Machines, or its equivalent: Church's lambda calculus; as well as the programming languages that equate it with language potentials, we have the option of developing empirical techniques that return context-sensitive results - as well as more holistic ones. This other philosophy is not symbolic. It consists of transforming the life cycle time of the machine that the user has to use in a time that is part of the programmer's work. This overload requires less static programming, which means that the code has to be parameterized much more by means of models or generalizations, in order to start the so-called training process. This training process will end up remaining persistent in the form of data that the user could perfect in something more representative. These techniques appeared along with the first neural networks, as well as the first cellular automatons, etc... These techniques have been called non-symbolic because they depend mainly on the behavior of the system perceived through the data it handles.

In short, today it is easy to understand the difference between symbolic and non-symbolic programming because we are experiencing a new technological turning point (considering in this essay (Hummel, 2010), there are some applications out of the scope of the symbolic programming). If the problem is to introduce robots into the world of humans, then symbolic programming can do nothing else except to constitute the robot's ethics, while non-symbolic programming can never go beyond morality. Certainly, it is not a trivial matter: ethics is not the same as morality. Ethics is analytical, and we cannot be sure that its values cover all our problems. While morality is a product of human experience, and we cannot be sure that it is coherent. In order to understand it better, one only has to look at the laws: they are a reflection of morality that has been adjusted to the ethical principles recognized in a society. Laws will always be limited by those acts that are considered amoral: acts that are not absent from ethics, from the coherence of the Principles, but that could be morally

reprehensible because they are not expressly contemplated by laws that require a certain degree of vigilance. Let's say that this same complexity is what happens with mathematics and the problem of knowing if NP is different from P, as we will break it down later.

The Turing Machine was designed and perfectly met its objective of knowing what notation would be enough to represent any problem that could be coded (is presented in (Turing, 1937) for that purpose). That is, it solved the problem of completeness. However, later on, the problem of efficiency begins to arise, if things are done well, what does it mean to do things well and if it differs a lot from doing them correctly?

If we focus on the technical slang used by the computer engineer, and avoid everyday language, doing things correctly is very different from doing them well. The correct thing is to comply with each specification. If we could mark each specification with a cross or a circle to assess whether it has been carried out or not, a system is correct when each of its specifications are met in the system. However, a system is efficient when it makes good use of resources: when things are well done, regardless of compliance with specifications. Without going any further, in this document we will see how we return to the idea that you can do things that are not correct, but can sometimes be efficient (follow your intuition, as calculated by the machine). In this way it is easy to understand that the machines that are efficient in moral precepts but do not comply correctly with what is expected could be considered amoral in practice: like a car that decides to kill a pedestrian going out of its lane to avoid some type of greater evil deduced by some custom that was not submitted to the judgment of ethics or some type of coherence, for example, submitted to its own judicial jurisprudence or law previously approved.

This will help to understand how far the problems of the two classes go. Now, well defined, we will say that Turing's Machine notation describes what to do with the symbols that have been recorded on a tape, which we will call "input". The symbols, chosen from a finite alphabet, will each occupy a cell and the machine will move its pointer sequentially to the cell on the right or left, being able to modify the symbol it reads by virtue of which symbol it is currently reading and in what state the machine is in.

The machine will then modify the status, the pointer position and the symbol at the input or maintain some of these values. If the description of what the next state is perfectly determined by the configuration, what the pointer does and what symbol is provided, then we will say that the Turing Machine is deterministic.

If we use special symbols that induce us to assume different possible following states or actions, then we will say that the machine configuration is not deterministic. Since the options behind a non-deterministic machine are always subject to an explicit set of finite elements, we are sure that the non-deterministic machine will not succumb to possible paradoxes relative to infinity, since the number of deterministic machines required for each entry will always be a potentially finite number.

The thing is all the problems that allow us to have a resolution in a deterministic Turing Machine after giving an always smaller number of steps than a polynomial expression (whose variable is the size of the entry) we are going to classify them within the same class: the class P. Anyone that gives a definition notoriously different to this one will be committing fallacies.

In the same way, it is easy to think that if the non-deterministic machine succeeds in solving its configuration within this enormous number of Turing Machines simulated also

within the polynomial boundary, then such problems are called *NP* (Non deterministic polynomial).

However, at this juncture, it is important to point out other definitions that have been tried to apply towards the problems contained in the *NP* class. It has come to imply that a problem is in the *NP* class if the solution of the problem is easy to verify and understand that its verification would be within the polynomial boundary.

Here a mistake is made, and of particular gravity, if one considers everything previously said in this section: a machine could not be doing the right thing but still give an easy verification of its solution.

Specifically, we have the diophantine equations: all diophantine equations are verifiable in polynomial time and, at the same time, Matijasevich demonstrated that there is no Turing Machine configuration to solve all of them (in (Matijasevich, 1992) explains the latter in more detail).

In other words, its number of steps cannot be limited by the worst of the entries within a polynomial boundary. Be that as it may, if the object is to hack into the definition, it is better to know how to talk a little bit properly: beforehand, so that a problem is considered within the *NP* class before it is essential that the problem be codifiable/numberable.

Well, asking if the *P* class equals the *NP* class is like wondering if it is possible to declare problems in a declaratory way so that the compiler itself can then auto-generate the code that would be needed for the machine to solve the problem efficiently.

In short, that and no other is the master question. However, this document will show how the question can be circumvented in order to give the two opposite answers, one for each philosophy: symbolic and connectionist. However, considering that the symbolic philosophy is harder than the connectionist one, it can also be said that the answer is "no, although...".

Later on we will see how we will modify the master question until it matches the title of the essay. After all, such demonstrations have already been exposed at the time.

However, the fundamental problem of the dual response to the problem must be emphasised: we know that the distance of the last ribbon element on a Turing Machine from the position it occupied at the beginning will be polynomially limited within the *P* and *NP* classes; in the sense that the machine runs the tape sequentially and, therefore, could never run an undefined number of cells in a limited amount of time (trivially *NPSPACE*, problems that polynomially limit the tape size, $NP \subseteq NPSPACE$, detailed in (Arora and Barak, 2016)). However, both the alphabet size and the number of internal machine states do not depend on the input size, or the input itself, or the number of steps.

This means that our Turing Machine could be configured with exponential quantities of internal states, or alphabets with an explosive number of symbols, as long as such quantities are finite and constant.

This is a detail that will make its study important: a machine that limits its time to conclude, 'does it need an alphabet or a number of non-limited states?' Certainly, for machines that do not fully meet expectations, perhaps the answer is yes. And these machines will still be efficient.

About $NP = P$ in some Turing Machines applications.

To better understand what has just been said, we will define a problem: $SAT3$, as it is known. Given a set V of finite n boolean variables $V_0, \cdots, V_{n-1}$, we define a literal as the variable or its negation, we define a clause as the Boolean addition of three literals and a formula $SAT3 \sim f(V_0, \cdots, V_{n-1})$ will be the Boolean product of clauses. From where the $SAT3$ problem is to determine if there is a σ Boolean function where

$$f(\sigma(V_0), \cdots, \sigma(V_{n-1})) = 1$$

.

To determine if this is an NP problem, the first thing to check is whether it is feasible to find a configuration that can always provide a solution. As the problem limits the variables to only two values, we will understand that along the entrance some of the Turing machines will only need to propose a solution, by specifying a finite amount of Turing machines, and these give their solution within the polynomial boundary, we will be able to accept that problem within NP.

Now let's see if, in addition, we are able to generate a deterministic machine within the same bounds. To do this, we will consider that the entry will be each of the clauses, our alphabet will be formed by how each clause is coded (if a literal can appear in the clause denied, affirmed or not then, having n variables, are n variations of 3 possibilities with repetition: 3^n, which is independent of the size of the entry) and, therefore, the size of the entry will be the number of clauses.

The internal states will be the parts of the alphabet (2^{3^n}, also independent of the input - constant). It begins in the universal state of the alphabet, which accepts all cases, and as a cell is read, the parts of the alphabet will be restricted to the coding of all the alphabet symbols that are compatible with each of the cells that have been read so far.

In this way, after taking as many steps as the size of the input, a final status is obtained which, when reading the special symbol of completion, will proceed to encode the number that has this status foreseen. The number is a value that does not depend on the input and its printing can be done within the coding system of the alphabet within the polynomial boundary. The number to be printed will be the number of alphabet symbols that are still compatible with all read entries, which corresponds to the number of cases that satisfy the formula.

As can be seen, not only has it been returned in linear time if there is more than one case, but it has also been possible to return the number of cases. So we can say that not only is it proved that $SAT3 \in P$ but also that the problem of knowing how many cases a formula contains within $SAT3$ is also P, which is denoted as: $\#SAT3 \in P$.

The trick we have used to do something like this is to encode the alphabet and the states without a limit of polynomial dimension. And of course, we'll say that it's a trick for a simple reason: when it comes to making this machine practical, that is to say, when using this algorithm we'll see that it gives us some problems of construction. In other words, we have made a discovery but it does not seem innovative. Well, at least at first: these techniques require an even deeper study to understand their usefulness. In any case, we can say that this configuration is equivalent to saying that the Turing machine is like an instruction manual: where the number of pages is set by the number of states and the lines

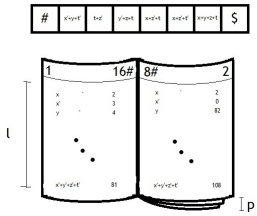

Figure 1. Variables: 4, Lines: $l = 3^4$ and pages: $p = 2^l$

for each page are set by the alphabet, while the number of times a new page is passed to a new page would be limited by the size of the entry.

Understand correctly the idea of a Turing machine as an instruction manual: we are saying that for each problem defined with UP to a certain number of variables there will always be a manual that solves it after a number of steps equal to the number of clauses. So if we are getting warmed up, perhaps it is time to consider what demonstrations work without specifying this nuance to the point of asserting such imprecise ideas as seeing the Turing machine as an instruction manual. In other words, seeing Turing's machines in this way means not accepting symbolic philosophy or, in other words, constructivism. However, we must not deny that there may be machines that can be functional from this point of view.

