https://dx.doi.org/10.22161/ijaers.73.34 *ISSN: 2349-6495(P) | 2456-1908(O)*

Revisiting Cook-Levin theorem using NP-Completeness and Circuit-SAT

Edward E. Ogheneovo

Department of Computer Science, University of Port Harcourt, Port Harcourt, Nigeria.

Abstract**—** *Stephen Cook and Leonard Levin independently proved that there are problems called NonPolynomial-complete (NP-complete) problems. The theorem is today referred to as Cook-Levin theorem. The theorem states that Boolean satisfiability problem is NP-complete. That is to say, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable. Therefore, if there exists a deterministic polynomial time algorithm for solving a Boolean satisfiability, then there exists a deterministic polynomial time algorithm for solving all problems in NP. Thus Cook-Levin theorem provides a proof that the problem of SAT is NP-complete via reduction technique. In this paper, we revisit Cook-Levin Theorem but using a completely different approach to prove the theorem. The approach used combines the concepts of NP-completeness and circuit-SAT. Using this technique, we showed that Boolean satisfiability problem is NP-complete through the reduction of polynomial time algorithms for NP-completeness and circuit-SAT.*

Keywords**— Cook-Levin, Boolean satisfiability, circuit-SAT, NP-complete, polynomial time.**

I. INTRODUCTION

In the 1970s, Stephen Cook and Leonard Levin independently discovered that there are problems in NP whose complexities are related to all other problems in NP. Those problems are called NP-complete problems [1] [2]. Cook-Levin theorem states that Boolean satisfiability problem is NP-complete. That is to say, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable [3]. Thus if there exists a deterministic polynomial time algorithm for solving a Boolean satisfiability, then there exists a deterministic polynomial time algorithm for solving all problems in NP. Therefore, the question of whether such an algorithm exists is called the P versus NP problem. However, it must be noted that the P vs NP problem is considered the most important unsolved problem in theoretical computer science [4] [5].

A deterministic computer is a machine that can solve a problem and provide correct answer whereas a non-deterministic computer is the one that can "guess" the right answer or solution. If a solution exists, computers will always guess it. One way to imagine it is by using a parallel computer that can freely spawn an infinite number of processes by using one processor on each possible answer or by using a machine in which all the processors try to verify that their solution works especially if a processor finds out that it has a working solution [6] [7] .NP can also be thought of as the class of problems 1) whose solutions can be verified in polynomial time, or (2) that can be solved in polynomial time on a machine that can pursue infinitely many paths of the computation in parallel. However, there also exist problems which are not NP. If a solution is known to NP-complete it can be reduced to single polynomial-time verification. A problem is NP-complete if it is NP and an algorithm for solving it is translated into one for solving any other NP-problem. There are many complexity classes that are much harder than NP. These include: PSPACE, EXPTIME, and undecidable problems [8] [9]. It is widely believed that P ≠NP. The question then is this: "Is P = NP? However, despite the wide belief, it has not been logically proved that P =NP. P implies polynomial time solvable while NP can be used to verify the correctness of a solution in polynomial time. *Mathematically*, it has not been established anywhere in literature that P = NP, although in recent times, modern mathematically equivalent definitions via efficient verification of purported solutions have been established.

https://dx.doi.org/10.22161/ijaers.73.34 *ISSN: 2349-6495(P) | 2456-1908(O)*

Thus Cool-Levin theorem provides a proof that the problem of SAT is NP-complete via reduction technique. Using the notion of reducibility of one problem to another, we say that a problem B can be reduced to A, $B \leq_p A$ if given access to a solution A, we can solve B in polynomial time using polynomial many calls to this solution for A. The notion of reducibility is very important to completeness of a problem for a class. If a problem L ∈ NP satisfies that for L[1]∈ NP satisfies that for proving NP-completeness by using Karp reduction.

II. THE P VS NP PROBLEM

A problem is said to be a polynomial problem if the number of steps needed to solve it is bounded by some power of the problem's size. Polynomial-time is one of the most fundamental complexity classes. It contains all decision problems that can be solved by a deterministic Turing machine in polynomial-time. That is, it is the class of decision problems that can have polynomial-time deterministic algorithms. Thus an algorithm exists for its solution such that the number of steps in the algorithm is bounded by a polynomial function of n, where n represents the length of the input for the problem. Hence, polynomial problems are said to be easy or tractable (i.e., a class of problem that is efficiently solvable) [10].

