Deciding Boolean Satisfiability in Polynomial Time

Deciding All SATs in Polynomial Time

Okoh Ufuoma[1]

[1]**Department of Mathematics and Science, Southern Maritime Academy, Uwheru, Delta, Nigeria**

Abstract

The goal of this work is to modify the famous DPLL algorithm to solve all SATs in polynomial time.

keywords : SAT, satisfiable, P versus NP, algorithm, polynomial time.

1 Introduction

What we commonly call SAT is by many termed Boolean satisfiability problem which imports the question as to whether a CNF Boolean formula is satisfiable [6]. About 1960, Davis, Putname, Longemann, Loveland, and others investigated the SAT problem, and their famous algorithm which solve all SATs is called DPLL algorithm [2],[5]. It is the most common algorithm for deciding SAT in computer science.

The major drawback of the DPLL algorithm is that it runs in exponential time [4]. The goal of this paper is to modify the DPLL algorithm to solve all SATs in polynomial time [3].

The rest of this work is divided into five sections. Section 2 is concerned with the definitions, notations and laws required to understand SAT. Section 3 deals with the DPLL algorithm for solving SAT. In Section 4 we modify the DPLL recursive splitting rule. Section 5 concerns the transformation of the sum of two CNF formulas into a single CNF formula. The 6th section is concerned with a novel polynomial time algorithm for solving SAT.

2 Definitions, Notations and Laws

Boolean Algebra is that branch of Algebra in which the relations of *truth values* are investigated by representing them by symbols or letters which may be either 0 or 1. It is customary in this Algebra to use the phrase *logical values* as synonymous with *truth values* and this meaning will be attached to the phrase throughout the present work.

Any letter used to represent an unspecified logical value is termed a **Boolean variable**. In Boolean algebra, the logical operations of addition $+$, multiplication $\cdot$ and negation $^-$ are performed on the Boolean variables. A Boolean variable or its negation is a **literal**. Any expression built up from Boolean variables, say $A, B, C, \ldots$ or $A_1, A_2, A_3, \ldots$ and the Boolean values 0 and 1 is called a **Boolean expression**. For instance $A + \overline{B}$ is a Boolean expression comprising two variables A and B or two literals A and $\overline{B}$.

The logical assumptions which are taken to be true without proof are called **Boolean axioms**. Theorems used to simplify Boolean expressions are known as **Boolean theorems**. Some special axioms and theorems are stated as follows.

1. **Addition law:**

   A1: $0 + 0 = 0$

   A2: $0 + 1 = 1$

   A3: $1 + 0 = 1$

   A4: $1 + 1 = 1$

2. **Multiplication law:**

   A5: $0 \cdot 0 = 0$

   A6: $0 \cdot 1 = 0$

   A7: $1 \cdot 0 = 0$

   A8: $1 \cdot 1 = 1$

3. **Annulment Law:**

T1: $1+A=1$

T2: $A\cdot 0=0$

4. **Identity Law:**

   T3: $0+A=A$

   T4: $A\cdot 1=A$

5. **Idempotent Law:**

   T5: $A+A=A$

   T6: $A\cdot A=A$

6. **Double Negation Law:**

   T7: $\overline{\overline{A}}=A$

7. **Complement Law:**

   T8: $\overline{A}+A=1$

   T9: $A\cdot\overline{A}=0$

As in ordinary algebra, the following laws hold in Boolean algebra: commutative and associative laws for addition and multiplication, distributive laws both for multiplication over addition and for addition over multiplication.

The logical sum of literals on distinct variables is called a **clause**. A clause with only one literal is referred to as a **unit clause**. The literals of a clause can be written in increasing order as in

$$A_1+A_4+A_7$$

or in alphabetical order as in

$$B+C+\overline{F}.$$

A **Boolean formula** is a logical expression defined over Boolean variables. A **Boolean assignment** to a set of Boolean variables is the set of logica values assigned to the variables in other to evaluate a Boolean formula. A **satisfying Boolean assignment** for a Boolean formula is an assignment such that the Boolean formula evaluates to logic 1. If the Boolean variables associated with a Boolean formula can be assigned logical values such that the formula turns out to be logic 1, then we say that the formula is *satisfiable*. If it is not possible to assign such values, then we say that the formula is *unsatisfiable*.