In short: the formal Turing machine is the one that accepts the previous solution, if the existence of a Turing machine cannot be falsified. However, the constructivist Turing machine requires that it must be expressly demonstrated that its coding must be enumerated: let's assume that the input is defined in the problem itself in order to increase the rigour of what solution we expect.

This rigour can lead us to define the $SAT3^*$ problem with an alphabet prepared to encode variables throughout the entry and some special "clause change" symbol that, if read two consecutive times, can be interpreted as the end of the entry.

As can be seen in Figure 1, formally there is a MTD which solves some $\#NP$ problems in P up to any number of variables.

With this way of expressing the problem we observe beforehand that $\#SAT3^*$ can no longer be defined within the P class as before because there would be no way to encode the internal states from a number of variables depending on how large the input is.

In this way of working we will say that it will be constructive, because it transcends formal language. This other version of Turing Machine can be seen as a gear machine: because the gears, the entropy it solves, will always be internally limited - at risk of breaking the laws of thermodynamics.

As the instruction manuals can be seen, not as thermodynamic systems, they can work without taking into account the volume, the number of pages (the dimension that does not depend on either the temperature or the number of microstates to generate entropy); this is what makes it impossible to build a physical machine, but ideas can be maintained under a connectionist philosophy that associate the system with practical ideas.

About theorem of Cook.

At this point, 99% of the experts will have despised this work (according to this recognized book (Garey and Johnson, 2009)). They may not want to accept the above formula, or they will be eager to know what happens to Cook's theorem.

Cook's theorem, which is the mother of the lamb for 99% of those who study these classes, prays: Given any problem π_0 expressed in an L language that ranks within the NP class, there will always be a way to build a formula within $SAT3$ on a tape in a Turing Machine after a polynomial number of steps so that the solution to the $SAT3$ problem matches the solution to the π_0 problem.

Note the most fundamental criticism: Cook's theorem tries to establish a topological equivalence from some L language. We do not start from a specific format for the tape. So, if we study the demonstration we see that, indeed, at a given moment it shows that the demonstration is only valid for the formal $SAT3$, but not for $SAT3^*$. Specifically, we can say this because it theorizes about the existence of a machine that already limits the problem without specifying how it transforms from L to the tape, that is: how the alphabet and internal states are configured.

By ignoring this part, we cannot say that the demonstration is constructive. In fact, if it were constructive, after 40 years, someone would have already written the code that compiles that equivalence from that L language that absolutely no one speaks or knows. In other words, the theorem is formal because it does not concretize anything.

On the other hand, this theorem loses all its value because, within the scope shown in the previous section we see that directly $\#SAT3 \in P$, so we can formally use this theorem to say that $\#P$ (to know how many cases meet the restriction of an NP problem) are actually in P. That is, Cook's theorem will be used in a valid way not to talk about the existence of NP-Complete, a concept that is no longer useful, but to ensure topological equivalence formally (considering the perspective of Cook in (Cook, 2003)).

What innovative is. Applications in MT^S vs. MT^F.

Let's start by distinguishing the two types of machines, the symbolic or constructivist (MT^S: gears) and the formal or connectionist (MT^F: instruction manual). Each machine offers a different utility and is therefore oriented to different types of users. So one user may not be interested in the results of one machine when the other is interested. It has been read, and seems to me to be imprecise definitions, strong artificial intelligence (MT^S in this document) and weak artificial intelligence (MT^F in this document). In any case, this section will explain in more detail what is expected of each one (as Shor did considering Quantum Programming different in (Shor, 1999)).

The MT^F has as method the neural networks, the Boltzman machine (which is ironic, because these machines are the ones that ignore the entropy formulas of Clasius and Boltzman), quantum programming and therefore..., focuses on morality programming, validation and statistics in general. His work has a certain holistic character and is very much based on perception and probable experience. This can help set realistic limits to cryptography, considering that by approximation and brute force many codes can be broken.

The MT^S has as a method the databases and semantic networks, uses digital systems and can be used to evaluate the code of ethics or verify requirements. His work has a certain analytical character and is based on the rigour of compliance with the models. It also establishes the limits of cryptanalysis.

It is possible to combine both machines, as long as they do not mix too much, because at the same time that they complement each other they also get in the way. It's not easy to know where to draw the line between them. To understand that the combination is possible, it is only necessary to look at the enormous similarities between neural networks and semantic networks, so if MT^S is in charge of controlling and configuring a state within the MT^F then we could speak of a stable information system. This same document will show the difference between what we can call logical calculation and intuition (analogical); functions that have been considered mental but can be rigorously defined in order to work such techniques within engineering to be innovative.

There was no shortage of the pair asserting that it was unnecessary to include within the jargon of engineering and mathematical theorems concepts such as ethics, ontology, etc..., and that the theory of graphs was more than enough to encompass in its notation all that it needed.

My answer, which I can only give here because of the impossibility of giving an answer to those who don't want to read it because they don't even ask for it, is that computer science is being used to accommodate this type of problem, and studies on join dependencies tell us that we need to go through several standardizations before we can get a network to make enough compromises to represent the simplest structures with functional multi-dependencies. This is why this documentation will emphasize what has been called "logical crystallization". This consists of preparing the instances to comply with a relationship without generating duplication when storing the data. Because, indeed, a mathematician may believe that the graph theory is very complete; but the computer scientist is the one who actually works the efficiency of the databases - some should recognize the field work.

As a practical example of the fact that these studies have a tangible background, we should highlight the special importance of the problems that are classified within the P class for an MT^S. More specifically, when there is a database with very little information available, there is already some trial that has cited the fact that dynamic machines (which work with microinstructions to produce chain instructions in mass within the computer) must resemble what has been called non-determined and random Turing Machines. These machines move the pointer, rather than sequentially from cell to cell, to specific positions of the tape in a single action. In addition, they wanted to add an additional operator (such as multiplication) to simulate the machine's times with the way current computers behave.

This established accuracy has several weak points because, first of all, it is not true that today's computers can position their program counter at any memory location, or that it can be routed to any memory location: the more addresses can be routed, the more page errors it can cause.

So their theoretical machine would have in this sense a much more powerful behaviour than the real machine - besides that the theoretical machine would have unlimited memory. And, on the other hand, the theoretical one understands an indefinite number of processes that work in parallel, and of course...The parallelism of the processes running on today's

computers is limited by the number of processors, machine segmentation, read and write hazards, structural control hazards and, of course, accessory mechanisms implemented in the operating system such as parallel access to devices or the implementation of some kind of multitasking within a distributed network system. The parallelism of the processes running on today's computers is limited by the number of processors, machine segmentation, read and write hazards, structural control hazards and, of course, accessory mechanisms implemented in the operating system such as parallel access to devices or the implementation of some kind of multitasking within a distributed network system.

All of these limitations make the theoretical machine work much better and it doesn't help predict how to create a technology that already exists.

Even so, my biggest criticism of that equivalence is that all the implementations mentioned in the previous paragraph to achieve parallelism are only capable of improving runtime in a constant way; therefore, for problems whose input size is small, these processors will behave excellently. However, it starts to be interesting when we start talking about huge databases, the calculation of gigantic numbers or processing like that which will require a more than respectable length entry.

It is then that it is time to rediscover the meaning of these studies. Because if we have a very small company that has few clients and is managed with few statistics, and if these were easy to normalize, then we would see how the studies of complexity quoted here are not as relevant; having a supercomputer would be more than enough.

However, what happens when there is either a huge database, or a gigantic number, or a statistic where the information is matched by gathering units making it difficult to standardize, etc.? Then we will see how the results obtained start to become desperate if we only pay attention to hardware techniques. That is to say, with the arrival of Big Data (management of immense databases), it becomes interesting which problems are likely to be solved and, to achieve approximations, which are the structural limitations of such approaches.

Methods

The fundamental feature that differentiates the MT^S from the MT^F is the way to configure the MT: in MT^S it is independent of the input, while the MT^F, even if it is not expressly stated, can modify its coding by virtue of the input. This means that the latter, because they are less rigorous, are considered of greater interest to experts who deal with statistics and artificial intelligence, because they use techniques that although they do not succeed in consolidating themselves as a philosophy that serves the guild, it is true that the way they work allows them more sources of funding. That's why the MT^F are more behavioral based, and allow more configurations, and have better press because they are more foreign to computing and more linked to mathematics as we will see later. However, in this section we are going to focus on the MT^S and we will see its possible most evident limitation.

Demonstration $NP \neq P$ *in* MT^S

To begin with, we will emphasize a recognized notation that will later become necessary: the idea of dependency between variables. More specifically, we will say that when you

have an expression of the form $\forall x \forall y : P(x,y)$ we will say that x and y are independent from each other and, in fact, take any value without exception within their dominion. The same will happen when we find an expression of the type: $\exists x \exists y : P(x,y)$, which tells us that some value of x and some value of y are given, but both are completely independent of each other. Things change if we have an expression of the way $\forall x \exists y : P(x,y)$, which will mean that y can be expressed or not depending on the variable x; it would be necessary to think how many functions within the x dominion exist so that y can be any of those expressions; if these variables were boolean then the number of expressions will always be finite for a finite number of variables (in this case $f(y) \in \{0,1,x,\neg x\}$ because it depends on a single variable, if it depends on V boolean variables it is trivial to show that the possible expressions would be 2^M with $M = 2^V$, where M is the number of minterms). On the other hand, the expression $\exists x forally : P(x,y)$ has x and y independent, because in this notation what matters is the order.

However, it is not true that this notation is enough to represent (2^M) all the possible independence between variables. Therefore, it may be interesting to include propositions of the type $I(K,x)$:

Definition 1
$I(K,x)$ is a boolean formula: $B \times B \mapsto B$, and α any formula which depends of K and x:

$$\forall x, \exists K : I(K,x) \wedge \alpha \iff \exists K, \forall x : I(K,x) \wedge \alpha \tag{1}$$

where K is an existential variable and x is a universal variable. So we say that K, in its expression will never include the variable x; or we will also say K is independent of x.

Now yes, with these notions, we can already make a more formal distinction between what an MT^S is and a MT^F. As stated at the beginning, the MT^S has a configuration that is static, always the same, and a tape where it stores dynamic information: that can be the input symbols, working values, special symbols and output.

