**P versus NP & SAT problem**

**T.M Ahad**

March 19, 2024

*‘P versus NP – a gift to mathematics from computer science’*

*-Stephen Smale*

**Abstract**

‘**P versus NP**’ problem is known to be a very hard problem in computer science and mathematics.

On this research paper,

This research paper delves into this complex issue by exploring the intricacies of the ‘**NP** Complete’ problem known as the Boolean satisfiability problem, commonly referred to as **SAT**. Through a comprehensive examination of **SAT** and its implications, this paper aims to find a polynomial-time solution or algorithm for **SAT**.

which, if proven to be flawless, would establish the equality of the complexity classes **P** and **NP**. which is enough for a groundbreaking solution for the ‘***p* versus *np***’ problem

**Contents**

|  |  |  |
| --- | --- | --- |
|  | ‘P vs np’ problem explained | **3-4** |
|  | ‘**SAT**’ problem explained | **4** |
|  | ‘**SAT**’ theory | ***5-11*** |
|  | Time complexity of the LRC algorithm | ***11-12*** |
|  | Proving ‘p is equals to np’ | **13** |

**P vs NP problem**

‘**P** versus **NP**’ problem is known to be a very hard problem in computer science and mathematics.

It asks if problems that can be solved in exponential time and the solution can be verified in polynomial

can be solved in polynomial time. We will discuss about the problem later on this research paper.

Here, some things that should be clarified:

**Solved in exponential time**:

Solved by exponential time means that the time complexity of the algorithm.

To solve the problem can be represented as an exponential function.

Exponential functions are functions that have variables as the exponent.

**Example:**

In this example, function *f* is an exponential function which has the variable a as the exponent.

**Solved in polynomial time:**

Solved by exponential time means that the time complexity of the algorithm.

To solve the problem can be represented as a polynomial function.

polynomial functions are functions that have variables as the base.

**Example:**

In this example, function *f* is a polynomial function which has the variable a as the exponent.

**NP Complete Problems:**

Any **NP** problem could be reduced to a **NP** Complete Problem.

Exponential time algorithm’s time to perform the task can grow exponentially compared to polynomial time algorithms.

**P** is the set problems that can be checked and solved in polynomial time. And NP is the set of problems

That the solution can verified in polynomial time but solved in exponential time.

‘**P** versus **NP**’ problem is included in the seven millennium problems provided by the Clay Mathematics Institute (CMI). Those 7 problems were never solved in the history. And each problem is allocated with

1 million USD for the solution. One of these problems ‘Poincare conjecture’ was solved by brilliant Russian mathematician ‘Grigori Perelman’.

**SAT Problem**

**SAT** problem (Boolean satisfiability problem) is a known **NP** Complete problem.

So, Any **NP** Problem could be reduced to a **SAT** problem.

That means, if anyone can find a polynomial time algorithm for solving the **SAT** problem.

Every **NP** problem can be reduced to a **SAT** problem and that can be solved in polynomial time. So, every NP problem can be solved in polynomial time. And every problem that can be solved is polynomial time is a problem that belongs to **P**. That proves the statement

‘**P** is equals to **NP**’.

This problem asks if, for a any given Boolean expression can any assignment make this expression true.

A solution for this problem would be a ‘**Brute Forcing**’ algorithm.

This algorithm will go through every assignment and evaluate the expression using the assignment. If any evaluation gives true. Then the algorithm says ‘Yes, The expression is satisfiable’.

If it does not find any assignment where the expression is true. Then the algorithm says,

‘No, the expression is not satisfiable’

But this algorithm is not a polynomial time algorithm. Because as the number of variables grows in the expression the time taken to solve it grows exponentially.

Here is the time complexity of this ‘**Brute Forcing**’ algorithm:

‘a’ is the number of variables used in that expression.

‘g’ is the number of gates used in that expression

**A polynomial time algorithm for solving the SAT problem**

This algorithm was invented by the author of this research paper **T.M Ahad**

It is called the LRC (Lowest requirement calculator) algorithm.