Definition:*A problem ispolynomial-time if the number of steps required to complete the algorithm for a given input is O(n^k) for some non-negative integer k, where n is the complexity of the input.*

Therefore, the class of P-problems is a subset of the class of NP-problems.An algorithm is said to be solvable in polynomial-time if the number of steps required to complete the algorithm for a given input is O(n^k) for some non-negative integer k, where n is the complexity of the input. Most mathematical operations such as addition, subtraction, multiplication, and division, as well as computing square roots, power, and logarithms, can be performed in polynomial time. It must be noted that computing the digits of most interesting mathematical constants, including π and *e*, can also be done in polynomial time [11] [12]. On the other hand, a problem is said to be NP-problem if it permits a nondeterministic solution and the number of steps to verify the solution is bounded by some power of the problem's size. They are class of decision problems that is solvable in polynomial-time on a nondeterministic machine or with a non-deterministic algorithm where non-deterministic simply means "guessing" a solution. A problem is in NP if a solution exists. Problems in NP are relatively easy if and only if we could "guess" the right solution, we could quickly test it [13] [14].

2.1 NP-Completeness

NP-complete problem is very important in the study of algorithms. The important thing about NP complexity class is that problems within that class can be verified by a polynomial time algorithm. A problem is classified as NP-complete if it can be shown that it is both NP-Hard and verifiable in polynomial time. However, there are some problems in which polynomial time solutions do not exist. NP-complete problems are a set of problems in which any other NP-problem can be reduced in polynomial time, and whose solution may still be verified in polynomial time. Problems in polynomial-time are said to be tractable these problems are because solvable and their solutions can be shown. NP-complete problems cannot be solved in polynomial time in any known way. They are therefore referred to as intractable problems. Examples of NP-complete are include the Hamilton Cycle and traveling salesman problems [15].A problem is said to be NP-complete when it is both in NP and NP-hard. An NP-complete problem encodes simultaneously all problems for which a solution can be efficiently recognized (i.e., a "universal problem"). The question is: can such a problem really exist? Cook [16] and Levin [17] both independently showed in their works that NP-complete problems exist. Also, Therefore, it is difficult to determine whether or not it is possible to solve a problem quickly in polynomial time. The NP-completeness can be thought of as a way of making the big P = NP question equivalent to smaller questions about the hardness of individual problems. Therefore, if we believe that P ≠NP, and we are able to prove some problem is NP-complete, then such a problem does not have a fast algorithm.

Definition: *A problem X is NP-complete if 1) X is in NP, and 2) every problem in NP is reducible to X in polynomial time. X can be shown to be NP by demonstrating that a candidate solution to C can be verified in polynomial time.That is, a problem is NP-complete if it is NP and an algorithm for solving it is translated into one for solving any other NP-problem*

NP-complete problems are defined in a precise sense as the hardest problems in polynomial time. Although, as stated

https://dx.doi.org/10.22161/ijaers.73.34 *ISSN: 2349-6495(P) | 2456-1908(O)*

earlier, it has not been proved that there is any problem in NP that is not in P. However, we can point to an NP-complete problem and say that if there is any hard problem in NP, that problem is one of the hard ones. Conversely, if everything in NP is easy, those problems are easy. NP-complete problems are in NP. That is, the set of all decision problems whose solutions can be verified in polynomial time. NP problems can be defined as the set of decision problems that can be solved in polynomial time using a non-deterministic Turing machine. A problem *p* in NP is NP-complete if every other problem in NP can be transformed (or reduced) into *p* in polynomial time.

III. THE CIRCUIT-SAT

[The circuit-SAT (CSAT) [21]is a decision problem of determining if a given Boolean circuit has an assignment of its inputs that makes the output true. That is, it asks whether the inputs to a given Boolean circuit can be consistently set to 1 or 0 such that the circuit outputs 1. If this is the case, the circuit is said to be satisfiable. Otherwise, the circuit is unsatisfiable. Thus the circuit-SAT asks that given a Boolean circuit C, is there is an assignment to the variables that causes the circuit to output 1?