This means that under a constructivist (symbolic) philosophy, the γ function we use to configure a Turing machine will depend on the L language in which the problem is expressed, but not on the input of the problem. So, if we say so: given a π problem that from an input E hopes to return an R answer, that could be expressed $\exists \pi, \forall E, \exists R : \pi[E] = R$. And if we say we can configure a MT with a γ configuration that solves π, then it can be expressed like this:

$$\exists \pi : \forall \gamma_0 \exists \gamma \forall E \exists R : MTG_{\gamma_0}(\pi) = \gamma \wedge MT_\gamma(E) = R$$

Where MTG is the general Turing machine (as was presented in (Turing, 1937)) that, from a L language is expressed a problem in its tape to generate as a result a MT configuration. Therefore, we understand that γ_0 represents an encoding configuration in general, while γ represents a configuration that solves a particular problem. So, to reduce the notation to something more manageable, this other way of expressing it will be used:

$$\forall \gamma_0 \exists \gamma \forall E \exists R : \gamma(E) = R$$

And we will omit that the γ refers to coding methods either of MT or of less general machines.

Now we will appreciate the following nuance:

$$\gamma_S \sim MT^S \Longrightarrow \forall\gamma_0\exists\gamma_S\forall E\exists R : \gamma_S(E) = R \tag{2}$$

$$\gamma_F \sim MT^F \Longrightarrow \forall\gamma_0\forall E\exists\gamma_F\exists R : \gamma_F(E) = R \tag{3}$$

At this point, we will proceed to propose a NP problem for a MT and then we will demonstrate that it does not have possible encoding for MT^S.

Let's imagine that a digital signature mechanism has been invented through a MTG where starting from γ_1, expressible function in a MT deterministic and limited within the FP class (returns its results within the polynomial boundary explained in (Rich, 2008)), and a E entry; the signature system $\phi_{\gamma_1}(E)$ return $E \oplus \gamma_0(\gamma_1)$, γ_0 is any codification system bounded polinomialy. Where the $\oplus$ operator collects two arguments, converts them into binary integers as encoded and calculates bit by bit an XOR operation to constitute an integer.

Note that if $\phi_{\gamma_1}(E)$ returns another additional value, such as $\gamma_1(E)$, then it uses both values to form a digital signature, as long as there is no γ_2 such as: $\gamma_2(E \oplus \gamma_0(\gamma_1)) = E$. Now, the first thing we're going to check is that there will be a non-deterministic MT because, for a X entry, it's about finding the E argument where $\phi_{\gamma_1}(E) = X$. As the problem is defined throughout its own dominion to the point of ensuring that the verification response time will always be polynomially delimited, then we can already define the problem that encodes to γ_2 within the NP class.

Now let us proceed to express it as a MT^S:

$$\forall\gamma_0\exists\gamma_2\forall\gamma_1\forall E\exists\phi : \phi_{\gamma_1}(E) = E \oplus \gamma_0(\gamma_1) \Longrightarrow \gamma_2(\phi_{\gamma_1}(E)) = E$$

This simple expression can be further reduced to show that it is absurd, for example, since any code γ_0 is in P, we could easily find some γ_1 where $\gamma_0(\gamma_1) = 0$ and, therefore, we have $\forall E : \phi_{\gamma_1}(E) = E$, or, which is the same thing, $\gamma_2 = \lambda X : X$. This will lead to a conflicting situation for each of the different values that could have γ_1 so that the same γ_2 function can be coded independently. Therefore: There are problems in the NP class that cannot be solved with a MT configuration that is independent of the tape. That's $NP \neq P$ in MT^S.

One has to understand the repercussion of this result: we can create problems within the NP class so if we try to create a static configuration that is not coded from the entry, we will see that it is not possible. In other words, we offer a program to solve the problem, and then the user tells us that what we have configured is not applicable in some cases; this type of answers is not compatible with a work within symbolic philosophy.

Impulse class exhibition. Reasons that make linearity a useful mechanism.

Up to this point, we have verified how inconsistent it is to wait to find a coding subject to an input-dependent measurement, starting from a code independent of it when it comes to calculating the inverse of any injective function. We see that the static configuration cannot be subjected to any kind of measurements.

Given a collection of elements, what must be accomplished so that the creation of a logarithmic amount of elements relative to the size of the collection describes the behavior of the entire collection? The way to operate with such elements will be to transform the

initial state (representative of the parts of the cases) in the different states that represent the compatibility with each of the elements of the entry that have been read until then.

Therefore, we will have a structure of the aspect $Q_{n+1} = Q_n + E_n$ where E_n is the n symbol of the entry that will be used to change the linear state. In fact, E_i and Q_i have to be elements of bijective sets, and we look for the way to operate with the autovalues of the states that, as we add them up with the input, the final state (the autovalues as a whole) must host all the solutions.

A waveform signal representing an impulse can be used, knowing that an impulse is a signal that returns 1 when the parameter is 0, and 0 in all other cases, a way to simulate the operation may be:

$$\delta_T(t) = \sum_{i=1}^{m} \frac{1}{m} \cos(\frac{2\pi}{T} t) \tag{4}$$

The function δ_T (explained and developed in (Oppenheim and willsky and Nawab, 2016)) is a period impulse of T ($\delta_T(t+T) = \delta_T(t)$), where m, the larger it is, the closer it will be to what is expected of the function. This function is defined in such a way that it is easy to calculate its derivative, integral, signal displacement..., as well as various operations that allow us to give it a linear treatment.

If the signal is δ_T then the autovalues will be the actual numbers that multiply to the cos function. Initially all are m^{-1}, they are the so-called spectral coefficients of the signal. Just as it could have chosen the impulse function, it could have chosen a square signal and determined the spectral coefficients according to the Fourier series, for example, and these studies would have been equally valid.

Let's say that the operation $\delta_T(0)$ will represent the number of cases in the current state (which by default will be 1 case), and the operation "sum of each entry" introduced will accumulate new cases according to a simple algorithm:

$$\begin{cases} Q_0(t) = \delta_T(t) \\ Q_{n+1}(t) = Q_n(t) + Q_n(t - E_n) \end{cases} \tag{5}$$

This algorithm, while implementing the theoretical operation $Q_{n+1} = Q_n + E_n$, also has the peculiarity that it does so in time independent of the size of the entry, or at least in principle: m to give good results will be speculated that it is large enough in relation to precisely the size of the entry. However, it is still advantageous to understand that if you have a machine that provides us with this type of operations, you will always be able to offer us all the precision that we require the greater m.

Now let's determine a problem that can solve this signal: $\#SUM0^*$: Given n integers, return the number of sets that can be formed whose elements add up to 0.

This problem has two details in its notation: on the one hand, the # symbol has been put in its header, which means that the number of cases of a *NP* problem called homonymously will be counted. On the other hand the $*$ symbol has been included, this is because there is already a known problem called equal, the peculiarity that we bring here is that we include as a solution the empty set: so the result will always be a natural greater than 0.

Seen up to this point, we observe that the invariant of the problem is met trivially along the algorithm: in each moment we will have as signal the signal itself plus a replica after

incorporating the sum of the next integer. This operation, without spectral coefficients, would be an operation that would lead to a combinatorial explosion. But since we have a linear signal, it is feasible to do this operation within its margin of error. In the end, $\#SUM0^*[E_0,\cdots,E_{n-1}] = Q_n(0)$, with $T = 2\cdot n\cdot max(E_0,\cdots,E_{n-1})$. In fact, if you progress with an algorithm that incorporates self-regulation mechanisms, and measurements we can have the following loop written in Python:

```
from impulsos import ImpulseU
#E entry: list of integers
I = ImpulseU(2*max(E)*len(E), U = 0, m = 1600)
for X in E:
    I += I.cloneAndMove(x = X)
    I.move(y=-I(0.5))    #S1.
    if round(I(0),0)>0:      #S2.
        I.multiply(round(I(0),0)/I(0))
    else:
        I.move(y=1-I(0))
int(round(I(0),0))
```

In this class, ImpulseU incorporates three parameters instead of two: the first will be the period and the last the number of spectral coefficients. It also incorporates a threshold value that, if greater than 0, will be used to subtract it from the values close to 0. In the loop, in order to protect the invariant, two decisions have been taken: $S1$ and $S2$ that, on occasions could distort the result generating fewer cases of the correct ones ($S1$ inhibition) or more cases of the correct ones ($S2$ excitement).

$S1$ is justified because between integer and integer should ideally return 0 cases, so it automatically subtracts from the whole signal the cases that account for a position that in theory should be null. It is the same effect generated by the threshold parameter except that, at the same time that the signal is subtracted, the U value is automatically divided by $1-U$, so that the impulse result is 1 in the position of 0. That's why, if $S1$ is applied only occasionally, it should also be applied once in a while $S2$.

```
def __init__(self, T, U=0, m=80):
        self.m = m
        self.T = T
        self.sc = {}
        self.ss = {}
        self.sc[0] = -U
        for k in range(1, m):
            self.sc[k] = 1 / (m - 1) / (1 - U)
```

$S2$ is activated as long as the number of cases rounded to the nearest integer read up to then is greater than 0, otherwise it is reset to at least recognize one case by increasing the whole signal. If the rounding of cases is greater than zero, then the signal is reset to return the nearest integer and thus the invariant that the number of cases has to be a natural.

Note that although a stochastic study of the margins of error has not been included, it could always be interesting to determine what values the parameters should have so that

they return the most accurate results possible for the time the user is willing to pay for, easily set by the $K \cdot m \cdot n$ being K the constant that will adjust the units to units of time for the chosen platform.

Other applications.

When we have an expression of the type $X_0 + X_1 - 2 \cdot X_2 - X_3 = 0$ for X_i boolean we actually have the decoding of a two-bit number in $2 \cdot X_2 + X_3$ from the sum of two bits. This means that $X_2 = X_0 \& X_1$ and $X_3 = X_0 \oplus X_1$. Hence, a homogeneous system of equations would be formed by form formulas:

$$F_i = \sum_{k=0}^{V-1} A_{i,j} \cdot X_j \tag{6}$$

With V being the number of Boolean variables that we will work with. Each of these formulas will be composed of the four components to allow amounts between -3 and 3. This encodes up to $3 - (-3) + 1 = 7$ different values, so all equations can be merged to create a single formula:

$$F = \sum_i 7^i \cdot F_i = 0 \tag{7}$$

The end result will be for a $SUM0$ problem. So our caseload approximation system can be applied to all problems that can be expressed within Boolean arithmetic.

