American Open Computer Science Journal
Vol. 2, No. 1, July 2015, pp. 1-11
Available online at http://rekpub.com/Journals.php

Research article

Relationship between P and NP. The ultimate definition.

Juan Manuel Dato Ruiz

Manager of Telaingenias, Ingeniería y Servicios Cartagena, Spain
E-mail" jumadaru@gmail.com

Abstract.

In this document I show almost two demonstrations for some scholars contradictorial. So that, I present an explanation to understand why both demonstrations are true because changing the philosophy of the machine. Is every problem in the NP class..., in the P class too? If we have two responses, what of them is more correct? Would we find a way to construct machines which could work in the other way? At last reality is only one, so that we cannot allow two results from the same reality and in this document, at last, we will understand how it is possible.

Keywords: NP, NP-co, P, #P,boolean logic.

1 Introduction

At the beginning of the XXIst Century, we cannot ensure how the Science works yet. Some scientists work in an empirical way thinking all the truth stay under they can see. Some other points to a more idealistic philosophy ensuring they are responsible of the results. Those philosophies are contradictorial, but no one consider their asserts false for that reason. That is so, the philosophy cannot interfere the results, it only changes the interpretation of the results for helping the research of new technologies.
From the ancient Greeks, they had been giving to the role of human intelligence and reason an important role: it should be able to solve the mysteries which were presenting mythological. The reason role appeared to emphasize the use of the demonstrations. So if you could not explain why you defend a concept, then your considerations could not be reasonable. In other words, if you could not set up a configuration in your language to construct a way to your ideas from the ideas of the rest, your discussion would be worthless.

That consideration could be worthy if we remembered there were two philosophies in mathematics: the constructive and the formalist. Indeed, scientific development would be doomed to have to be defined within an environment or idea where the truthful itself was clear, that is, before you know what we mean by scientifically valid you should know what constitutes convincing. And if we have two philosophies for the mathematical demonstrations then, what can we find if we merge documentations of different types?

So the first, and above all, was the philosophy of science: the way we choose to give to fundamental postulates a role in science for our intellectual environment convincing. Something rigorous that we thought it was of rigorous writing, it would be something more like ambiguous. And the role of the philosophy is to fill the gaps before answering the questions.

That is the why we will have two definitions for the Machine of Turing and his NP problems: one definition for a constructive demonstration, and another one for a formalist one. Both definitions will accept a Turing Machine is merely a tape that is modified according to an instruction set. When our instruction manual requires us to give an answer within a time expressible polynomial bound on the size of the input, we say that the machine is polynomially bounded. This is very important because it means that we could budget their construction so that one could study the feasibility of its use.

However, these machines can present a distortion within individual instructions describing the operation of the machine itself: if you are able to describe how to get to the final state regardless of the intermediate states through which must pass the machine, then the machine is called to be not deterministic. That is, with a nondeterministic machine we must say *what we want* and the machine is concerned with *how to get* to that state.

When we combine the two concepts (dimension determinism and polynomial) the key statement will be displayed: polynomially bounded deterministic machines are often called P, nondeterministic machines polynomially bounded are often called NP. The point is, if we can express NP like P, then we could build machines that support more descriptive instructions without losing its deterministic character. So, P is equal to NP? That question is regarded by a million of dollars by Claymath Institute. So the question is still open and in this document we will answer it. And the main problem we really have is that many of the demonstrations surrounding studies of P and NP are stained with a philosophy unfit for scientific development concepts: mixed formalism with constructivism, and Turing Machine does not extend beyond constructivism. However, we will see in this document a philosophy to get a formalist Turing Machine.

The incompleteness theorem of Gödel is essential to understand all these problems: when we combine concepts endless as **ω-unification** with an ability to demonstrate inductively we will get inconsistent results. The **ω-unification** all it means is that we replace the parameters in the corresponding variable to infinity. And of course, there's something left over, can we really carry out an infinite number of assignments? Can we combine our results with arithmetic to prove any statement? Many statements of arithmetic can be proven in a Turing machine in a polynomial bound, so how long can we extent a nondeterministic machine definition for solving what we know we can not solve?