We will be interested in Boolean formulas in a certain special form, the conjunctive normal form; it is the generally accepted norm for SAT solvers because of its simplicity and usefulness. A **conjunctive normal form** CNF is a multiplication of clauses. A K-CNF is a CNF in which every clause contains at most K literals. If the negation of a literal does not appear in a CNF formula, we refer to it as a **pure literal**. The formula

$$d_1=(A+\overline{B})(\overline{A}+B+C)(A+B+C)$$

is a 3-CNF Boolean formula with three variables

$$A,B,C,$$

five literals

$$A,\overline{A},B,\overline{B},C,$$

and three clauses

$$(A+\overline{B}),(\overline{A}+B+C),(A+B+C).$$

The negation of the literal C does not appear in the formula and so C is a pure literal.

3 Deciding SAT by DPLL Algorithm

The DPLL algorithm is the most popular complete satisfiability (SAT) solver. While its worst case complexity is exponential, three rules are applied to speed-up the decision process [1].

1. Unit Propagation Rule. This rule states that *one can set the value of the only unassigned literal of a unit clause in such a way that the clause is satisfied*.
2. Pure Literal Rule. The pure literal rule states that *if an unassigned literal appears while its negation does not, we can set the value of the literal to 1*.
3. Splitting Rule. This states that *one should choose an assignment of* 1 *or* 0 *for a Boolean variable in a formula, simplify the formula based on that choice, then recursively check the satisfiability of the simplified formula*. If the simplified formula is satisfiable, the original formula is satisfiable, otherwise, the same recursive check is done assuming the opposite logical value.

We furnish an instance of the way in which DPLL algorithm is used to solve SAT.

Example 1. *Use the DPLL approach to decide the satisfiability of*

$$\phi = (\overline{A}+B+C)(\overline{A}+\overline{B}+C)(A+B+\overline{C})(A+B+C).$$

Unit propagation is not possible as there are no unit clauses. Pure literal rule is not applicable as there is no literals that occur only positively or only negatively. We apply the splitting rule by selecting some literal, say A. We put $A=0$ and propagate. This results in

$$\phi_{A=0} = (1+B+C)(0+\overline{B}+C)(0+B+\overline{C})(0+B+C)$$

which becomes

$$\phi_{A=0} = (\overline{B}+C)(B+\overline{C})(B+C).$$

Unit propagation and pure literal are still not applicable. Apply splitting rule for the next literal B. Set $B=0$ and propagate:

$$\phi_{A=0,B=0} = (1+C)(0+\overline{C})(0+C).$$

which becomes

$$\phi_{A=0,B=0} = (\overline{C})(C).$$

This formula consists of two unit clauses and so it is possible to apply unit propagation, which results in

$$\phi_{A=0,B=0} = 0.$$

Since $\phi_{A=0,B=0}=0$, we backtrack, set $B=1$ and propagate:

$$\phi_{A=0,B=1} = (0+C)(1+\overline{C})(1+C)$$

which results in

$$\phi_{A=0,B=1} = (C).$$

We apply unit propagation or the pure literal rule and conclude that this formula and hence the original formula is satisfiable.