LRC can only solve Boolean expression with AND, OR and NOT gate but every other gate can be constructed with these three gates.

So, Let’s solve a Boolean expression using LRC. Here is the Boolean expression

**((A AND B) OR (C AND (D OR E))) AND (F OR G)**

**First Step: Parsing:**

At first, LRC algorithm parses the expression to a tree of gates.

Which is done in a polynomial time.

Here is the tree for our input.

**The tree is on the next page**

**Figure:** 1.0 Logical Gate Tree

**Final Step: Analyzing and Solving:**

After, parsing the expression into a tree of logical gates. It traverses through the tree.

Calculates the Least requirement (LR) to the gate to be true for every gate.

This calculation needs a new theory called the **SAT** theory. It is a theory that works with sets and calculates the Least, Exact, Highest requirement for a logical Boolean gate to be true.

Let’s review and learn the fundamental of this theory.

**Example Tree**:

In this figure, there is a OR gate with variables d and e. If this OR gate is named ag.

Then, to represent the dependency variables of ag using the **SAT** theory,

We can write this,

In ‘**SAT**’ theory, the ‘set’ operator is used for representing the dependency and the ‘set’ operatorresults a set of strings which is the name of those variables.

The variable set of an **AND** gate is the union of two input gate’s variable set. The variable set of an **OR** gate is the variable set of the gate with the least requirement among two inputs. The variable set of an **NOT** gate is the variable set of the input gate.

To represent the least requirement of being variables being true (LR) of out gate using the ‘**SAT**’ theory, this expression can be used

Let’s break down this expression, ‘lr’ operator is used to represent the least requirement of a gate & and the function ‘l’ is representing the least requirement.

The function ‘l’ works like this,

|  |  |  |
| --- | --- | --- |
| 1. |  | Least ‘n’ numbers of element to be true |
| 2. |  | Least ‘n’ number of element to be false |

So now can understand, the least requirement for gate ‘ag’ to be true is least 1 element to be true from the dependency set.

So, to be gate ‘ag’ true at least, the assignment needs one of the elements from its dependency set (d or e) need to be true. **SAT** theory also have several functions for representing least, exact, highest requirement.

Here is, all of them

|  |  |  |
| --- | --- | --- |
| 1. |  | Least ‘n’ numbers of element to be true |
| 2. |  | Least ‘n’ numbers of element to be false |
| 3. |  | All of the elements to be true |
| 4. |  | All of the elements to be false |
| 5. |  | None of the elements to be true |
| 6. |  | None of the elements to be false |
| 7. |  | Exactly ‘n’ numbers of element to be true |
| 8. |  | Exactly ‘n’ numbers of element to be false |
| 9. |  | Maximum ‘n’ numbers of element to be true |
| 10. |  | Maximum ‘n’ numbers of element to be false |

Now, Let’s discuss about requirement arithmetic in ‘**SAT**’ theory. I will mostly discuss about lowest requirement.

**Requirement Arithmetic (SAT theory)**

Requirement arithmetic is a part of the ‘**SAT**’ theory. It is an arithmetic that performs operations like AND, OR, NOT on requirments of logical gates to be true. Let’s get a figure of a Boolean logic gate tree and get an understanding of what is **RA** (Requirement arithmetic).

**The tree is on the next page**

**Example Tree**:

**Figure:** 2.0 Logical Gate Tree

Using **RA**, it can be found if this tree is satisfiable or not. Let’s call the OR gate og, AND gate ag and the NOTgate ng. First define, the least requirement of the and gate. Then we will define the least requirement of other gates using **RA.**

The least requirement of ag be true is defined. Now, make the NOT operation on the least requirement of ag. which can be expressed like this.