The mathematical formalism is only concerned to prove the statements are coherent. But that kind of universe could allow us possible combinations which are impossible to construct. In any case, my intention here was only to show the difference between symbols constructivists (those from the symbol 1) and symbols formalists (those from logic). Understanding that the formalism will worry about consistency, while constructivism will worry about if it can be demonstrated or built. The formalism is more general and goes further with more claims, while constructivism is simply accurate.

It is, therefore, at this point we realize that the same statement for natural outcome and for the irrational could give the opposite result. But all our considerations will go to the wonderful demonstrations able to do this when the definition of natural was not necessary in the show.

One example is Fermat's theorem. Through a known theorem in reals we can show that the equation $\mathbf{x}^n = \mathbf{z}^n + \mathbf{y}^n$ with **n**, **x**, **y** and **z** are greater than 2 will find a solution for any **n**, **y**, **z** natural in a **x** real. And of Fermat's last theorem, we understand that this equality with **x** natural is impossible. There is a fake counterexample Fermat's theorem that is very famous: Those four naturals work the equation on any calculator, but that can not be corroborated in a computer. That is, from the point of view of natural and constructivism the mathematical statement is exactly as enunciated by Fermat. However, in terms of approaches you can take the opposite case in little machines. We can formally give a practical reality that works coherently with those irrational numbers that makes impossible the statement. And that practical reality could be a short of mind calculator.

That were the reasons we could work with the two philosophies which will answer the same question with the both answers possible.

2 Constructively, P class is different to NP class.

For our first analysis we will study what is the answer by a constructive philosophy. To get demonstration, firstly we have to define our Turing machine as if it were a function, to get it more comfortable.

2.1 Working with lambda calculus.

Given a tape that can encode through a natural number called **E**, we can adopt a configuration set by another natural number which we call **C**, the result is the encoded tape **S**, where **S** = **C** (**E**).

For example, if **C** is the function that adds one to the entry, we can say:

$$\begin{gathered}\mathbf{C} = \lambda\mathbf{N}.\ \mathbf{N}+1 \\ \mathbf{E} = 1 \\ \mathbf{S} = 2\end{gathered} \tag{1}$$

If it is of our interest, we could consider the configuration **C** as another number, so we can work in defining its code. For example, in our language we could recognize three types of tokens used in expressions described in **Table 1**. So the expression *λ**N**. **N**+1* can be expressed in this way: ***$0 + 1***.

Table 1. Tokens in the language which codes a function so general so a Turing Machine.

Type	*Description*	Cardinal
0	symbols	**+**, **-**, *****, **/**, **(**, **)**, ...
1	arguments	**$0**, **$1**, **$2**, ...
2	naturals	**0**, **1**, **2**, ...

Otherwise, if we take pairs with the format ***(cardinal*, *type)*** then

$$\$0 + 1 \equiv (\mathbf{0}, \mathbf{1}), (\mathbf{0}, \mathbf{0}), (\mathbf{1}, \mathbf{2}) \tag{2}$$

At this point we notice that (**A**, **B**) Ξ 3·**A**+**B** = **N** does not loose information, because

$$(A,B) = (N\ DIV\ 3,\ N\ MOD\ 3) \qquad 3$$

So the expression results: **$0 + 1** Ξ **1**, **0**, **5**. At last we only have to use the numbers of Gödel to find an integer, so if we want to encode the function λ**N**. **N**+1, we should make some opperations:

$$\lambda\mathbf{N}.\ \mathbf{N}{+}1 \ \Xi\ cod(\lambda\mathbf{N}.\ \mathbf{N}{+}1) = 2^{\mathbf{1}}\cdot 3^{\mathbf{0}}\cdot 5^{\mathbf{5}} = \mathbf{6250} \quad (4)$$

If we factorized **6250** we could decode the operations λ**N**. **N**+1. Now, let's define a function **XOR** natural: it could be defined as the result of applying the function itself bitwise **XOR** between such natural, so if there is **A XOR B** = **C** then **A** = **B XOR C**, for any **A**, **B**, **C** natural.