4 Modification of the Splitting Rule of DPLL

Based on the DPLL splitting rule already mentioned, we choose a literal say A_k from the initial CNF formula d_k consisting of the variables $A_k, A_{k+1}, \ldots, A_n$ and assign the logic value 1 to it. The resulting CNF formula is denoted $[d_k]_{A_k=1}$. Notice the use of the square brackets around d_k. In fact, the notation

$$[d_k]_{A_k=1}$$

indicates that the logic value 1 is to be substituted for the variable A_k in the CNF formula d_k. We check if $[d_k]_{A_k=1}$ is satisfiable; if this is the case, the initial CNF formula d_k is satisfiable; otherwise, we do the same check, assuming the opposite logical value 0. Thus, we see that by the DPLL spliting rule, the initial CNF formula d_k is split into two simpler CNF formulas, $[d_k]_{A_k=1}$ and $[d_k]_{A_k=0}$. Hence, the original or initial formula d_k is satisfiable if either $[d_k]_{A_k=1}$ or $[d_k]_{A_k=1}$ or both are satisfiable.

In set theory, the set that consists of all elements belonging to either set A or set B or both is called the union of A and B, denoted as $A+B$. Thus the statement *either* $[d_k]_{A_k=1}$ *or* $[d_k]_{A_k=1}$ *or both are satisfiable* implies the new Booleam formula

$$d_{k+1} = [d_k]_{A_k=1} + [d_k]_{A_k=0} \tag{1}$$

is satisfiable. It follows that if d_{k+1} is satisfiable, then d_k is satisfiable, and if d_{k+1} is unsatisfiable, then d_k is unsatisfiable. Thus the problem of satisfying d_k is equivalent to the problem of satisfying d_{k+1}. Since the number of variables of d_{k+1} is smaller than that of the variables of d_k by one, deciding the satisfiability of d_{k+1} will be easier than deciding the satisfiability of d_k.

In the following insatance we show how to derived the new formula d_2 from the original CNF formula d_1.

Example 2. *Given the CNF formula*

$$d_1 = (A_1 + \overline{A_2})(A_1 + \overline{A_3})(\overline{A_2} + A_3)$$

find d_{k+1}.

Letting $A_1 = 1$ gives

$$[d_1]_{A_1=1} = (1 + \overline{A_2})(1 + \overline{A_3})(\overline{A_2} + A_3)$$

which becomes

$$[d_1]_{A_1=1} = (\overline{A_2} + A_3).$$

Setting $A_1 = 0$ gives

$$[d_1]_{A_1=0} = (0 + \overline{A_2})(0 + \overline{A_3})(\overline{A_2} + A_3)$$

which becomes

$$[d_1]_{A_1=0} = (\overline{A_2})(\overline{A_3})(\overline{A_2} + A_3).$$

Hence, we get

$$\begin{aligned} d_2 &= [d_1]_{A_1=1} + [d_1]_{A_1=0} \\ &= (\overline{A_2} + A_3) + \overline{A_2}\,\overline{A_3}(\overline{A_2} + A_3). \end{aligned}$$

The recursive formula d_{k+1} is the sum of two CNF formulas. If it can be transformed to a CNF formula, we will be able to recursively reduce the number of variables of d_k easily and continuously until the satisfiability of d_k becomes decidable, employing the unit propagation and pure literal rules. The next section will be devoted to a method of transforming the sum of two CNF formulas into a single CNF formula.

5 Transforming Sum of CNFs to a Single CNF

I shall here invent a technique for transforming the sum of two CNF formulas to a Single CNF formula.

Theorem 1. *Let* $f_1, f_2, \ldots, f_p$ *and* $g_1, g_2, \ldots, g_n$ *be Boolean formulas. Then*

$$f_1 f_2 \cdots f_p + g_1 g_2 \cdots g_q = (f_1 + g_1)(f_1 + g_2)\cdots(f_1 + g_q)(f_2 + g_1)(f_2 + g_2)\cdots(f_2 + g_q)\cdots(f_p + g_1)(f_m + g_2)\cdots(f_p + g_q).$$

For what purpose were all mathematical theorems before they can be employed in mathematics but to convince, in terms not to be misunderstood, the readers of their soundness. A mathematical proposition then would be vain without the demonstration of its validity. Hence, I shall prove this novel theorem to convince the reader of its truth.