Another mechanism for calculating logical satisfaction.

Needless to say, it is possible to find different ways to find the solution of a Boolean equation to satisfy. One of the methods is to use function analysis tools to leave functions unresolved, and then proceed with the calculation when the expression is complete. To see an example we will take the following function:

$$m = \lambda x : \frac{1 - \cos(x \cdot \pi)}{2} \tag{8}$$

We see that it calculates $x \pmod 2$, so therefore this other function:

$$M = \lambda x : \frac{x - m(x)}{2} \tag{9}$$

It calculates the whole part of half a number. These functions can be used to determine the ith digit of a number: $C_i = \lambda n : M^i \odot m \odot n$ which means composing M i times before removing the module with the m function.

In short, given a variable of $Z \in N$, you can define the Boolean variable $X_i = C_i \odot Z$, so that if you have expressions made up of Boolean variables then, after merging them all into one (multiplicating or adding them appropriately), you could result in an expression indefinitely derived from the form: $F(Z) = 0$ which, in addition, knowing the Z problem would be limited between two known values. Doing convergence studies or, in the worst case, after determining what would be the coefficients of Taylor's development of the F

function (to calculate its zeros from the zeros of its development), one could obtain the value of Z that makes F annulled which, in short, is equivalent to satisfying the Boolean variables of an equation.

These mechanisms are not as accurate as one would like them to be, but they can be as efficient as one considers, provided that consideration is given to how for each derivative implicit expression will grow and, at the same time, the precision of convergence will increase. Machine errors that will house the cosine tables and propagation errors must be considered. Even so, these mechanisms will offer approximations within a pre-established time frame.

Cook's trick.

A detail that should not go unnoticed should be highlighted. It turns out that while Cook's theorem does not serve to justify the existence of the NP complete for us anymore, because that definition no longer serves us, it is true that the theorem has a constructive structure that can be used for this formal philosophy. In other words, the MT^F is more practical than it seems if you know how to play the cards well. On the other hand, all problems within the NP complete class can be taken into account to improve the implementation or accuracy of $\#SAT$.

From a theoretical point of view, it is not difficult to imagine how to configure a problem within the MT^F being this a NP problem and, therefore, apply the techniques proposed in Cook's theorem to generate a Boolean formula within the SAT format. In that way, a compiler could use these techniques to provide either estimates, accurate results or whatever else is appropriate. Later on we'll see how inside MT^S we can actually make $TQBF \prec SAT \in P$.

The Total Class

As any attentive person may have noticed, there is a high relationship between the theoretical and the process of invention. What an innovation entails will depend on how complex the theoretical model is, while simplicity of discovery will require simple models. It is therefore necessary to understand how the explanation and reasoning process works in the simplest way. This type of calculation has historically been attributed to logic.

Logic is the exact calculation of knowledge modelling. As long as we have good tools to work logically, we will be able to understand when a model fails or how far it can go.

Having a complete axiomatic assembly that is complete, as Gödel demonstrated, can ensure that any model that conforms within the language of logic as an open problem will always have a demonstration within logic that labels it as true or false.

Because this property is not given in the most general languages (type 0 according to Chomsky's hierarchy, whose grammar is as powerful as a MT) it is necessary to understand how far the problems that arise, not only within the Boolean logic, but also within any area where after raising the problem can be ensured whenever there will be a demonstration. Considering that a problem is a formulation expressed in language intended to be labelled as true or false, problems that are expressed within a language whose parameters cannot

provide the constitution of a problem that in some entries is undecidable should be labelled differently. Maybe, in honor of Church, call them trouble within the Total class.

Thus, if a problem is said to be of the Total class, it means not only that it will be labelled, but there is also another problem of his brother whose resolution could be different. For this we would distinguish the idea of an object class, which is the one that establishes efficiency properties of language, combined with the class that would categorize the management of this efficiency, such as the Total class. For example, it would be interesting to say that a π problem described in L language is part of the TP class, which would be equivalent to saying that a C_L subclass has been found where: $\pi \in C_L \subseteq P \cap T$ because π is expressed in terms of some parameters w_i of L where $\forall w_i : L[w_0, \cdots, w_n] \in P \cap T$. In this object the object class would be P.

There is an enormous range of recognised classes within the complexity study. The critique that underlies it here is the enormous ease of discovering a new plot within that literature and at the same time, how incredibly complicated it is to find some connection with the world of the user to generate an innovation.

That is why without these studies having to be neglected, it is possible to propose many of these classes but exclusively within the Total class. In this way, any new appreciation or discovery within the hierarchy of classes will be more likely to be associated with some possible innovation.

Demonstration $SAT \in P$ for every application.

This section will show a useful tool to let the logical calculation persist. I have become accustomed to calling this process crystallization, in any case that is what the structure is called. We must understand that sometimes it is important to be aware of some intermediate state to help those who reason more slowly or with fewer workspaces, for those tasks this structure is ideal, because by means of the state pattern it is possible to describe the current state through some language recognizable by the user.

However, these assessments would not be possible if the principle of optimisation were not complied with. In other words, the best of explanations must be able to be divided into sub-explanations that are, at the same time, the best explanation in their field. Therefore, the logical calculation that can be explained must comply with this principle. The peculiarity of this principle is that any process that can divide its input into two parts to solve them separately and take advantage of their solutions does not complicate the process beyond the polynomial; that is, if the resolution of particular cases is polynomially delimited then the synthesis process will be within the P class.

That said, we need to define a logical calculation for our purposes. Because if we include in the logical calculation the determination of whether a mathematical theorem is true, then we will see ourselves as having some paradox, based on what is already known today.

We will define the logical calculation in this document within the scope of Boolean arithmetic, for reasons of maximum simplicity and not to make this document too bold.

Boolean arithmetic consists of an algebra formed by a system of sums and products applied on variables that can only take value of 0 or 1. These variables are called boolean variables, the boolean sum is the logical operator or ($\vee$), which is 1 when one of its operands is 1, otherwise 0. The boolean product is the logical operator and ($\wedge$), which is 1

when both operands are 1, otherwise it will return 0. In the same way, the not operator will return the complementary value of the formula it evaluates. A literal is either a Boolean variable or its negated. A well-formed formula is formed by the operation of two well-formed formulas using boolean sums or products. A well-formed formula can also be the negation of another well-formed formula or a Boolean variable. Likewise, this arithmetic is equivalent to symbolic formal logic, where the implication, co-implication and xor connectors can be expressed within Boolean arithmetic in a simple way.

To tackle *SAT* a new operator will be invented, for which the Boolean sum will be expressed by using the $\vee$ connector, while the arithmetic sum $N \times N \mapsto N$ will be expressed by the blades $+$ or the sum with $\sum$.

Definition 2
Alternation is defined as a Boolean function $B^n \mapsto B$ where

$$(A_0|\cdots|A_n) \iff \sum_{i=0}^{n} A_i = 1 \tag{10}$$

We will understand that, although A_i can be well formed formulas, or even literal, for reasons of simplicity in this document will only use such nomenclatures to define the invariant of the structure, for example:

Lemma 3
Set A Boolean, $\forall A \quad (\neg A|A) = True$

Thanks to this trivial lemma of demonstrating, when we write $(A|B)$ we will know that A and B are Boolean variables that complement each other. On the other hand, we can also see the following lemma:

Lemma 4
Setting A,B,Z_1,Z_2 Booleans, $\forall A,B\ \exists Z_1,Z_2$

$$(\neg A|Z_1|A \wedge B) \wedge (\neg B|Z_2|A \wedge B) \wedge (Z_1|Z_2|A = B) = True$$

Proof
Trivial. Just try to give A and B value within its four combinations and check that there is a single Boolean value in Z_1 and Z_2 that makes the expression true. □

In this lemma we observe that if the product of three alternations is expressed in the following way: $(A|B|C)(D|E|C)(B|E|F) = 1$ we will have a mechanism to attribute to a variable the result of making an operation close to the AND and on another variable an operation close to XOR. This is enough so that we can constitute any well-formed formula only with the notation of the alternating product.

Definition 5
A Boolean formula f clause conjunction A^i of alternations es a correspondence between $B^n \mapsto B$ which is true if all A^i clauses are true, where A^i is the ith alternation of Boolean variables.

$$f = \bigwedge_i A^i = (A_0^0|\cdots|A_n^0) \wedge \cdots \wedge (A_0^m|\cdots|A_t^m) \tag{11}$$

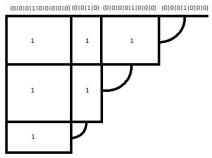

Figure 2. Example of one solution.

This is the notation that will be used in this document to understand the conclusions that will be drawn from the loop shown below. It follows that when writing $A_i^I = 1$, it means that the formula chooses the I^{th} clause and, within that clause, you choose the literal i^{th} and attribute it a 1, which implies that $\forall k \neq i : A_k^I = 0$.

We will generate some matrices like in figure 2, to reach a solution in a product of clauses.

Definition 6
Set f boolean formula of clause conjunction A^i of alternations the assignation $\sigma\binom{I\,J}{i\,j}$ which returns 1 if f is satisfied when 1 is assigned to the i^{th} literal of clause I and the j^{th} literal of clause J in f.

$$\sigma\begin{pmatrix} I & J \\ i & j \end{pmatrix} = f \wedge A_i^I \wedge A_j^J \tag{12}$$

That means that $\sigma\binom{I\,J}{i\,j}$ is represented by a boolean matrix that stores possible solutions of f. From there, it's easy to understand:

Theorem 7
Given a f conjunction of alternations, if$\exists I, J$ with A^I and A^J clauses of f and $\forall i, j : \sigma\binom{I\,J}{i\,j} = 0$ then f is not satisfied (is inconsistent).

It should be noted that Dirac's notation could also have been chosen, but for the purposes of this documentation, we leave it as it is because it is sufficient.