Let's choose a value for **A**, in instance, **A** = **342**; otherwise, **B** = **6250**. To opperate **342 XOR 6250** we have to transform both numbers to binary:

$$\begin{gathered}\mathbf{6250_{10} = 1100001101010_2} \\ \mathbf{342_{10} = 101010110_2}\end{gathered} \quad (5)$$

So that, we only have to operate as in **Table 2**.

$$\mathbf{\mathit{1100100111100_2} = \mathit{6460_{10}}} \quad (6)$$

With the **XOR** function definition we will declare **X** so that **X** = **C XOR E**. And here is where we can ask the big question: given the values **X** and **S** (where ***S*** = ***C (E)***), can we deduce **E**?

Moreover, we can simplify our problem: suppose that **C** is bounded polynomially (as we know as **XOR** is), is it possible to deduce **E** within a polynomial bound validating **S** and **X**? If we had all Turing machines at our disposal and some ability to manage at once, the answer is yes: but because we would be using a non-deterministic Turing machine.

Table 2. How works the XOR function on two naturals.

$\mathbf{6250_{10}}$	$\mathbf{342_{10}}$	$\mathbf{6250_{10}}$ **XOR** $\mathbf{342_{10}}$
1	0	1
1	0	1
0	0	0
0	0	0
0	1	1
0	0	0
1	1	0
1	0	1
0	1	1
1	0	1
0	1	1
1	1	0
0	0	0

From our example: if we remember **C** = **6250**, so our entry **E** = **342**. That means that

$$\boldsymbol{S} = \boldsymbol{C}\ (\boldsymbol{E}) = (\lambda\mathbf{N}.\ \mathbf{N}{+}1)(342) = 342{+}1 = \mathbf{343} \quad (7)$$

So if I tell you **X** = **C XOR E** = **6460**, and **S = C (E) = 343, how do you pretend to discover any E?**

The answer that question ensures we are asked to find a **C** that meets: **S** = **C** (**C XOR X**). Then **C** will certificate the return of **E** with the formula **E** = **C XOR X**. Now, it is intended that a Turing machine configuration is able to handle all these possibilities, discriminating configurations that cause the machine to crash, or even those that are not polynomially bounded? If we had such a good mechanism we could use it to index only the machines that works and are polynomial. However, in the proof of Gödel there is no reference to the detail of the dimension polynomial, so finding the equivalence means accepting that we would have a mechanism for dealing with constructible configurations, where the incompleteness theorem ensures otherwise.

2.2 A definition of a function impossible to code.

The explanation in 2.1 may not satisfy some other scholar, however you can expect an alternative proof even more rigorous and elaborated that demonstrates booster for those who still harbor doubts: it is the using of a function called **H**.

We say that **H** is an integer representing the configuration of a Turing machine, where if its entry is **B** then its output will match a single **X** that comply necessarily:

Rule 1. if **X** (**a1**) = **b** and **X** (**a2**) = **b** then **a1** = **a2**. That is, X is injective.
Rule 2. . **X** (**X**) = **A**
Rule 3. . **X** (**X** + **A**) = **B**

In addition we will emphasize that **H** is also injective defined in a way.
Let get an example to define a pair in the correspondence of **H**:

$$\lambda N.\ N+1,\ \textit{injective function}$$
$$\lambda N.\ N+1\ \Xi\ 6250 = X$$
$$A = X\ (X) = (\lambda N.\ N+1)(6250) = 6251$$
$$B = X\ (X + A) = (\lambda N.\ N+1)(6250 + 6251) = 6250 + 6251 + 1 = 12502$$

For that reason we could say that there is a correspondence in **H** in the way **H**(**12502**)=**6250**. Otherwise, if we could find a **Y** for our encoding such that: **Y**(**Y**+**Y**(**Y**))= **12502** and **Y**≠**6250**, we would say that **H** won't find **Y**, because its definition must be injective *in a way*.

As a result, we know that this function makes sense because **X** always find configurations where for a coding system settings: **X** (cod (**X**)) = **A** and, on the other hand, **X** (**A** + cod(**X**)) = **B** so that **H** (**B**) will be defined, even it will be bounded if **X** was.