Proof.

$$\begin{aligned} f_1 f_2 \cdots f_p + g_1 g_2 \cdots g_q &= \overline{\overline{f_1 f_2 \cdots f_p + g_1 g_2 \cdots g_q}} \\ &= \overline{\overline{f_1 f_2 \cdots f_p}\,\overline{g_1 g_2 \cdots g_q}} \\ &= \overline{(\overline{f_1} + \overline{f_2} + \cdots + \overline{f_p})(\overline{g_1} + \overline{g_2} + \cdots + \overline{g_q})} \\ &= \overline{\overline{f_1}(\overline{g_1} + \overline{g_2} + \cdots + \overline{g_q}) + \overline{f_2}(\overline{g_1} + \overline{g_2} + \cdots + \overline{g_q}) + \cdots \overline{f_p}(\overline{g_1} + \overline{g_2} + \cdots + \overline{g_q})} \\ &= \overline{\overline{f_1}\,\overline{g_1} + \overline{f_1}\,\overline{g_2} + \cdots + \overline{f_1}\,\overline{g_q} + \overline{f_2}\,\overline{g_1} + \overline{f_2}\,\overline{g_2} + \cdots + \overline{f_2}\,\overline{g_q} + \cdots + \overline{f_p}\,\overline{g_1} + \overline{f_p}\,\overline{g_2} + \cdots + \overline{f_p}\,\overline{g_q}} \\ &= (f_1 + g_1)(f_1 + g_2)\cdots(f_1 + g_q)(f_2 + g_1)(f_2 + g_2)\cdots(f_2 + g_q)\cdots(f_p + g_1)(f_p + g_2)\cdots(f_p + g_q). \end{aligned}$$

□

With this transformation theorem, the Boolean formula d_{k+1}, the sum of the two CNF formulas, $[d_k]_{A_k=1}$ and $[d_k]_{A_k=0}$, can be transformed into a single CNF formula. We proffer an instance to show how it may be applied.

Example 3. *Transform the sum of CNF formulas*

$$d_2 = (A_2 + \overline{A_3})(\overline{A_3} + A_4) + A_2(A_3 + \overline{A_4})(\overline{A_2} + \overline{A_3} + A_4)$$

into a single CNF formula.

By the Transformation Theorem 1, we have

$$\begin{aligned} d_2 &= (A_2 + \overline{A_3} + A_2)(A_2 + \overline{A_3} + A_3 + \overline{A_4})(A_2 + \overline{A_3} + \overline{A_2} + \overline{A_3} + A_4) + (\overline{A_3} + A_4 + A_2)(\overline{A_3} + A_4 + A_3 + \overline{A_4})(\overline{A_3} + A_4 + \overline{A_2} + \overline{A_3} + A_4) \\ &= (A_2 + \overline{A_3})(A_2 + \overline{A_3} + A_4)(\overline{A_2} + \overline{A_3} + A_4) \\ &= (A_2 + \overline{A_3})(\overline{A_2} + \overline{A_3} + A_4). \end{aligned}$$

6 Polynomial-time Algorithm for Deciding SAT

We will now look at a new polynomial-time algorithm for solving SAT. This algorithm involves the continuous reduction of the original or initial CNF Boolean formula into a smaller and smaller CNF Boolean formulas until logic 1 or 0 emerges.
Given the CNF Boolean formula

$$d_1 = F(A_1, A_2, \ldots, A_n)$$

take the following steps to decide the satisfiability of d_1.

1. Set $k = 1$.
2. Set $d_{k+1} = [d_k]_{A_k=1} + [d_k]_{A_k=0}$.
3. Express d_{k+1} in CNF using Theorem 1.
4. If $d_{k+1} = 0$, print "d_1 is unsatisfiable"and stop.
5. If $d_{k+1} = 1$, print "d_1 is satisfiable ". Otherwise, go to step 6.
6. Set $k = k + 1$ and return to step 1.

6.1 Time Complexity of the Algorithm for Deciding SAT

First, we notice that two operations of substitution are required for generating the formula d_{k+1}, namely $[d_k]_{A_k=1}$ and $[d_k]_{A_k=0}$.