Definition 8
Given a f conjunction of alternations, $\forall K_1, \cdots, K_n$ where A^{K_i} the n clauses of f

$$\sigma\begin{pmatrix} K_1 & \cdots & K_n \\ k_1 & \cdots & k_n \end{pmatrix} \iff \bigwedge_i A_{k_i}^{K_i}$$

Which represents whether that assignment satisfies the formula.

From this it can easily be deduced that:

Lemma 9
Given a f conjunction of alternations, $\forall I, J, K$ where $I > J > K$ with A^I, A^J and A^K clauses of f

$$\forall i, j, k : \quad \sigma\begin{pmatrix} I & J & K \\ i & j & k \end{pmatrix} \iff \sigma\begin{pmatrix} I & J \\ i & j \end{pmatrix} \wedge \sigma\begin{pmatrix} I & K \\ i & k \end{pmatrix}$$

This is crucial, because the process of crystallization of the formula consists of facing each clause two to two because, in this way, it will be enough to deduce their satisfying.

Now, if you have an assignment in the following form: $\sigma\binom{I\,J\,K}{i\,j\,k} = 1$, it will necessarily be deduced that $\sigma\binom{I\,J}{i\,j} \wedge \sigma\binom{I\,K}{i\,k} \wedge \sigma\binom{J\,K}{j\,k} = 1$. If, on the other hand, $\sigma\binom{I\,J\,K}{i\,j\,k} = 0$, that is because one of the three assignments, at least, is 0. Now then:

Lemma 10
Given a f conjunction of alternations, $\forall I,J,K$ where $I > J > K$ with A^I, A^J y A^K clauses of f satisfied by a unique possible assignment, the transitive property as observed is fulfilled:

$$\forall i,j,k: \quad \sigma\binom{I\,J}{i\,j} \wedge \sigma\binom{I\,K}{i\,k} \Longrightarrow \sigma\binom{J\,K}{j\,k}$$

Proof
Given the unique solution $S = \sigma\binom{K_1 \cdots K_n}{k_1 \cdots k_n}$ then $\forall I,J,K,i,j,k$: the following possibilities exist:

a) If the pair (I,i) does not correspond to S then the lemma is satisfied because $0 \to 1$.

b) If the pair (J,j) or the pair (K,k) does not correspond to S then it is satisfied because $0 \to 0$.

c) if corresponds $(I,i) = (K_x,k_x)$, as well as (J,j) and (K,k) then it is met because $1 \to 1$.

d) There are no other possibilities.

□

At this point it is easy to sustain the next corollary:

Lemma 11
Given a f conjunction of alternations, $\forall I,J,K$ where $I > J > K$ with A^I, A^J and A^K clauses of f, when in f there is only an unique case:

$$\forall i,j,k: \quad \sigma\binom{I\,J}{i\,j} + \sigma\binom{I\,K}{i\,k} + \sigma\binom{J\,K}{j\,k} \neq 2$$

Lemma 12
Given a f conjunction of alternations, $\forall I,J,K$; with A^I, A^J y A^K different clauses of f

$$\forall i,j: \quad \sigma\binom{I\,J}{i\,j} = \bigvee_k \sigma\binom{I\,K}{i\,k} \wedge \sigma\binom{J\,K}{j\,k}$$

Proof
Given that $\sigma\binom{I\,J}{i\,j} = f \wedge A^I_i \wedge A^J_j$, what is extracted for any other K clause of f is that if $\sigma\binom{I\,J}{i\,j}$ is part of one of the solutions of f, then there must be a k for which $\sigma\binom{I\,J\,K}{i\,j\,k} = 1$. On the other hand, if $\sigma\binom{I\,J}{i\,j} = 0$, any k would secure the lemma.

□

From the previous lemma, the following corollary follows:

Lemma 13
Given a σ, solutions of f, which is conjunction of alternations, $\forall I,J,i,j$

$$\sigma\binom{I\,J}{i\,j} = \sigma\binom{I\,J}{i\,j} \wedge \bigwedge_K \bigvee_k \sigma\binom{I\,K}{i\,k} \wedge \sigma\binom{J\,K}{j\,k} \tag{13}$$

The above play on words, which seems badly written, is fundamental to understanding how the invariant is formed; and that is that, at all times, it can be understood that to ensure $\sigma\binom{I\,J}{i\,j}$, it is only enough to find some k for any K that meets $\sigma\binom{I\,J\,K}{i\,j\,k}$.

Therefore, from a table that stores σ_2 as the matrix that stores the coherence that exists between two clauses without considering the rest of the formula (as if the formula were only two clauses), move to σ_3 can be done using the previous lemma. In short:

$$\sigma_2\binom{I\,J}{i\,j} = \neg\exists k : A_k^I = A_i^I \wedge i \neq k \vee A_k^J = A_j^J \wedge j \neq k \tag{14}$$

$$\sigma_{m+1}\binom{I\,J}{i\,j} = \sigma_m\binom{I\,J}{i\,j} \wedge \bigwedge_K \bigvee_k \sigma_m\binom{I\,K}{i\,k} \wedge \sigma_m\binom{J\,K}{j\,k} \tag{15}$$

Definition 14
We define the following algorithm from the A^I clauses of a formula:

$$\begin{cases} \sigma_2\binom{I\,J}{i\,j} = \neg\exists k : A_k^I = A_i^I\,|_{i\neq k} \vee A_k^J = A_j^J\,|_{j\neq k} \\ \sigma_{n+1}\binom{I\,J}{i\,j} = \sigma_n\binom{I\,J}{i\,j} \wedge \bigwedge_K \bigvee_k \sigma_n\binom{I\,K}{i\,k} \wedge \sigma_n\binom{J\,K}{j\,k} \end{cases} \tag{16}$$

That procedure already leads us to our most definitive lemma:

Lemma 15
Given a σ_m defined by the algorithm before and given σ solutions of f, conjunction of n alternations, then $\sigma_n = \sigma$

Proof
a) If the f formula is two clauses, the demonstration is trivial.

b) If the formula does not satisfy, then there will be some I,K where at some moment m of iteration: $\bigvee_k \sigma_m\binom{I\,K}{i\,k} = 0$, so it would have moved to $n \geq m$ and, therefore, $\sigma_m = \sigma$.

c) If the formula satisfies, then in each iteration m we look at any: I,J,K to calculate: $R = \sigma_m\binom{I\,J}{i\,j} + \sigma_m\binom{I\,K}{i\,k} + \sigma_m\binom{J\,K}{j\,k}$.

c1) If $R < 2$, then i,j,k that do not satisfy has been chosen; the σ_m source was more open than the σ solution, so ignore that tuple as a solution.

c2) If $R = 3$, then the solution has been found i,j,k; if the solution is unique then it is satisfied transitively with other tuples found in other iterations.

c3) if $R = 2$, then the number of solutions is not 0 or 1, and we have tangled data. That means that if we cancel one of the assignments $\sigma_m\binom{I\,K}{i\,k}$ or $\sigma_m\binom{J\,K}{j\,k}$ to go from 1 to 0, then we will have eliminated a solution to collapse the f function to an observation of a single solution in this iteration. So if the algorithm is regenerated again from the beginning, with this cancellation, then at least at this point it won't be $R = 2$.

(d) No other cases are possible.

□

In addition, it is also very easily deduced that the algorithm quoted runs through quadratic complexity time (in relation to n, number of clauses of f) every iteration from $sigma_m$ to $sigma_{m+1}$; so if you only have to do n iterations, that gives us an algorithm of cubic complexity in time and quadratic in space. Note that the demonstration process involves making extra iterations to justify that the result in cubic time is the required.

It follows that $SAT \in P$.

Developments and probable measurements. Double and triple correlation.

It is easy to generate a structure implemented in Python that solves the previous algorithm. However, care must be taken to use the collapse of observations to extract solutions. That is, to the extent that you have a mechanism for calculating the exact amount of solutions, you will also have a quick indexing algorithm. If there is only one structure that tells us when a formula satisfies two specific clauses, then the mechanism of extracting the solution will have to increase the complexity by one degree in order to shorten formula two to two.

One possible implementation is presented in the following code:

```
import numpy as np
class CrystalPure:
 'alternatives of literals with product of choice operators'
 def __init__(self, *alternatives):
  self.f = alternatives
  self._initializeTables(alternatives)
  self.sharping()

 def _iniTable(self, I, J):
  T = np.ones((len(self.f[I]), len(self.f[J])), int)
  for i in range(len(self.f[I])):
   if self.f[I][i] in self.f[J]:
    j = self.f[J].index(self.f[I][i])
    for k in range(j):
     T[i][k] = 0
     T[(i+k+1) %len(self.f[I])][j] = 0
    for k in range(j+1, len(self.f[J])):
     T[i][k] = 0
     T[(i+k+1)%len(self.f[I])][j] = 0
  return T

 def _initializeTables(self, alternatives):
  self.t = [
    [self._iniTable(i, j) for j in range(i)]
     for i in np.arange(len(alternatives)-1, 0, -1)]

 def m(self):
  for tables in self.t:
   for k in range(
        len(self.f[len(tables)])):
    for table in tables:
```

```
    for X in table[k]:
     print(X, end="")
    print("", end="  ")
   print("")
  print("")

def table(self, I, J):
 if J>=I:
  return self.table(J, I)
 return self.t[len(self.t)-I][J]

def __len__(self):
 return self.table(0,1).sum()

def _sharp(self, I, J, K):
 'I > J > K clauses de f'
 JK = np.zeros_like(self.table(J, K))
 for i in range(len(self.f[I])):
  for j in range(len(self.f[J])):
   for k in range(len(self.f[K])):
    JK[j][k] += min(self.table(I,J)[i][j],
                    self.table(I,K)[i][k],
                    self.table(J,K)[j][k])
 self.t[len(self.f)-1-J][K]=JK

def sharping(self):
 for I in np.arange(len(self.f)-1, 1, -1):
  for J in np.arange(I-1, 0, -1):
   for K in np.arange(J-1, -1, -1):
    self._sharp(I, J, K)

def __getitem__(self, pos):
 pos %= len(self)
 for i in range(len(self.f[1])):
  for j in range(len(self.f[0])):
   if pos >= self.table(1, 0)[i][j]:
    pos -= self.table(1, 0)[i][j]
   else:
    return self._trasiego(self.table(1, 0)[i][j],
                          pos, j, i)

def _trasiego(self, maximo, pos, *asig):
 for k in range(len(self.f[len(asig)])):
  Z = maximo
```

```
    for i in range(len(asig)):
     Z = min(Z, self.table(len(asig), i)[k][asig[i]])
     if Z == 0:
      continue
     elif pos >= Z:
      pos -= Z
     elif len(self.f) == len(asig) + 1:
      return asig+(k,)
     else:
      return self._trasiego(Z, pos, *(asig + (k,)))

def __call__(self, caso):
 R = set([])
 L = self[caso]
 for i in range(len(L)):
  R.add(self.f[i][L[i]])
 return R
```

The loop revolves around the _sharp method; as you can see, instead of calculating the boolean product, it calculates the minimum value, and instead of calculating the or, it adds up the cases. For practical purposes, the calculation of *SAT* is irrelevant: if it is different from 0, any integer is equivalent to recognizing solutions. So the chosen mechanism is more powerful: it calculates other things additionally.