Now suppose that our coding is defined such that **H** (cod (**H**)) = cod (**H**), which is also easy to define because after calculating **H** (cod '(**H**)) is easy to change the coding on the other. Because encryption is independent of the way **H** has to be resolved in the Turing Machine (or in our notation of Alonzo Church, lambda-calculus). Ie **H** (**A**) = cod (**X**) and **H** (**A'**) = cod' (**X**) must refer to the same **X** even altering the way to encode it.

Therefore:
Step 1. **H** (**H**) = **H**, defining the selected coding.
Step 2. . **H** (**H**) = **A**, by **Rule 2**.
Step 3. . **H** (**H** + **A**) = **H**, by **Rule 3** combined with **Step 1**
Step 4. Contradicts: **H** is not injective, for **Steps 1** and **3**.

We know that, so being able to define **H** injective so we would only set cod(**H**) = 0, which questions the computing power of the machine to not be able to be defined within the natural and undetermined manner.

As **H** is linked to the encoding mechanism, then we cannot validate the existence of that function.

On the other hand, if there were a mechanism to determine which **C** holds for an **S** and **X** where **S** = **C** (**C XOR X**) for any encoding then we would be able to implement **H**. In any case the link with the dimension polynomial is easy to adhere to the demonstration for the required response. So the impossibility of the existence of **H** in a deterministic polynomial bounded Turing Machine, points **NP≠P**.

Without imposing this type of dimension, it can be argued more general statements about the impossibility of finding configurations from their outputs. Now, what this result tells us is that the algorithms zero-knowledge are valid, as long as you know how to implement them. For example, the **NP**-completeness of Cook should not be valid, because his theorem is proved with a formalist philosophy, that concept will be discussed later.

3 Formalist philosophy and # P = P.

As we mentioned at the beginning, formalism offers us more vague and general results. The inaccuracy comprises the inclusion of infinity like an enumerable value, or even it makes use of transfinite numbers thanks to endless expressions that allow us to work on some kind of algebra.

All of this will lead us to models with a long variety of degrees of accuracy. And it takes us to the possibility that certain statements completely change sign: whatever could be false before now could be true.

That's why we study **#P**: **#P** resolutions returns the number of solutions behind a statement which can be validated within a narrow polynomially Turing machine.

The formalism allows us to find a secure connection between these two classes if we accept as valid any correspondence whose existence is consistent and not in terms of what are we capable of constructing. That is, this way of answering questions will tell us that there will be a setup, but we do not know how to get it.

At this point, we will need to get something more close to the original notation of the Turing Machine: we will say that a Turing machine tape is composed of cells.

In each one of them we can read a symbol belonging to a finite set of symbols, which is its alphabet. The configuration of the machine consists of a description of what state to what state transitions, depending on the symbol read from a pointer to the ribbon, to determine whether to change the symbol or whether to move the pointer to the left or right.

3.1 Working with Boolean Formulas.

From this more original definition from Alan Turing, we will define what we want to solve: **#SAT** is equivalent to know how many solutions has a Boolean equation. To put it in our Turing machine we must stay with the definitions:

1. **Boolean variable**: Variable that takes the values 0 or 1, but not both.
2. **Literal**: Variable affirmed or denied within Boolean logic.
3. **Clause**: Sum of Boolean literals.

Any Boolean function can be expressed easily and quickly in product clauses: for this, every time we see an expression other than a sum between literals, then the expression is changed by a new variable that is equivalent to the expression. As the only operation that sum is not the product (for the denial of a product is the sum of its denied), then the process is quite simple.

So we will need to define a lemma:

$$(\mathbf{Z} = \mathbf{XY}) \leftrightarrow (\neg\mathbf{Z} + \mathbf{X})\ (\neg\mathbf{Z} + \mathbf{Y})\ (\neg\mathbf{X} + \neg\mathbf{Y} + \mathbf{Z}) \tag{8}$$

Table 3. Demonstration of (8).