Let

$$[d_k]_{A_k=1} = f_1 f_2 \cdots f_p$$

and

$$[d_k]_{A_k=1} = g_1 g_2 \cdots g_q$$

where $f_1, f_2, \cdots, f_p$ are the p clauses of the CNF formula obtained by setting $A_k = 1$ in the CNF formula d_k and $g_1, g_2, \cdots, g_q$ are the q clauses of the CNF formula obtained by setting $A_k = 0$ in the CNF formula d_k. Then

$$d_{k+1} = f_1 f_2 \cdots f_p + g_1 g_2 \cdots g_q.$$

To transform d_{k+1} to a single CNF formula using Theorem 1, the maximum number of possible combinations $f_i + g_j$ for $i = 1$ to p and $j = 1$ to q is pq. Thus, there will be at most $pq + 2$ operations for any given variable. Since there are m variables, we say that there are at most $m(pq + 2)$ operations. Consequently, the algorithm proposed in this work requires at most $O(n^3)$ operations. Thus the algorithm is said to have polynomial time complexity.

6.2 Instances of SAT

In what follows we proffer instances of the way in which the new procedure proposed can be employed.

Example 4. *Decide the satisfiability of the Boolean formula*

$$d_1 = (A_1 + \overline{A_2})(A_1 + \overline{A_3})(\overline{A_2} + A_3).$$

We begin with the recurring formula

$$d_{k+1} = [d_k]_{A_k=1} + [d_k]_{A_k=0}.$$

Putting $k = 1$, we have

$$\begin{aligned} d_2 &= [d_1]_{A_1=1} + [d_1]_{A_1=0} \\ &= (\overline{A_2} + A_3) + (\overline{A_2})(\overline{A_3})(\overline{A_2} + A_3) \\ &= (\overline{A_2} + A_3 + \overline{A_2})(\overline{A_2} + A_3 + \overline{A_3})(\overline{A_2} + A_3 + \overline{A_2} + A_3) \\ &= (\overline{A_2} + A_3). \end{aligned}$$

Next, putting $k = 2$, we get

$$\begin{aligned} d_3 &= [d_2]_{A_2=1} + [d_2]_{A_2=0} \\ &= (A_3) + (1) \\ &= 1. \end{aligned}$$

The fact that $d_3 = 1$ suggests that d_2 and hence d_1 are satisfiable.

Example 5. *Decide the satisfiability of the Boolean formula*

$$d_1 = (\overline{A_1} + \overline{A_3})(A_1 + A_2 + A_3)(A_1 + \overline{A_2} + A_3)(A_1 + \overline{A_3})(\overline{A_1} + A_3).$$

We begin with the recurring formula

$$d_{k+1} = [d_k]_{A_k=1} + [d_k]_{A_k=0}.$$

Setting $k = 1$ gives

$$\begin{aligned} d_2 &= [d_1]_{A_1=1} + [d_1]_{A_1=0} \\ &= (\overline{A_3})(A_3) + (A_2 + A_3)(\overline{A_2} + A_3)(\overline{A_3}) \\ &= (\overline{A_3})(A_2 + A_3)(\overline{A_2} + A_3). \end{aligned}$$

Next, letting $k = 2$, we obtain

$$\begin{aligned} d_3 &= [d_2]_{A_2=1} + [d_2]_{A_2=0} \\ &= (\overline{A_3})(A_3) + (\overline{A_3})(A_3) \\ &= 0. \end{aligned}$$

The logical value of d_3, as we have seen, is 0. This means that the CNF Boolean formula d_3 is unsatisfiable. The implication of this is that the original CNF Boolean formula d_1 is unsatisfiable.