3.1 Satisfiability Problem

The SATISFIABLE (SAT) problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. SAT is the problem of deciding if there is an assignment to the variables of a Boolean formula such that the formula is satisfied [18]. That is, SAT try to find out if the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE such that the formula evaluates to TRUE. If this can be established, then the formula is said to be satisfiable otherwise it is unsatisifiable. For example, a SAT problem can look as follows:

(a ∨ b ∨ c) (a ∨ ***b***) (b ∨**c**) (c ∨**a**) (**a**∨**b** ∨ **c**)

The above is a Boolean formula in conjunctive normal form (CNF). The formula contains a collection of clauses (in brackets), with each clause consisting of the disjunction (logical or (denoted V)) of several Boolean variables such as (a) or negations of one such as **(b)**. A SAT assignment must evaluate to true or false. The SAT problem is: Given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists. Thus the main idea about SAT is that an expression exists in conjunctive normal form, which is a way of saying that there are a series of expressions joined by ORs that must be TRUE. For example:

(a ∨b) AND (b ∨**c**) AND (d ∨**e**∨ f)

Definition:*A Boolean φformula is defined over a set ofproposition variables*$x_1, \ldots, x_n$*, using the standard propositional connectives* $\sim, \vee, \wedge, \rightarrow, \leftrightarrow$ *and parenthesis. The domain of propositional variables is* {0, 1}.

Definition: *A formula φ in conjunctive normal form (CNF) is a conjunction of disjunctions (clauses) of literals, where a literal is a variable or its complement. A formula is satisfied if all its clauses are satisfied. A formula is unsatisfied if at least one of its clauses is unsatisfied.*

3.2 The 2-Satisfiability (2-SAT) Problem

The 2-satisfiability (2-SAT) problem can be described as the problem of determining whether a collection of Boolean variables with constraints on pairs of variables can be assigned values satisfying all the constraints. The "2" in this name stands for the number of literals per clause, and "SAT" stand for satisfiable, a type of Boolean expression. 2-SAT is a special case of the general SAT problem and can be solved in polynomial time (i.e., tractable problems) in which the running time is upper bounded by a polynomial expression in the size of the input for the algorithm in which $T(n) = O(n^k)$ for some constant k. Generalized SAT problems involve constraints on more than two variables, and of constraint satisfaction problems, which can allow more than two choices for the values of each variable[19] [20] .

2-SAT problems are examples of NL-complete problems, i.e., problems that can be solved non-deterministically using a logarithmic amount of storage.NL-complete problems are among the hardest and the most difficult solvable problem in computer resource bound. A 2-SAT problem can be described using a Boolean expression with a special restricted form: a conjunction or disjunctions, where each operation has two arguments that may either be variables or the negations of variables. These variables or their negations appearing in this formula are referred to as literals and the disjunctions of pairs of literals are referred to as clauses. As an example, consider the following conjunctive normal form (CNF), having seven variables (i.e., x_0, x_1, …, x_6) and eleven clauses.

$$(x_0 \vee x_2) \wedge (x_0 \vee \sim x_3) \wedge (x_1 \vee \sim x_3) \wedge (x_1 \vee \sim x_4) \wedge (x_2 \vee \sim x_4) \wedge (x_0 \vee \sim x_5) \wedge$$

$$(\sim x_1 \vee \sim x_5) \wedge (x_2 \vee \sim x_5) \wedge (x_3 \vee x_6) \wedge (x_4 \vee x_6) \wedge (x_5 \vee x_6)$$

https://dx.doi.org/10.22161/ijaers.73.34 *ISSN: 2349-6495(P) | 2456-1908(O)*

Figure 1 shows an example of the use of literals and clauses in determining the truth or falsity of an assignment statement. Thus the SAT problem is a problem for determining a truth assignment to these variables that makes a formula of this type true: we must choose whether to make each of the variables true or false, so that the assignment sets at least one literal to true in every clause.

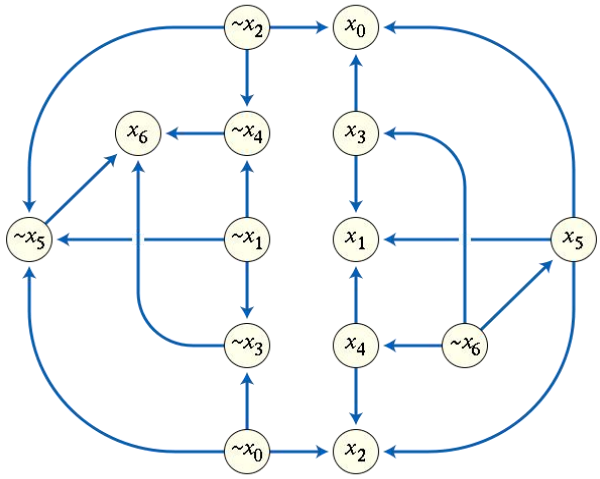

Fig. 1: An example of the use of literals and clauses.