Lemma 1: $(\mathbf{Z = XY}) \leftrightarrow (\neg\mathbf{Z + X})\ (\neg\mathbf{Z + Y})\ (\neg\mathbf{X} + \neg\mathbf{Y + Z})$

(Z = XY)				↔	(¬Z + X)			(¬Z + Y)			(¬X +¬Y + Z)			
Z	**X**	**Y**	**=**		**¬Z**	**X**	**+**	**¬Z**	**Y**	**+**	**¬X**	**¬Y**	**Z**	**+**
0	0	0	**1**	*1*	1	0	**1**	1	0	**1**	1	1	0	**1**
0	0	1	**1**	*1*	1	0	**1**	1	1	**1**	1	0	0	**1**
0	1	0	**1**	*1*	1	1	**1**	1	0	**1**	0	1	0	**1**
0	1	1	**0**	*1*	1	1	**1**	1	1	**1**	0	0	0	**0**
1	0	0	**0**	*1*	0	0	**0**	0	0	**0**	1	1	1	**1**
1	0	1	**0**	*1*	0	0	**0**	0	1	**1**	1	0	1	**1**
1	1	0	**0**	*1*	0	1	**1**	0	0	**0**	0	1	1	**1**
1	1	1	**1**	*1*	0	1	**1**	0	1	**1**	0	0	1	**1**

From that lemma is easy to demonstrate another:

Step 1. $(\mathbf{Z = XY}) \leftrightarrow (\neg\mathbf{Z + X})\ (\neg\mathbf{Z + Y})\ (\neg\mathbf{X} + \neg\mathbf{Y + Z})$, from (8)

Step 2. $(\neg\mathbf{Z} = \neg\mathbf{X}+\neg\mathbf{Y}) \leftrightarrow (\neg\mathbf{Z + X})\ (\neg\mathbf{Z + Y})\ (\neg\mathbf{X} + \neg\mathbf{Y + Z})$, *applying DeMorgan*

Step 3. $(\mathbf{Z' = X'+Y'}) \leftrightarrow (\mathbf{Z'} + \neg\mathbf{X'})\ (\mathbf{Z'} + \neg\mathbf{Y'})\ (\mathbf{X'} + \mathbf{Y'} + \neg\mathbf{Z'})$, by substitution

So that points to the equation (9).

$$(\mathbf{Z = X+Y}) \leftrightarrow (\mathbf{Z} + \neg\mathbf{X})\ (\mathbf{Z} + \neg\mathbf{Y})\ (\mathbf{X + Y} + \neg\mathbf{Z}) \tag{9}$$

In any event, you will have a very simple sums of a product which should take a value of **1**. And we wonder how many solutions has the formula.

For example, let's study the formula $\mathbf{X_1 \cdot X_2 \cdot (X_3 + \neg X_2 \cdot X_5 \cdot (\neg X_1 + X_3 \cdot X_4)) = 0}$. We want to convert it to a product of sums of literals whose result were 1.

First: We have to convert the result to 1.

$$\neg\mathbf{X_1} + \neg\mathbf{X_2} + \neg(\mathbf{X_3} + \neg\mathbf{X_2}\cdot\mathbf{X_5}\cdot(\neg\mathbf{X_1}+\mathbf{X_3}\cdot\mathbf{X_4})) = 1$$

Lastly: We will apply lemmas 1 or 2 on the product or sum of literals where needed.

$$\mathbf{X_3 \cdot X_4 = Z_1}$$

$$(\neg\mathbf{X_1} + \neg\mathbf{X_2} + \neg(\mathbf{X_3} + \neg\mathbf{X_2}\cdot\mathbf{X_5}\cdot(\neg\mathbf{X_1}+\mathbf{Z_1})))\cdot(\neg\mathbf{Z_1} + \mathbf{X_3})\cdot(\neg\mathbf{Z_1} + \mathbf{X_4})\cdot(\mathbf{Z_1} + \neg\mathbf{X_3} + \neg\mathbf{X_4}) = 1$$

To the new clauses we will call them $\boldsymbol{\alpha_1}$, so:

$$(\neg\mathbf{X_1} + \neg\mathbf{X_2} + \neg(\mathbf{X_3} + \neg\mathbf{X_2}\cdot\mathbf{X_5}\cdot(\neg\mathbf{X_1}+\mathbf{Z_1})))\cdot\ \boldsymbol{\alpha_1} = 1$$