Example 6. *Decide the satisfiability of the Boolean formula*

$$d_1 = (\overline{A_1} + A_2 + \overline{A_3})(A_1 + \overline{A_2} + A_3)(A_2 + A_4)(\overline{A_2} + A_4 + A_5)(\overline{A_3} + A_4 + \overline{A_6})(\overline{A_4} + \overline{A_5} + \overline{A_7})(\overline{A_4} + A_8).$$

We begin with the recurring formula

$$d_{k+1} = [d_k]_{A_k=1} + [d_k]_{A_k=0}.$$

Letting $k = 1$ gives

$$\begin{aligned} d_2 &= [d_1]_{A_1=1} + [d_1]_{A_1=0} \\ &= [(A_2 + \overline{A_3}) + (\overline{A_2} + A_3)](A_2 + A_4)(\overline{A_2} + A_4 + A_5)(\overline{A_3} + A_4 + \overline{A_6})(\overline{A_4} + \overline{A_5} + \overline{A_7})(\overline{A_4} + A_8) \\ &= (A_2 + A_4)(\overline{A_2} + A_4 + A_5)(\overline{A_3} + A_4 + \overline{A_6})(\overline{A_4} + \overline{A_5} + \overline{A_7})(\overline{A_4} + A_8). \end{aligned}$$

Letting $k = 2$, we have

$$\begin{aligned} d_3 &= [d_2]_{A_2=1} + [d_2]_{A_2=0} \\ &= [(A_4 + A_5) + (A_4)](\overline{A_3} + A_4 + \overline{A_6})(\overline{A_4} + \overline{A_5} + \overline{A_7})(\overline{A_4} + A_8) \\ &= (A_4 + A_5)(\overline{A_3} + A_4 + \overline{A_6})(\overline{A_4} + \overline{A_5} + \overline{A_7})(\overline{A_4} + A_8). \end{aligned}$$

Putting $k = 3$, we have

$$\begin{aligned} d_4 &= [d_3]_{A_3=1} + [d_3]_{A_3=0} \\ &= [(A_4 + \overline{A_6}) + (1)](A_4 + A_5)(\overline{A_4} + \overline{A_5} + \overline{A_7})(\overline{A_4} + A_8) \\ &= (A_4 + A_5)(\overline{A_4} + \overline{A_5} + \overline{A_7})(\overline{A_4} + A_8). \end{aligned}$$

Setting $k = 4$, we have

$$\begin{aligned} d_5 &= [d_4]_{A_4=1} + [d_4]_{A_4=0} \\ &= (\overline{A_5} + \overline{A_7})(A_8) + (A_5)(1) \\ &= (A_5 + A_8). \end{aligned}$$

Letting $k = 5$, we have

$$\begin{aligned} d_6 &= [d_5]_{A_5=1} + [d_5]_{A_5=0} \\ &= (1) + (A_8) \\ &= 1. \end{aligned}$$

Since the logical value of d_6 is 1, we infer that the original CNF Boolean formula d_1 is satisfiable.

Instances of this algorithm might be multiplied to any extent. But we must stop here for our limit reminds us that we must be brief.

References

[1] Aaronson S. and Wigderson A., Algebrization: a new barrier in complexity theory, in STOC, 2008, pp. 731–740

[2] Aho, Alfred V.; Hopcroft, John E.; Ullman, Jeffrey D. (1974). The Design and Analysis of Computer Algorithms. Addison-Wesley. Theorem 10.4.

[3] Baker T. P., Gill J, and Solovay R., Relativizations of the P=?NP question, SIAM Journal on Computing 4:4,431-442, 1975

[4] Cook S. A., The complexity of theorem proving procedures, Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 151-158, 1971.

[5] Davis M., Logemann G. , and Loveland D., Communications of the ACM 5, 394–397 (1962).

[6] Mironov, Ilya; Zhang, Lintao (2006). Biere, Armin; Gomes, Carla P. (eds.). "Applications of SAT Solvers to Cryptanalysis of Hash Functions". Theory and Applications of Satisfiability Testing — SAT 2006. Lecture Notes in Computer Science. Springer. *Mathematics Handbook for Science and Engineering*, Springer, New York, 2006, 5th ed.