One possible satisfying assignment for the expression above is the one that set all seven of the variables to true. Thus the 2-SAT instance represented by the expression is satisfiable. Formulas of the form described above are called 2-CNF formulas: the "2" in this name stands for the number of literals per clause, and "CNF" stand for Conjunctive Normal Form, a type of Boolean expression in the form conjunction of disjunctions. Each clause in a 2-CFN formula is logically equivalent to an implication from one variable or negated variable to the other. For example,

$$(x_0 \vee \sim x_3) \equiv (\sim x_0 \Longrightarrow \sim x_3) \equiv (x_3 \Longrightarrow x_0)$$

However, it must be noted that a 2-SAT instance may be written in implicative normal form because of the equivalence between these different types of operations. In this case, each operation is replaced in the CNF by both of the two implications to which it is equivalent. The 2-SAT can also be explained using graphical illustration by using implication graph. An implication graph is a directed graph in which there is one vertex per variable or negated variable, and an edge connecting one vertex to another whenever the corresponding variables are related by an implication in the implicative normal form of the instance. The implication graph is skew-symmetric directed graph G(V, E) having vertex set V and directed edge E. In this graph, each vertex in V represents the truth status of a Boolean literal, and each directed edge from vertex u up vertex v represents the material implication. An instance is satisfiable if and only if no literal and its negation belong to the same strongly connected component of its implication graph, which can be used to solve 2-SAT instances in linear time [21].

3.3 3-SAT

The 3-SAT is a special case of SAT that is very useful in proving NP-hardness results. It is so called because it has 3 literals in each of its clauses [22]. The 3-SAT problem is used to find a solution that will satisfy the expression where each of the OR-expressions has exactly 3 Booleans.

$$(x_1 \vee \boldsymbol{b} \vee \mathbf{c}) \text{ and } (a \vee \boldsymbol{b} \vee \boldsymbol{d}) \text{ and } (b \vee \mathbf{c} \vee d)$$

A solution to this logical statement might be (a = T, b = T, c = F, d = F). Thus the best way to solve this problem is to use a guess and check method by trying different combinations until match that work is formed. Recall that a Boolean formula is in conjunction normal form (CNF) if it is a conjunction (AND) of several clauses, each of which is the disjunction (OR) of several literals, each of which is either a variable or its negation. For example,

$$(x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (x_2 \vee \sim x_3 \vee \sim x_4) \wedge (\sim x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee \sim x_2).$$

Theorem (Cook-Levin): SAT is NP-complete

Proof

Cook-Levin theorem states that any problem in NP can be solved in polynomial-time by a non-deterministic Turing machine. Therefore, by definition of NP-complete, it will be

https://dx.doi.org/10.22161/ijaers.73.34 *ISSN: 2349-6495(P) | 2456-1908(O)*

necessary to first show that SAT is in NP and also show that SAT is in NP-Hard. Consider Turing machine RAM t = 0, 1, 2, …, $O(n^2)$ having a read-only memory, program, and read/write memory respectively as shown in figure 1.

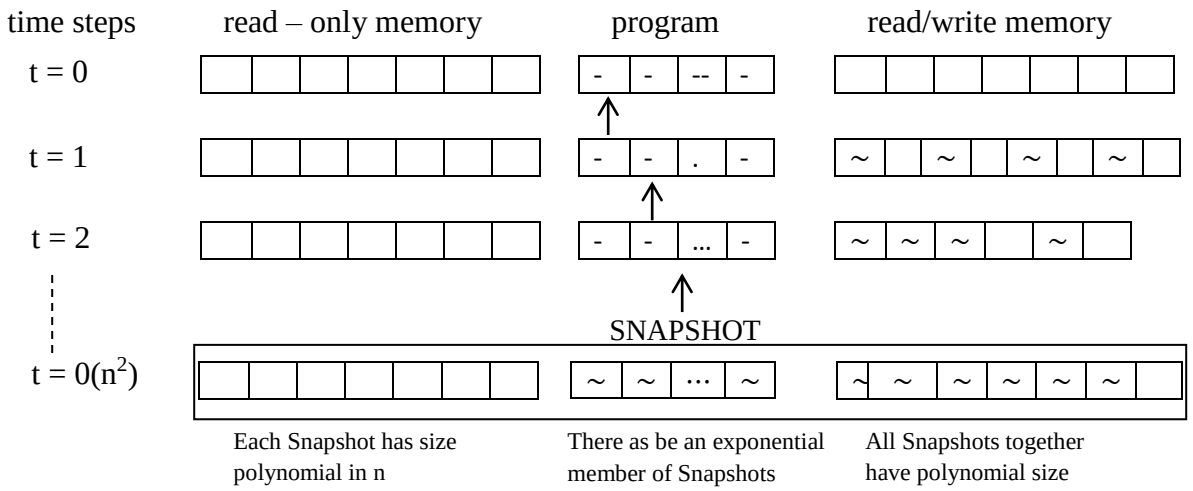

Fig. 2: A Turing machine having read-only memory, program, and read/write memory respectively