Now we continue:

$$\neg\mathbf{X_1} + \mathbf{Z_1} = \mathbf{Z_2}$$

$$(\neg\mathbf{X_1} + \neg\mathbf{X_2} + \neg(\mathbf{X_3} + \neg\mathbf{X_2}\cdot\mathbf{X_5}\cdot\mathbf{Z_2}))\cdot\ \boldsymbol{\alpha_1}\cdot\ \boldsymbol{\alpha_2} = 1$$

$$\boldsymbol{\alpha_2} = (\mathbf{Z_2} + \mathbf{X_1})\cdot(\mathbf{Z_2} + \neg\mathbf{Z_1})\cdot(\neg\mathbf{Z_2} + \neg\mathbf{X_1} + \mathbf{Z_1})$$

Continuing with substitutions:

$$\mathbf{X_5 \cdot Z_2 = Z_3}$$

$$\neg\mathbf{X_2}\cdot\mathbf{Z_3} = \mathbf{Z_4}$$

$$\mathbf{X_3 + Z_4 = Z_5}$$

At last, we will conclude:

$$(\neg\mathbf{X_1} + \neg\mathbf{X_2} + \neg\mathbf{Z_5})\cdot\boldsymbol{\alpha_1}\cdot\ \boldsymbol{\alpha_2}\cdot\ \boldsymbol{\alpha_3}\cdot\ \boldsymbol{\alpha_4}\cdot\ \boldsymbol{\alpha_5} = 1$$

3.2 Demonstration #P=P.

To solve it in our setup formalist, we only have to enter in each cell a clause in full. So if our formula has 5 clauses then the size of the input could be exactly 5, the why is we will not need intermediate cells. Our alphabet, therefore, must decode the variables **V** in 3^V combinations representing a clause in full, plus the relevant symbols to manage the output of the machine. The configuration will perform a linear path to meet an invariant: in the current state of the same subscript solutions encrypt all clauses read so far.

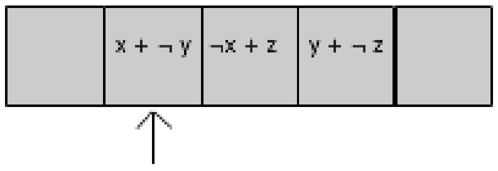

Fig. 1. A tape which alfabet are clauses.

That is why we must label the states of the Turing machine with an aditional integer: the subscript encodes a binary matrix whose columns are the variables and as whose rows are each possible solution to the clauses that have been read.

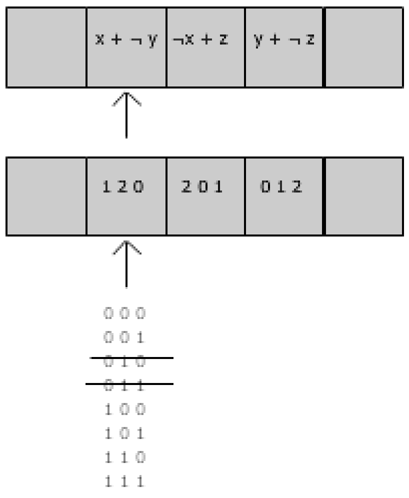

Fig. 2. How to get encode the query.

In **Fig. 2** we can see each clause is transformed in a tuple of three digits where **1** means literal is affirmed, **2** means the literal is negated an **0** means literal does not appear in the clause. So if the order is (**x y z**), then $\boldsymbol{x} + \neg \boldsymbol{y}$ is represented as (**1 2 0**).

Our query must be represented by a number which will encode the matrix bellow. In the matrix we see a column for each variable (**x**, **y** and **z**), and we can see some rows crossed out. Those rows are the combination that does not match with the clause pointed ($\boldsymbol{x} + \neg \boldsymbol{y}$). So we can count 6 solutions from the matrix encoded in our query.