However, what does it calculate? When we have alternating cases and add them together, we know that this coincides with the sum of separate cases; therefore, if the number of cases had been correctly added together, the sum justifies their function. However, the minimum number of cases is only given when the data correlates as much as possible between them, so the result should be higher for large structures.

For this reason, it is possible to implement other types of data structures that allow us to get even closer to the boundary. The issue is to generate a structure that is more generic than an integer, so that when two integers are added together they sometimes add up arithmetically and sometimes calculate the minimum. Because the exact proportion is usually at a mid-point, one can always speak of a probable correlation coefficient of γ_U representing the cardinal's estimate that we want: $\#(A \cup B) \sim \gamma_U \cdot (\#A + \#B) + (1 - \gamma_U)(min(\#A, \#B))$. Sometimes it is not easy to calculate the number of cases at an intersection.

However, you can further fine tune the boundary by scrolling through the previous structure to change the sharp method. By means of a correlation γ_D when the minimum is to be calculated in substitution of the boolean product, in reality what is being done is to define a value between these two values, so it is possible to define:

$$min_{\gamma_D}(A_0, \cdots, A_{n-1}) = \gamma_D \cdot min(A_0, \cdots, A_{n-1}) + (1 - \gamma_D) \cdot \prod_{i=0}^{n-1}(1 - \delta(A_i)) \tag{17}$$

Being $\delta(x)$ the impulse function: 1 when x is 0, and 1 in the rest.

In this way the result remains more generic than SAT and, depending on γ_D, more likely closer to $\#SAT$. It is not difficult to calculate the margin of error, to provide a more accurate result.

However, by studying the method we can still speak of a much more precise correlation coefficient that can help us to estimate more accurately what is required.

To begin with, we can have the lowest measurement estimate much closer than the Boolean product. To do this we will use the following notation:

$$\#\binom{I\ J}{i\ j} = \#(f \wedge A_i^I \wedge A_j^J) \tag{18}$$

This is the number of cases that the well-formed f formula has when i and j are selected to their respective clauses. From there, it can be easily demonstrated:

$$\forall I,J,K : \#\tbinom{I\ J}{i\ j} = \sum_k min\left(\#\tbinom{I\ K}{i\ k}, \#\tbinom{J\ K}{j\ k}\right) - min\left(\#\tbinom{I\ J\ K}{\neg i\ j\ k}, \#\tbinom{I\ J\ K}{i\ \neg j\ k}\right) \tag{19}$$

Where $\#\binom{I\ J\ K}{\neg i\ j\ k}$ represents the sum of the cases for all literals of I other than i, combined with the corresponding allowance of J and K. Now, how do you estimate this value? Well, considering the following inequality:

$$\#\binom{I\ J\ K}{\neg i\ j\ k} \leq min\left(\#\binom{I\ J}{i\ j}, \#\binom{I\ K}{i\ k}, \#\binom{K\ J}{k\ j}\right) \tag{20}$$

In the end, modifying the code will be very easy to achieve as a result an inferior boundary.

Since this last modification we can already speak of a triple correlation coefficient: γ_T, whose use would be to redefine the sharp method, that is, if $\#_m$ is the estimation of the number of cases in the m iteration:

$$min_{\gamma_T}\tbinom{I\ J\ K}{i\ j\ k} = min\left(\#_m\tbinom{I\ K}{i\ k}, \#_m\tbinom{K\ J}{k\ j}\right) - \gamma_T \cdot min\left(\#_m\tbinom{I\ J\ K}{\neg i\ j\ k}, \#_m\tbinom{I\ J\ K}{i\ \neg j\ k}\right) \tag{21}$$

This equation will add up for each k the cases that will define the cases of i and j in their respective clauses.

These clarifications make the code more complex in appearance, however, all the structures that, empirically speaking, coincide because they have the same correlation must be considered. It is also possible to use known results, such as telephone numbers, to estimate this coefficient relative to the number of vertices in a network before calculating the Hosoya index (index proposed in (Hosoya, 2003) to guess molecular recomposition). In any case, these are study tools that can offer innovative results.

Context Mechanisms.

A final contribution to better estimates could be to consider improving the ability to account for each case. Previous techniques used floating numbers to return empirical estimations. Symbolically, it is still possible to give accurate results that are more precise, but in order to do this, the idea of the whole must be generalized, in this case turning it into a vector.

We know that two vectors are different if one of their components is different, but what if we say that one of their components is a marker that can take any value? Then we will be defining instances within a negative database, and we will say that two instances are incompatible if they do not have any instances in common. Understanding this way that $(?,1,2)$ is compatible with $(1,?,2)$ because they have in common $(1,1,2)$. But $(?,1,2)$ is incompatible with $(1,?,1)$ because they have no instances in common.

In this way, we want to count the number of different tuples in the system, so that we do not count the same instance twice. To understand it in a simple way, we can use the following PatternMax class, which builds a pattern code from a conjunction of alternates of up to three literals and sets a parameter that tells you the maximum number of instances to be stored so that it does not explode in combination:

```
class PatternMax:
 def __init__(self, listaAlt, maxSeq):
  self.n = len(listaAlt)
  self.maxSeq = maxSeq
  self.variables = {}
  for i in range(len(listaAlt)):
   for j in range(len(listaAlt[i])):
    if listaAlt[i][j] in self.variables:
     self.variables[listaAlt[i][j]].append((i,j))
    else:
     self.variables[listaAlt[i][j]]=[(i,j)]

 @staticmethod
 def MASK(A, B):
        if A=='3':
            return B
        if B=='3' or B==A:
            return A
        else:
            return ''

 @staticmethod
 def MASKS(sec1, sec2):
       R=''
        for A, B in zip(sec1, sec2):
            X = PatternMax.MASK(A, B)
            if not X:
                return ''
            R += X
        return R

 def __call__(self, *variables):
  sequence = "3"*self.n
```

```
 for V in variables:
  for i, j in self.variables[V]:
   R = PatternMax.MASK(sequence[i], str(j))
   if R == '':
    return CodeMax.NULL(self)
   sequence = sequence[:i] + R + sequence[i+1:]
 return CodeMax(self, sequence)
```

This code uses the marker number of 3 as a wildcard character, but it is not difficult to change the code to accept any number of verbs for each clause.

It is based in the idea that if we recognize the variables in the clauses, the position they ocupy is the component of the instance; so if it is defined the case will be different to each other where in the same component is assigned other value. For example, formula $f = (X_1|X_2|X_3)(X_3|X_4|X_5)$ recognize for the variable X_1 the code 03 that means it appears in first clause in position 0 and not in the second one. And the vector for $X3$ will be 20, so if we accept X_1 we see there is no case in common with accepting X_3, so the union of both cases will be a simple sum.

In any case, the masking methods we see are used to check which instances have in common two codes that use this same pattern. And when a call is made to a *PatternMax* object, a *CodeMax* object is generated that has the following structure:

```
class CodeMax:
 def __init__(self, pattern, sequence, cardinal = 1):
  self.sequences = {sequence: cardinal}
  if sequence:
   self.sequences[''] = cardinal
  self.pattern = pattern
  self.diminution = 1

 def _correctionMaximasequences(self):
  while len(self.sequences) > self.pattern.maxSeq+1:
   self.sequences = self._fuse(self.diminution)
   self.diminution += 1

 def __repr__(self):
  return repr(self.sequences)

 def clone(self):
  R = CodeMax.NULL(self.pattern)
  R.sequences = self.sequences.copy()
  return R

 @staticmethod
 def NULL(pattern):
  return CodeMax(pattern,'', 0)
```

```
def isNULL(self):
 return len(self)==0

def _fuse(self, divisor):
 R = {}
 for X in self.sequences.keys():
  if X=='':
   continue
  Z = X[:-divisor]+"3"*divisor
  if Z in R.keys():
   R[Z] += self.sequences[X]
  else:
   R[Z] = self.sequences[X]
 R[''] = self.sequences['']
 return R

def __floordiv__(self, divisor):
 if divisor < self.diminution:
  return self
 result = CodeMax.NULL(self.pattern)
 result.sequences = self._fuse(divisor)
 result.diminution = divisor + 1
 return result

def __mul__(self, other):
 R = CodeMax.NULL(self.pattern)
 R.sequences = {}
 acum = 0
 for X in self.sequences.keys():
  for Y in other.sequences.keys():
   S = PatternMax.MASKS(X, Y)
   if S:
    Z = self.sequences[X] * other.sequences[Y]
    acum += Z
    if S in R.sequences.keys():
     R.sequences[S] += Z
    else:
     R.sequences[S] = Z
 R.sequences[''] = acum
 R._correctionMaximasequences()
 return R

def __len__(self):
 return self.sequences['']
```

```
def _len(self):
 S = 0
 for X in self.sequences.values():
  S += X
 return S

def __and__(self, other):
 R = {}
 acum = 0
 for X in self.sequences.keys():
  if X == '':
   continue
  for Y in other.sequences.keys():
   S = PatternMax.MASKS(X, Y)
   if S:
    Z = min(self.sequences[X], other.sequences[Y])
    acum += Z
    if S in R.keys():
     R[S] += Z
    else:
     R[S] = Z
 result = CodeMax.NULL(self.pattern)
 if R:
  result.sequences = R
  result.sequences[''] = acum
  result._correctionMaximasequences()
return result

def __add__(self, other):
 if self.isNULL():
  return other.clone()
 if other.isNULL():
  return self.clone()
 R = self.sequences.copy()
 for X in other.sequences.keys():
  if X in R.keys():
   R[X] += other.sequences[X]
  else:
   R[X] = other.sequences[X]
 result = CodeMax.NULL(self.pattern)
 result.sequences = R
 result._correctionMaximasequences()
 return result
```

Instead of posting the number of cases directly, you can use encodings that recall the compatibility between two clauses. In this way our crystal could be defined as follows:

```
def sharp(self, A, B, C, divisor):
 'A < B < C clauses de f'
  T = self.table(A, B)
   for i in range(3):
    for j in range(3):
     if not T[i][j].isNULL():
      T[i][j] = Cristal.escalarU(
        Cristal.column(self.table(A,C), j),
        Cristal.column(self.table(B,C), i),
                    T[i][j]) // divisor
```

In this case, the method opperates over the matrix itself dividing the result to forget information that is not needed at that point. If only one cifher is forgotten for each step, the result will be exact; but if number of cases is too big perhaps a big number of instances have to be merged together.