Before calculating **LR**. Let’s look at what happens when **RA** operator **NOT** is applied to a “L” or “least” function

Now, back to our calculation

Here b is the Boolean wither true or false. So, in this case the least requirement of the NOT gate is ‘n(true)’. Or, to be the NOT gate true, all of the variables (e & a) that is used to perform the NOT operation should be false.

Now, define the least requirement for the OR gate. It should look like this

Here, we applying the **RA**’s OR operator to find out the least requirement to og to be true. The OR operator of **RA** when applied results the least requirement among the two operands.

There are two operands one is the least requirement for the gate which is ‘’ and the other one (‘ur’) is the requirement of a single variable named d.

The opposite of ‘ur’ is ‘nr’ or ‘Non unit requirement’. We can express this like this.

This requirement is a special type of requirement in **SAT** theory. It is called the ‘Unit Requirement. This requirement ‘ur’ can be expressed as ‘’, ‘’, ‘’, ‘’ or ’ as there is only one variable that the gate can depend on.

Back to our calculation. According to **SAT** theory ‘Unit Requirement’ is the lowest requirement possible beside of “Nr” (Non requirement). I shall talk about “None requirement” later. This theory can be expressed as this.

The equation is expressing that ‘Unit requirement is lower than r where r belongs to the universal set of requirements (rs) or any requirement possible but unit requirement.

As we know, when OR gate is applied to two requirements it results the lowest requirement among the them. So. For our case the result in unit requirement or ur.

Now I shall express the least requirement of the root gate of the tree of **Figure** 2.0.

But, there still a problem the problem is accuracy. Let’s review another LGT (Logical Gate Tree) for this.

**Example Tree**:

**Figure:** 2.1 Logical Gate Tree

Here, if we calculate the **LR** of the AND gate it would be ‘’. it works for the assignment

Let’s name the AND gate on figure 2.1 ag.

‘{e: true, a: false}’ but not for assignment ‘{e: false, a: true}’. But both assignment satisfies the lowest requirement we calculated. It happens because the and gate is asymmetrical.

To fix this, **SAT** theory introduces **SATES.**

**SATES**

**SATES** is a way to add accuracy to the requirments for asymmetrical logical gates. For asymmetrical logical gates, often **RA** calculation is not enough to correctly express the lowest requirement of the logical gate.

**SATES** is a map data structure like a set. A map is just pairs of key-value mapped together.

**SATES** are extra assignments that is needed to be included in the satisfiable assignment for the gate to be true. Let’s express the requirement of the gate using **SATES.** Cause, the gate is an asymmetrical logical gate.

Using **SATES,** we can express that to ag to be true it needs at least one variable to be true & a must be false. The extra statement ‘|a: false|’ is a **SATES.** Any assignment that makes a gate true must include the **SATES** of the gate. So, in an assignment where a is true. This assignment would not be result true if it would be applied to the gate ag.

If two gate’s **SATES** shares at least one variable and have different values. Then, these two gates are not co-satisfiable or is not satisfiable together in the same context. This statement can be expressed as this

Now, I shall clarity some things of the expression I mentioned. There are two gates and . By, ‘’ it means it’s the set of the keys in the **SATES** of and ‘’ means the set of the values of the **SATES** of the gate . Same for . ‘’ is an expression that uses map indexing where it is indexing all of the common keys of **SATES** of and into the values of the **SATES** of and making the set of values. Same for .

**Note**: Operator like ‘’, ‘’ and is used differently in **SAT** theory rather than classical mathematics.

**Calculating Satisfiability**

We’ve learned the fundamentals of the **SAT** theory. But I have not talked about calculating satisfiability using **SAT** theory. Let’s calculate satisfiability using **SAT** theory.

First, look at the satisfiability for a **OR** gate.

Now, I shall explain it. This equation is expressing the intersection of keys of **SATES** of the two inputs gates is an empty set satisfiability for a **OR** gate. If any of the input gates are satisfiable then the **OR** gate is satisfiable.

Now, I shall express the satisfiability of the **AND** gate satisfiability.