[Now consider the Boolean Formula:

$(x_1 \vee x_2\, x_3 \vee x_4 \vee x_5 \vee x_6) \wedge$

$(x_1 \vee (\overline{x2\ x3 \vee x4 \vee x5 \vee x6})) \wedge$

[

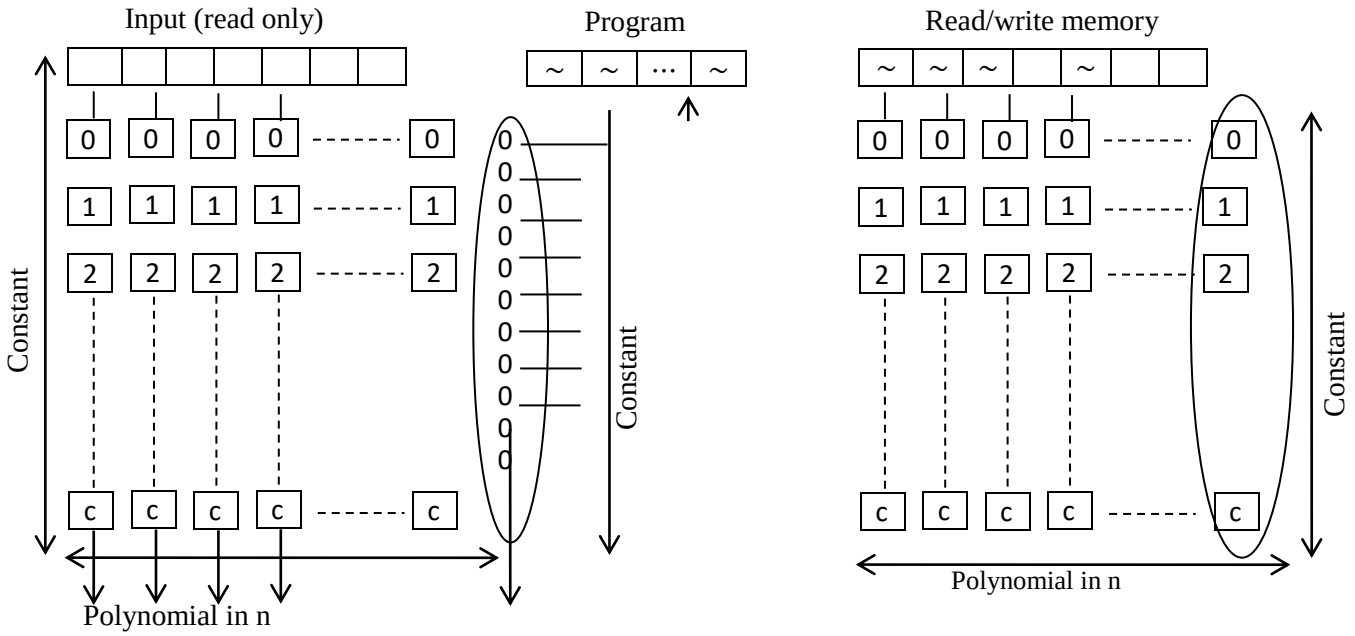

Fig. 3: A Turing machine with read/write memory

https://dx.doi.org/10.22161/ijaers.73.34 *ISSN: 2349-6495(P) | 2456-1908(O)*

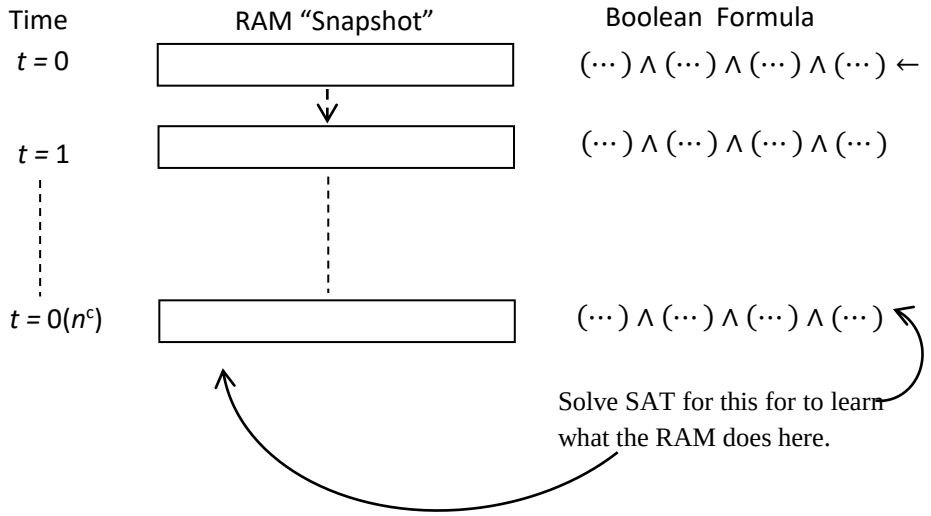

Fig. 4: *A RAM machine for solving SAT problem*

It must be noted that no restrictions are placed on the number of literals in each of a CNF formula. Also, a formula is said to be a CNF formula if it is the case that is a CNF formula with exactly three literals in each clause.

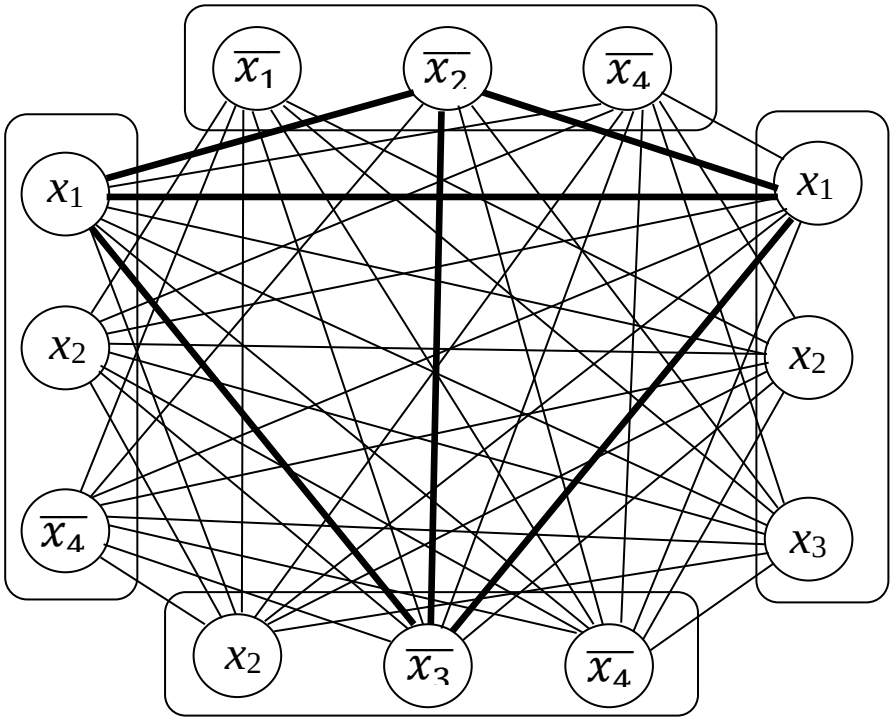

Fig. 5: A snapshot of the Boolean formula

Charge every gate clause into a CNF formula. There are only three types of clauses, one for each type of gate:

$x_1 = x_2 \wedge x_3 \rightarrow (x_1 \vee \sim x_2 \vee \sim x_3) \wedge (\sim x_1 \vee x_2) \wedge (\sim x_1 \vee x_3)$

$x_1 = x_2 \vee x_3 \rightarrow (\sim x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \sim x_2) \wedge (x_1 \vee \sim x_3)$

$x_1 = x_2 \rightarrow (x_1 \vee x_2) \rightarrow (\sim x_1 \vee \sim x_2)$

iv). Ensure that every clause has exactly three literals. Introduce new variables into each one- and two-literals clause, and expand it into two clauses as follows:

https://dx.doi.org/10.22161/ijaers.73.34 *ISSN: 2349-6495(P) | 2456-1908(O)*

$x_1 \rightarrow (x_1 \vee x \vee y) \wedge (x_1 \vee \sim x \vee y) \wedge (x_1 \vee x \vee \sim y) \wedge (x_1 \vee \sim x \vee \sim y)$

$x_1 \vee x_2 \rightarrow (x_1 \vee x_2 \vee x) \wedge (x_1 \vee x_2 \vee \sim x)$.