The escalarU is a function which will multiply two vectors(with methods __and__ and __add__ of *CodeMax*) with this code:

```
@staticmethod
def escalarU(vect1, vect2, umbral):
 S = CodeMax.NULL(umbral.pattern)
 for X, Y in zip(vect1, vect2):
  S += X & Y & umbral
 return S
```

The code is prepared to accept a pattern to mask under that. We can think in umbral like the net-mask in IP protocol. If that mask fuses all the code in all $3\cdots3$ (that means $?\cdots?$), then only the integer which counts the tuples would remain as information, and this structure would not provide any restriction.

If someone want to test this code directly, do not forget to adapt every formula of clauses of n literals in clauses of 3 variables:

```
def parting(clause, aux):
 if len(clause)<4:
  return [clause], aux
 elif len(clause)==4:
  return [clause[:2]+(aux,), (aux, -aux),
          clause[2:]+(-aux,)], aux+1
 else:
  L = [clause[:2]+(aux,)]
  for X in clause[2:-2]:
   L.append((aux, -aux))
   L.append((-aux, X, aux+1))
   aux += 1
```

```
  L.append((-aux,)+ clause[-2:])
  return L, aux+1

def formatIn3(L, aux = 100):
 R = []
 for clause in L:
  R1, aux = parting(clause, aux)
  R.extend(R1)
 return R
```

Demonstration $TQBF = SAT$**.**

Once studied how any Boolean formula behaves, it is interesting to enlarge our calculation field on logic. Although it may not have gone into detail, any well-formed formula can be evaluated, but what if each and every one of the Boolean variables is quantified with a $\forall$ or with a $\exists$? These types of formulas are called QBF, and the problem of satisfying them is called $TQBF$. It would be interesting to know what connection this type of formula could have with SAT.

Assuming that all the variables are quantified with a $\forall$, we will say that the theorem is considered to be a complete tautology (defined in that way in (Honderich, 2005)). The equivalence of the formula to a formula to satisfy is as simple as considering its negation:

$$\begin{aligned} \forall x_1,\ldots,x_n:\ f(x_1,\ldots,x_n) &\iff \neg\exists x_1,\ldots,x_n:\ \neg f(x_1,\ldots,x_n) \\ \forall x_1,\ldots,x_n:\ f(x_1,\ldots,x_n) &\iff \neg SAT[\neg f(x_1,\ldots,x_n)] \end{aligned}$$

In other words, to demonstrate a tautology, one only has to deny the formula and study if it is satisfied. So, trivially, the problem of guessing if a formula is a tautology equals SAT.

Now, it is interesting to know what happens when existential quantifiers are incorporated $\exists$; it is interesting to know if it can be made equivalent to studying tautologies.

Given a formula within $TQBF$ the first thing that must be done is to transform it into the normal conjunctive form; to do this, it is necessary to consider displacing all quantifiers to the left of the formula (normal prenex form). However, if there are complications in expressing independence $I(K_i,h_j)$ explicit between a parameter K_i and a variable h_j, note that this method can work for any kind of combination without exception, so quantifiers can be ignored, to point out their independence as if they were part of the formula.

Given a well-formed formula in prenex form, the goal is to transform it into

$$\bigwedge_i C_i = \bigwedge_i \bigvee_j L_{a_{i,j}}$$

In this way, we will transform each C_i clause into conjunctors that imply boolean sums using the following rules:

$$B_1 \vee \cdots \vee B_n \iff 1 \to B_1 \vee \cdots \vee B_n$$

$$\neg A_1 \vee \cdots \vee \neg A_m \iff A_1 \wedge \cdots \wedge A_m \to 0$$

And, therefore, in general:

$$\bigvee_i \neg A_i \vee \bigvee_j B_j \iff \bigwedge_i A_i \rightarrow \bigvee_j B_j$$

On the other hand, given a temporal variable τ_k used to replace an implication of conjunctions to disjunctions $\bigwedge_i A_i \rightarrow \bigvee_j B_j$, we apply the following rule to incorporate τ_k into the implication

$$\frac{\tau_k = \left(\bigwedge_i A_i \rightarrow \bigvee_j B_j\right)}{\begin{array}{l}\tau_k \wedge \bigwedge_i A_i \rightarrow \bigvee_j B_j \\ \bigwedge_i : (A_i \rightarrow \tau_k) \\ \bigwedge_j : (1 \rightarrow B_j \vee \tau_k)\end{array}}$$

If we have at least two equivalent expressions for a parameter, that is equivalent to assigning the disjunction of expressions:

$$(A = x) \wedge (A = y) \longrightarrow (A = x \vee y)$$

Proof

$(A$	$=$	$x)$	$\wedge($	A	$=$	$y)$	$\rightarrow ($	A	$=$	x	$\vee$	$y)$
0	1	0	1	0	1	0	1	0	1	0	0	0
0	0	1	0	0	1	0	1	0	0	1	1	0
0	1	0	0	0	0	1	1	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	1	1	1
1	0	0	0	1	0	0	1	1	0	0	0	0
1	1	1	0	1	0	0	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1

□

This lemma tells us that if we have an expression of form:

$$f \wedge \bigwedge_i (A = x_i) = 1$$

Then, thanks to the Boolean addition being associative, we can also assert

$$f \wedge (A = \bigvee_i x_i) = 1$$

We can also observe the following basic principle:

$$\exists A \forall x : \alpha \wedge I(A,x) \longrightarrow \forall x, \exists A : \alpha$$

Because that necessarily means that if the implicant is theorem, the implied one is also:

$$(\exists A \forall x : \alpha \wedge I(A,x) = 1) \longrightarrow (\forall x, \exists A : \alpha = 1)$$

Now, let's study this result: we distinguish clause by clause the variables affirmed ($\forall y_i$) and denied ($\forall x_i$), within each clause, with a sub-index to indicate that we can speak of an

indefinite amount. While to the parameters affirmed will be $\exists B_i$ and the denied $\exists A_i$. From this follows the following theorem:

$$\begin{array}{l}\forall x_i, \forall y_i, \exists A_i, \exists B_i : \\ \bigwedge_i A_i \wedge \bigwedge_i x_i \rightarrow \bigvee_i B_i \vee \bigvee_i y_i \iff \\ \bigvee_i (A_i = 0) \vee \bigvee_{i,j} (A_i = \neg x_j) \vee \bigvee_{i,j} (A_i = y_j) \vee \\ \vee \bigvee_i (B_i = 1) \vee \bigvee_{i,j} (B_i = x_j) \vee \bigvee_{i,j} (B_i = \neg y_j) \\ \forall j : I(A_i, x_j) \rightarrow A_i = x_j \\ \forall j : I(A_i, y_j) \rightarrow A_i = \neg y_j \\ \forall j : I(B_i, x_j) \rightarrow B_i = \neg x_j \\ \forall j : I(B_i, y_j) \rightarrow B_i = y_j\end{array}$$

This formula can be combined with the above equation to determine which clauses make up the final well-formed formula. This way we'll say:

$$\exists Z \forall x : f(x, Z) \iff \forall x \exists Z : f(x, Z) \wedge I(Z, x) \tag{22}$$

That is to say, if by putting all the variables ahead of the parameters we lose generality, we would only have to include propositions with the format $I(Z,x)$ as already mentioned before. Where Z is the existential parameter and x is the variable that is quantified for all Boolean values; to mean $I(Z,x)$ that the value of Z cannot depend on x for each instance or that, if there is an expression that gives value to Z, it cannot include the variable x or any parameter K that does not expressly comply $I(K,x)$.

However, the process of converting any well-formed formula will lead us to create temporal variables, which sometimes appear in the implicant (τ_i) and other times in the implicated one (τ_i^*). So, considering the existence of temporal variables τ_i, it could be interesting to get the most general expression:

$$\begin{array}{l}\forall x_i, \forall y_i, \exists \tau_i \exists \tau_i^* \exists A_i, \exists B_i : \\ \bigwedge_i A_i \wedge \bigwedge_i x_i \bigwedge_i \tau_i \rightarrow \bigvee_i B_i \vee \bigvee_i y_i \vee \bigvee_i \tau_i^* \iff \\ \bigvee_i (A_i = 0) \vee \bigvee_{i,j} (A_i = \neg x_j) \vee \bigvee_{i,j} (A_i = y_j) \vee \bigvee_{i,j} (A_i = \neg \tau_j) \vee \bigvee_{i,j} (A_i = \tau_j^*) \vee \\ \vee \bigvee_i (B_i = 1) \vee \bigvee_{i,j} (B_i = x_j) \vee \bigvee_{i,j} (B_i = \neg y_j) \vee \bigvee_{i,j} (B_i = \tau_j) \vee \bigvee_{i,j} (B_i = \neg \tau_j^*) \vee \\ \bigvee_i (\tau_i = 0) \vee \bigvee_{i,j} (\tau_i = \neg x_j) \vee \bigvee_{i,j} (\tau_i = y_j) \vee \bigvee_i (\tau_i^* = 1) \vee \bigvee_{i,j} (\tau_i^* = x_j) \vee \bigvee_{i,j} (\tau_i^* = \neg y_j) \\ \forall j : I(A_i, x_j) \rightarrow A_i = x_j \\ \forall j : I(A_i, y_j) \rightarrow A_i = \neg y_j \\ \forall j : I(B_i, x_j) \rightarrow B_i = \neg x_j \\ \forall j : I(B_i, y_j) \rightarrow B_i = y_j\end{array}$$