Now, I shall explain it. There are two scenarios, where the **AND** gate is satisfiable. The **AND** gate is satisfiable where the intersection of keys of **SATES** of the two inputs gates is an empty set. The second scenario is where the intersection of keys of **SATES** of the two inputs gates is not empty set but the key-value pairs of the **SATES** of those two inputs gate does not collide. And, in both scenarios, the **RA** operator **NOT** applied on least requirement of one of those two gates does not equals to other gate’s least requirement. There is another scenario, where the variable set of two input gates does not intersect. Where the satisfiability is true where two input gates are satisfiable.

There is also another scenario, that is too complex to represent as an expression. Where if choose the least number of variables needed to true to be the input gate to be true from the two input gates. The chosen two sets would always intersect.

In this case, the **AND** is satisfiable where, the **RA** operator applied on the LR of one of the input gates does not results the other gate’s **LR** and the **SATES** of the two input gates does not collide.

Now, Let’s express the satisfiability of the **NOT** gate.

To understand the satisfiability of **NOT** gate. we have to understand the lowest of lowest requirement possible the “Nr” or “None requirement”. “None requirement” can be expressed as several expression

Here is some of them

**The Lowest requirement**

“None requirement” is the requirement of a gate that is always true no matter what assignment is applied. It would always result true. That’s why it is named “None requirement” or there is no requirement to be the gate to be true

There is also the opposite of Nr which is pr or “Not possible requirement”. Here is the expression for expressing as the highest requirement.

**The Highest requirement**

requirement is the requirement of a gate that is not satisfiable. The relation of and Nr can be expressed as this

Or this,

Now, I shall express the satisfiability of the **NOT** gate.

Now, I shall explain it. **NOT** gate applied to a gate would not be satisfiable only if the least requirement of the gate is . But, on all of the other case, the **NOT** gate would be satisfiable.

After, learning about calculating satisfiability. We can go back to our original **LRC** solver. And solve the given tree in polynomial time.

With the calculation of satisfiability in **SAT** theory, the algorithm can start calculating satisfiability for the deep branches of logical gates and then calculate the higher once & eventually find if the root gate is satisfiable or not which is done in a polynomial time.

Let’s discuss about the time complexity of this algorithm to prove that this problem solves SAT problem in polynomial time.

**Time complexity (LRC algorithm)**

If n is considered the number of logical gates used in the expression. The time complexity would be . Because, for every gate the algorithm calculates the satisfiability. And makes the variable set by picking or merging variable sets.

But if n is considered the number of variables used. We have to calculate the number of gates to find the time complexity. The least number of gates is where there is no **NOT** gate. for this type of gate, the time complexity calculated for n as the number of variables used is the lowest. let’s express relation between the number of variables and the gate numbers where no **NOT** gate is used.

So, the time complexity for the best case for the LRC algorithm where is the numbers of variable is expressed here

If was infinite and the time complexity is an infinite series then the time complexity would be. If we round this summation figure we get this

The following expression is for where the depth is not big. Here is the expression where the depth of the tree parsed from the binary expression is bigger.

For bigger depth, is slightly bigger than

Now, I shall calculate the worst case which is the case where maximum **NOT** gates are used. here, I expressed the relation between the number of gates and the maximum **NOT** gates possible in numbers of gate.

**The expression is on the next page**

So, the time complexity is . Cause, the way to get maximum **NOT** gates are use not gates between a **OR** or **AND** gate**.** but the **NOT** gate does not take two inputs. So, the number of gates remains the same as the one lower depth gates.

Here is the time complexity expressed.

After rounding up this figure we get these two expressions

**Smaller depth**

**Bigger depth**

With the worst of worst case and best of best cases every time the LRC algorithm solves a **SAT** problem in polynomial time.

**Solution for the P versus NP problem**

With the introduction of the polynomial time solution for the SAT problem, exemplified by the LRC algorithm, I boldly assert a significant leap towards resolving the elusive P vs NP problem.