For example, the formula is not a 3CNF formula, but the following, but the following formula is:

$\varnothing = (x_2 \vee x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee x_2 \vee x_3)$

Therefore, a Boolean formula $\varnothing$ in the variables $x_1, \ldots, x_n$ satisfiable if there exists a Boolean assignment for the variables that cause the formula to evaluate to 1. For example, both formulas $\varnothing$ and C are satisfied: the assignment $x_1 = 1$, $x_2 = 2$, $x_3 = 3$, $x_4 = 0$ works for both of them.

Figure 6 shows a diagram for reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction has a variable for each wire in the circuit

$$\begin{aligned}\emptyset = X_{10}^{1} \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)) \\ \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\ \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\ \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ \wedge (x_6 \leftrightarrow (\sim x_4)) \\ \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\ \wedge (x_4 \leftrightarrow x_6)\end{aligned}$$

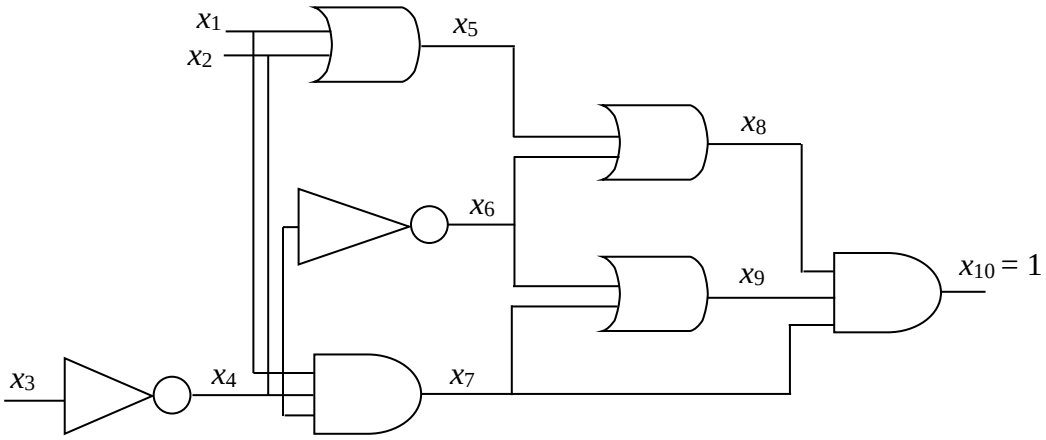

Fig. 6: A typical Circuit-SAT

This shows that it is satisfied and it is an NP-complete problem

$x_{10} = (x_7 \wedge x_8 \wedge x_9) = (x_1 \wedge x_2 \wedge x_4) \wedge (x_5 \vee x_6) \wedge (x_6 \vee x_7)$

$= (x_1 \wedge x_2 \wedge \bar{x}_3) \wedge (\bar{x}_4 \vee (x_1 \vee x_2)) \wedge \bar{x}_4 \vee (x_1 \vee x_2)) \wedge (x_4 \vee (x_4 \wedge x_1 \wedge x_2)$

Using inductive method, we have

x	y	$x \leftrightarrow y$
1	1	1
1	0	0
0	1	0
0	0	1

Therefore, $x_{10} \leftrightarrow (x_8 \wedge x_9 \wedge x_7)$

X_{10} is true if and only if $(x_8 \wedge x_9 \wedge x_7)$

IV. CONCLUSION

In this paper, we revisit Cook-Levin Theorem but using a completely differ approach to prove the theorem. The theorem states that Boolean satisfiability problem is NP-complete. That is to say, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable. Thus if there exists a deterministic polynomial time algorithm for solving a Boolean satisfiability, then there exists a deterministic polynomial time algorithm for solving all problems in NP. Thus Cook-Levin theorem provides a proof that the problem of SAT is NP-complete via reduction technique. The approach used combines the concepts of NP-completeness and circuit-SAT. Using this technique, we showed that Boolean satisfiability problem is NP-complete

https://dx.doi.org/10.22161/ijaers.73.34 *ISSN: 2349-6495(P) | 2456-1908(O)*

through the reduction of polynomial time algorithms for NP-completeness and circuit-SAT.