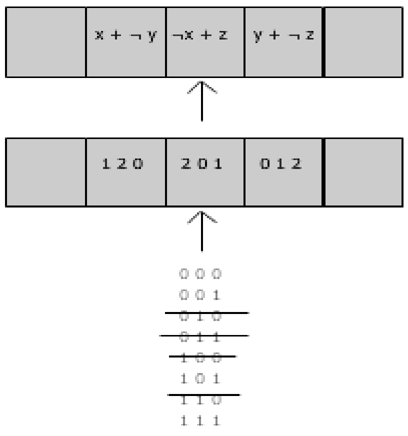

Fig. 3. How the query changes.

As we can see, if **c** is the number of columns, then the number of states must be 2^c. And if V is the number of variables, then **c** = 2^V. We will study this more clearly with an example later.

When the pointer moves to the right to read the next clause, we have to intersect rows of the actual matrix with the new one pointed. So if the new clause obligates us to cross out some rows, then we only have to do it on the matrix of the query. Thus is the why we have defined a state transition function consistent with the required solution.

In our example in Fig 3, we can see how the clause of the second cell obligates us to cross out two new rows. That is the why the number of solutions of the formula (***x*** + ¬***y***) (¬***x*** + ¬***z***) is four. So if we want to know how many solutions the entire formula have got, we only have to let the algorithm continue until the end.

Because the size of the alphabet and the number of states is not related to the complexity of the decision, the solution of **#SAT** is linear. In fact, the result (the number of solutions of the formula) can be printed after reaching a blank cell.

If someone tells us that a mechanical machine could not opperate with such number of states, because it had needed to activate a great number of clevers. I will respond him "*yes*" but, under our formalism philosophy, we can generate a very complicated machine with a great constant which will ensure us that problem can be solved in a linear time.

To get a better understanding, I will give you an example of how to see this no so-mechanical Turing Machine:

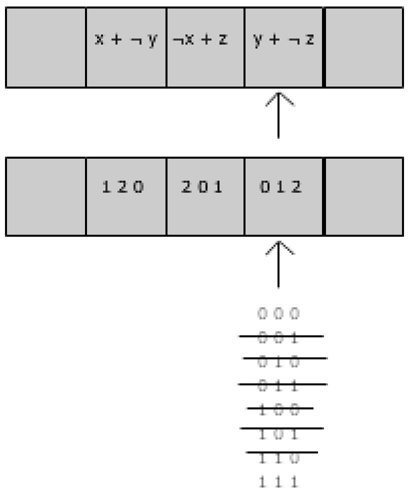

Fig. 4. How the algorithm ends.

Imagine someone shows us in a piece of paper the formula shown in **Fig. 1**. To respond how many solutions validate that formula, we will use a manual. How big the manual must have? Our configuration of Turing Machine tries to solve that boolean formula of **V** variables, so if $\mathbf{c} = 2^V$ then we will need 2^c pages in that book with 3^V lines in a page. That means that in our example we will need a book of **256** pages with **27** rows in each page. Every page is of the format

“*expression* → *page*”.

That means that in that page if we read exactly the clause as the *expression* then we will have to reach the *page* in the manual, for reading the next token.

After reading each clause we will know how many cases match with that page, because that number is printed in every page.

So if our philosophy is able to construct that kind of machine, then we are able to solve **#SAT**.

Someone could say that kind of machine is impossible to construct..., well, perhaps not at all. And the respond would be answered in superconductors, spintronics or quantum computing for instance.

This result tells us that **#SAT** is a **P** class, from a formal point of view. However, the curious result goes far beyond.

Because *Cook's theorem* were defined in a formalistic way, we know that an **NP** language can be expressed by **SAT** in a Turing machine. So from this point of view we might well say that it is shown that **#P** = **P**. However, to reach this conclusion must overcome a double inaccuracy: the existence of a map of any **NP** to **SAT**, and the existence of the correspondence described herein.

Another thing to consider is the existence of machines about tend to produce results increasingly polynomial. That is, if there is a correspondence **P** that solves a **#P**, so it could not be unreasonable to think that there could be slow training mechanisms which would help us pull together to resolve those configurations.

One consideration to take into account is the fact that it has taken to establish a maximum number of variables. This means that if the problem would have taken any number of variables, then if we want to solve it for any type of entry and configuration it would have needed to use the "trick Cook" to solve the other problems.