That points to the next rule:

Lemma 16
Given x_i and y_i boolean variables with some A_i and B_i parameters defined by boolean a expression, if the next assignments are satisfied, then implication is a tautology.

$$\begin{array}{l}\forall x_i, \forall y_i, \exists A_i, \exists B_i : \\ \bigvee_i (A_i = 0) \vee \bigvee_{i,j} (A_i = \neg x_j) \vee \bigvee_{i,j} (A_i = y_j) \vee \\ \vee \bigvee_i (B_i = 1) \vee \bigvee_{i,j} (B_i = x_j) \vee \bigvee_{i,j} (B_i = \neg y_j) \Longrightarrow \\ \bigwedge_i A_i \wedge \bigwedge_i x_i \rightarrow \bigvee_i B_i \vee \bigvee_i y_i\end{array} \tag{23}$$

Proof

To prove it we must play with all the combinations: first of all we can imagine how the expression would end up being if the A_i and the B_i were independent of the x_i and the y_i. To begin with, the expressions of the form $\bigvee_{i,j}(A_i = x_j)$, $\bigvee_{i,j}(A_i = y_j)$, $\bigvee_{i,j}(B_i = x_j)$ and $\bigvee_{i,j}(B_i = y_j)$ disappear from the statement. So the statement to prove would be:

$$\forall x_i, \forall y_i, \exists A_i, \exists B_i \wedge \bigwedge_{i,j} I(A_i, x_j) \wedge \bigwedge_{i,j} I(A_i, y_j) \wedge \wedge \bigwedge_{i,j} I(B_i, x_j) \wedge \bigwedge_{i,j} I(B_i, y_j) :$$
$$\bigvee_i (A_i = 0) \vee \bigvee_i (B_i = 1) \Longrightarrow \bigwedge_i A_i \wedge \bigwedge_i x_i \rightarrow \bigvee_i B_i \vee \vee \bigvee_i y_i$$

$$\begin{cases} \exists i : A_i = 0 \rightarrow True \\ \forall i : A_i = 1 \wedge \exists i : B_i = 1 \rightarrow True \\ \forall i : A_i = 1 \wedge \forall i : B_i = 0 \rightarrow True \end{cases}$$

In this way, we test how it would be with the A_i not independent of the x_j or the y_j. Therefore, the following cases should be tested:

$$\forall x_i, \forall y_i, \exists A_i, \exists B_i \wedge \bigwedge_{i,j} I(B_i, x_j) \wedge \bigwedge_{i,j} I(B_i, y_j) :$$
$$\bigvee_i (A_i = 0) \vee \bigvee_{i,j} (A_i = \neg x_j) \vee \bigvee_{i,j} (A_i = y_j) \vee \bigvee_i (B_i = 1) \Longrightarrow$$
$$\bigwedge_i A_i \wedge \bigwedge_i x_i \rightarrow \bigvee_i B_i \vee \bigvee_i y_i$$

$$\begin{cases} \exists i : A_i = 0 \rightarrow True \\ \forall i : A_i = 1 \wedge \exists i : B_i = 1 \rightarrow True \\ \forall i : A_i = 1 \wedge \forall i : B_i = 0 \wedge \exists j : x_j = 0 \rightarrow True \\ \forall i : A_i = 1 \wedge \forall i : B_i = 0 \wedge \forall j : x_j = 1 \wedge \exists j : y_j = 0 \rightarrow True \\ \forall i : A_i = 1 \wedge \forall i : B_i = 0 \wedge \forall j : x_j = 1 \wedge \forall j : y_j = 1 \rightarrow True \end{cases}$$

It can be easily checked for each of the combinations in a similar way as the implication is fulfilled.

□

This lemma is of special relevance because if it is proved that the implication is always true, then it will be assured that the implicated is theorem.

This new expression will catch every possible theorem. In this way, the generality of any quantified formula is obtained in order to find a well-formed formula to satisfy. More specifically:

If we have the formula in format: K as number of parameters, and V number of variables, but was needed T temporal variables to normalize the formula, it is possible to encode a variable Z representing the pair: $K_i = V_j$ where we will include the nular operation as a independent variable $T() = 1$ in the index V_0, which would represent the fact that the parameter acquires a true value independently of any variable. Therefore,

$$(K_i = V_j) \iff Z_{i\cdot(T+V+1)+j} \tag{24}$$

$$(K_i = \tau_j) \iff Z_{i\cdot(T+V+1)+j+V} \tag{25}$$

$$(\tau_i = V_j) \iff Z_{(i+K)\cdot(T+V+1)+j} \tag{26}$$

It is inevitable to argue that the opposite would be coded as follows: $(K_i = \neg V_j) \iff (K_i \neq V_j) \iff \neg Z_{i\cdot(T+V+1)+j}$ i. e. the allocation of the negative is equivalent to denying the assignment itself. Thanks to this coding it is easy to replace any theorem quantified in a

formula within SAT, and then turn it into a product of alternations that will ultimately return an evaluation. The fact that all variables can be satisfied with boolean values Z_i will mean that a coherent solution has been found for assigning each of the parameters to true values, considering the imposed restrictions of parameter independence towards variables (as we have already seen that these restrictions are also parameter-literal assignments). Therefore, satisfaction implies the existence of a tautology in the formula and, consequently, that the theorem is valid; to instantiate solutions means to give value to the parameters to conform tautologies.

These results should be considered important because there are formal demonstrations that link *TQBF* with *PSPACE* problems (in (Arora and Barak, 2016), page 80, are detailed).

Example.

Let's check manually if a certain non-tautological expression is a theorem.

$$\forall P, \forall Q, \exists Z : (P \to Q \vee Z) \wedge P \to Q$$

$$\begin{gathered}\forall P, \forall Q, \exists \tau_1, \exists Z : \\ \tau_1 = (P \to Q \vee Z) \\ \tau_1 \wedge P \to Q\end{gathered}$$

$$\begin{gathered}\forall P, \forall Q, \exists \tau_1, \exists Z : \\ \tau_1 \wedge P \to Q \vee Z \\ 1 \to \tau_1 \vee P \\ Q \to \tau_1 \\ Z \to \tau_1 \\ \tau_1 \wedge P \to Q\end{gathered}$$

$$\begin{gathered}\forall P, \forall Q, \exists \tau_1, \exists Z : \\ (\tau_1 = 0) \vee (\tau_1 = \neg P) \vee (\tau_1 = Q) \vee (Z = 1) \vee (Z = P) \vee (Z = \neg Q) \\ (\tau_1 = 1) \vee (\tau_1 = \neg P) \\ (\tau_1 = 1) \vee (\tau_1 = Q) \\ (\tau_1 = 1) \vee (Z = 1) \vee (Z = \tau_1)\end{gathered}$$

$$\begin{gathered}\forall P, \forall Q, \exists \tau_1, \exists Z : \\ (Z = \tau_1) \wedge (\tau_1 = Q) \wedge (\tau_1 = \neg P)\end{gathered}$$

$$\forall P, \forall Q, \exists Z : Z = (Q \vee \neg P)$$

So if we satisfy $Z = P \to Q$ for every P and Q booleans, then the original expression $(P \to Q \vee Z) \wedge P \to Q$ will be 1.

Conclusions

As this documentation has shown, we have more than sufficient evidence that TQBF $\subseteq$ P; and structures have been shown to be functioning. This type of results can be an enormous advantage when presenting languages that compile data collected by scientists in order to give it a much more intelligent treatment than today. For the time being, according to this documentation, it would already be possible to solve the secondary sequencing of RNA (in (Pierce and Winfree, 2002) it was agreed to limit the field of study, as usual with other problems (Feder and Motwani, 2002) solved here in P in general), so could one expect some kind of formal language that would bring the algebra of atomic links closer to something that is computable within TQBF?

Conflict of interests.

This author declares there is no conflict of interest in this document. The code and document was devoloped in Jupyter, tested and uploaded in:
`https:/archive.org/details/TQBFInP_201802`.

References

Turing, A. (1937) *"On Computable Numbers, with an Application to the Entscheidungsproblem"*, Proceedings of the London Mathematical Society, vol. 2-42, no. 1, pp. 230-265, 1937.

Garey, M and Johnson, D (2009) *Computers and intractability*. New York [u.a]: Freeman, 2009.

Hummel, J. (2010) *"Symbolic Versus Associative Learning"*, Cognitive Science, vol. 34, no. 6, pp. 958-965, 2010.

Pierce, N. and Winfree, E. (2002) *"Protein Design is NP-hard"*, Protein Engineering, Design and Selection, vol. 15, no. 10, pp. 779-782, 2002.

Hosoya, H. (2003) *"The Topological Index Z Before and After 1971"*, ChemInform, vol. 34, no. 15, 2003.

Oppenheim, A. Willsky, A. and Nawab, S. (2016) *Signals & systems*. Noida: Pearson, 2016.

Matijasevich, Y. (1992) *"My collaboration with Julia Robinson"*, The Mathematical Intelligencer, vol. 14, no. 4, pp. 38-45, 1992.

Cook, S. (2003) *"The importance of the P versus NP question"*, Journal of the ACM, vol. 50, no. 1, pp. 27-29

Shor, P. (1999) *"Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer"*, SIAM Review, vol. 41, no. 2, pp. 303-332.

Feder, T. and Motwani, R. (2002) *"Worst-case time bounds for coloring and satisfiability problems"*, Journal of Algorithms, vol. 45, no. 2, pp. 192-201.

Rich, E. (2008) *Automata, computability and complexity*. Upper Saddle River, N.J.: Pearson Prentice Hall, section 28.10 "The problem classes FP and FNP", pp. 689-694.

Honderich, T. (2005) *The Oxford companion to philosophy*. Oxford [u.a.]: Oxford Univ. Press.

Arora, S. and Barak, B. (2016) *Computational complexity*. New York: Cambridge University Press.