REFERENCES

[1] Mazza, D, (2016). Church Meet Cook and Levin. In Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'16) July 05 – 08, 2016, New York, NY USA, PP.827 – 836.

[2] Armoni, R., A. Ta – Shona, A. Wigderson and S. Zhou (1997). SLCL 4/3. In Proceedings of the 29th ACM Symposium on Theory of Computing, pp. 230 – 239.

[3] Tian, W., W. Cuco and He, M. (2018). On the Classification of NP. Complete Problems and Their Duality Feature. Int'l Journal of Computer Science & Technology (IJCSIT), Vol 10 No. 1, pp. 64 – 78.

[4] Ogheneovo, E. E. (2019). Theory of Computation: Formal Languages, Automata & Computability: Shack Publishers, 1st Edition, Owerri, Nigeria, pp. 567 – 582.

[5] Immerman, N. (1988). Nondeterministic Space is Closed Under Complementation. SIAM Journal on Computing, Vol. 17, pp. 935 – 938.

[6] Karp, R. M. (1972). Reducibility Among Combinatorial Problems. In Raymond E. Miller, James W. Thatcher (eds.). Complexity of Computer Computations, New York, Planum, pp. 85-103

[7] Levin, L. A. (1986). Average Complete Problems. SIAM Journal on Computing, Vol. 15, No. 1, 285 – 286.

[8] Cook, S, A. (1971). The Complexity of Theorem Proving Procedures. In Proceedings of the 3rd ACM Symposium on Theory of Computing, pp. 151 – 158.

[9] Ciasarch, W. I. (2002). The P = NP? Poll. SCGACT News, Vol. 33, No. 2, pp. 34 – 47.

[10] Fortnow, L. (2009). The Status of the P versus NP Problem. Communications of the ACM, Vol. 52, Issue 1, pp. 78 – 86.

[11] Woeginger, G. J. (2003). Exact Algorithms for NP-hard Problems: A Survey. Combinatorial Optimization Eureka you Shrink!, pp 185 – 207. Springer – Verlag, New York, Inc; New York, NY, USA

[12] Ogheneovo, E. E. (2014). Universal Turing Machine: A Model for all Computational Problems. Int'l Journal of Innovative Research in Computer and Communication Engineering, Vol. 2, Issue 5, pp. 4436 – 4446.

[13] Ladner, R. E. (1975). On the Structure of Polynomial-Time Reducibility. Journal of ACM, Vol. 22, No. 1, pp. 155-171.

[14] Ogheneovo, E. E. (2014). Turing Machine and the Conceptual Problems of Computational Theory. Research Inventy: Int'l Journal of Engineering and Science.

[15] Ogheneovo, E. E. (2016). The Limits of Turing Machine as a Computational Model. Digital Innovations & Contemporary Research in Science & Engineering, Vol. 4 No. 4, pp. 1 – 12.

[16] Dantsin, D., A. Coverall, E. A. Hirsch, R. Kannan, J. Kleinberg, C. H. Papadimitriou, P. Raghavan, and U. Schoning (2002). A Deterministic (2 – 2) (k + 1)^n Algorithm for k – SATT Based on Local Search Theoretical Computer Science, Vol. 289, pp. 69 – 83

[17] Cook, S. (1971). The Complexity of Theorem Proving Procedures. In Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, pp. 151-158.

[18] Levin, L. (1973). Problems of Information Transmissions, Vol. 9, No. 3, pp. 115-116.

[19] Traversa, F. L; C. Ramella, F. Bonani and M.D. Ventra (2015). MemComputing NP -complete Problems in Polynomial Time Using Polynomial Resources and Collective States, Science, Vol. 1, No. 6, pp.

[20] Williams, R, (2011). A Casual Tour Around a Circuit Complexity Bound. SIGACT News,

[21] Karp, R. M. (1972). Reducibility Among Combinatorial Problems. In Miller R. E and J. W. Thatcher, Editors, Complexity of Computer Computations, pp. 85 – 103.

[22] Calabro, C., R. Impaliazzo, and R. Paturi (2009). The Complexity of Satisfiability of Small Depth Circuits. Int'l Workshop on Parameterized and Exact Computation Springer LNCS 5917: 75 – 85.

[23] Calabro, C., R. Impaliazzo, and R. Paturi (2006). A Duality between Clause Width and Clause Deasity for SAT. In IEEE Conference on Computational Complexity, pp. 252 – 260.