Conclusions

Until this point, we can understand the same question is able to generate two answers. But what a lot of scientists do not know is that a philosophy by itself is not enough to get the expected result. We can combine different philosophies avoiding contradictions.

A good example is in the interpretation of the cycle of life of the Science. Kuhn thought Science works in paradigms. In the point of view of Kuhn, the beginning of a new paradigm destroys the paradigm before. In example, the paradigm opened by Einstein destroyed the paradigm of Newton to create a new Physics. That view is equivalent to the philosophy of the formalist: you can construct every model that is coherent. So finding a contradiction is the only way to change the paradigm. But, of course, that model was criticized considering we nowadays live better than in the Middle Age. That is the why formalism is not complete.

In other hand, constructivism is not complete neither: we can considerate the falsationism of Popper . That author ensured every concept which could not be denied by a demonstration could not be accepted. That way of allowing asserts is equivalent to be capable of constructing demonstrations. So that philosophy is really constructive: because you need to create a demonstration in a language to allow an affirmation. That was different to the paradigms of Kuhn because Kuhn allowed every coherent paradigm. But the fact is Popper could not accept easy things like Ethics (something too trivial to get worried). People could allow a moral stronger than allowing some new researches in a laboratory: so the constructivism is not complete too.

American Open Computer Science Journal
Vol. 2, No. 1, July 2015, pp. 1-11
Available online at http://rekpub.com/Journals.php

There must be an intelligent merge between the constructive machine and the formalist one. But to get the correct conclusions, how does the formalist machine look like? How is made the oracle machine described by Alan Turing? We can imagine a digital system solving constructive problems but, how can get constructed a formalist machine?

The response to that problem could be easier than we would though if we study all the ways we have to find Turing Machines from different sources. The best example could be our proper life: too far to be digital. So we need a notation that, working under a Turing Machine, will get us the benefits of the formalist philosophy with materials existed on Earth.

That example is in the Dirac notation: imagine we have some sub particles that are distributed like in **Fig. 5**. As we can observe, some doors set up an output per mutating the initial state to another without loosing information. In this way, if we could determine the energy of the wave of the last qbit, we would calculate, dividing a constant, #SAT. Does exist a material able to give us that kind of information? – Of course, it is called superconductor.

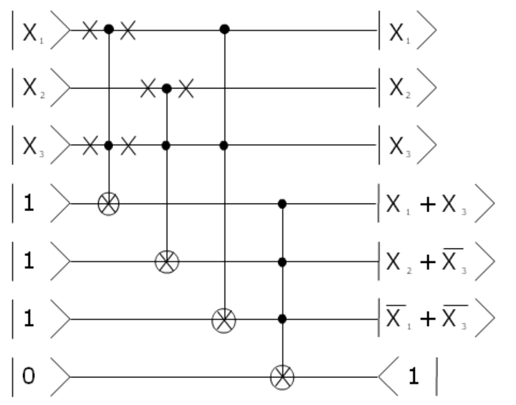

Fig. 5. Example of relation between #((X_1 + X_3) (X_2 + ¬X_3) (¬X_1 + ¬X_3) = ***1***) and quantum machines.

Our experience tells us there is no quantum computers of too many qbits; and experimental observations nowadays requires exponential time to collapse the state..., but I even have a continuation of this document in that direction. What we only need to understand before is the difference between digital electronic and spintronic to see how the cybernetic works.

So the intelligent relation between semiconductors and superconductors can give us a machine whose design will calculate so fast as a Personal Computer analyzing better than the best live logician on Earth.

References

[1] Michael R. Garey / David S. Johnson: Computers and Intractability. A guide to the theory of NP-Completeness.
[2] S.A. Cook : http:// www.claymath.org/millenium/P vs NP/
[3] Harvey M. Friedman: Clay Millenium Problem P = NP. Mathematics Colloquium in Ohio State University October 20, 2005
[4] Alan M. Turing : On computable numbers, with an application to the entscheidunsproblem.
[5] Ludwig Wittgenstein: Sobre la certeza.
[6] A. Pérez de Laborda: La ciencia contemporánea y sus implicaciones filosóficas.