A NATURAL FEASIBLE ALGORITHM THAT CHECKS SATISFIABILITY OF 2-CNF FORMULAS AND, IF THE FORMULAS IS SATISFIABLE, FINDS A SATISFYING VECTOR

Kosheleva, Olga, PhD, Associate Professor
Kreinovich, Vladik, PhD, Professor
University of Texas at El Paso
olgak@utep.edu, vladik@utep.edu

Abstract: One of the main results in Theory of Computation courses is the proof that propositional satisfiability is NP-complete. This means that, unless P = NP (which most computer scientists believe to be impossible), no feasible algorithm is possible for solving propositional satisfiability problems. This result is usually proved on the example of 3-CNF formulas, i.e., formulas of the type C_1 & … & C_m, where each *clause* C_i has the form a ∨ b or a ∨ b ∨ c, with no more than three *literals* – i.e., propositional variables v_i or their negations $\sim v_i$. Textbooks usually mention that for 2-CNF formulas – in which every clause has at most 2 literals – the corresponding problem can be solved by a feasible algorithm. From the pedagogical viewpoint, however, the problem with known feasible algorithms for 2-CNF formulas is that they are based on clever tricks. In this paper, we describe a natural feasible algorithm for solving 2-CNF, an algorithm whose ideas are similar to Gauss elimination in linear algebra.

Keywords: propositional satisfiability, 2-CNF, Gauss elimination, feasible algorithm.

Feasible vs. non-feasible algorithms. It is known that some algorithms are practically feasible, while others are not. For example, an algorithm that requires 2^n computational steps on inputs of length n requires, for reasonable size inputs of size n = 300, more computation time than the lifetime of the Universe. It is therefore desirable to formally describe which algorithms are feasible and which are not.

Coming up with a perfect definition of feasible algorithms is still an open problem. The best definition we have now defines an algorithm to be feasible if its computation time is bounded by a polynomial of the length n of the input. In most cases, this definition is adequate, but not always: e.g.,

- according to this definition, an algorithm that requires $10^{100}n$ steps is feasible, while
- from the practical viewpoint, it is clearly not.

In spite of these limitations, this is the best definition we have, and thus, this definition is used in theoretical computer science as a formal definition of a feasible algorithm.

NP and NP-complete. In most practical problems, once we have a candidate for a solution, we can feasibly check whether this candidate is indeed a solution to our problem. For example:

- In mathematics, we are given a statement x, and we want to find a proof y of either this statement x or of its negation ~x. Coming up with a proof is often very difficult, but once a detailed proof is given, we can check, step-by-step, whether this proof is correct: proof-checking programs are available since the 1960s.
- In physics, we are given a set of observations x, and we need to find a physical law y that explains all these observations (think Ohm's law as an example). Coming up with such a law is difficult, but once this law is described, checking whether each observation is consistent with this law is feasible.
- In engineering, we are given specification x. For example, when we design a bridge, we can specify how much weight it should withstand, how much it should cost, how much wind it should tolerate, etc. The problem is to find a design y that satisfies all these specifications. Coming up with such a design is often difficult, but, once the design is described, we can

use known software packages to check whether this design indeed satisfies all the specifications.

Problems of this type – when there is a feasible algorithm for checking whether a candidate for a solution is indeed a solution – are known as problems from the class NP.

Some problems from this class can be solved in feasible (polynomial) time; the class of all such problems is denoted by P. A natural question is: can every problem from the class NP be solved by a feasible algorithm, i.e., is P = NP? This is an open problem; most computer scientists believe that P is different from NP, but no one has proved it yet. What is known is that there are some problems which are the hardest of all problems from the class NP, in the sense that every problem from the class NP can be reduced to this problem. Such problems are known as *NP-complete*.

Propositional satisfiability and 3-CNF formulas. Historically the first problem for which NP-completeness was proven was the following *propositional satisfiability* problem. A propositional formula is any formula that can be obtained from propositional (Boolean, true-false) variables v_1, …, v_n, by using propositional connectives & (and), $\vee$ (or), and ~ (not). The problem is:

- given a propositional formula,
- check whether there exists values of the variables v_i that make it true, and
- if yes, find these values.

NP-completeness is usually proven for 3-CNF formulas, i.e., formulas of the type C_1 & … & C_m, where each *clause* C_i has the form $a \vee b$ or $a \vee b \vee c$, with no more than three *literals* – i.e., propositional variables v_i or their negations $\sim v_i$.

2-CNF formulas. Textbooks usually mention that for 2-CNF formulas, i.e., formulas in which each clause has two literals $a \vee b$, there exists a feasible algorithm.

The corresponding algorithms, however, rarely appear in the textbooks, since known feasible algorithms are based on clever tricks and do not naturally follow from the problem.

What we do in this paper. In this paper, we describe a natural feasible algorithm for solving 2-CNF formulas. This algorithm is based on the idea of variable elimination – similar to the well-known Gauss elimination in linear algebra.

Main idea behind the proposed algorithm. The main idea behind the proposed algorithm is that a disjunction $a \vee b$ is equivalent to $\sim b \rightarrow a$. Indeed, if either a is true or b is true, and b is false, this means that a should be true.

Computers usually describe true as 1, and false as 0. In this case, as one can easily see, $a \rightarrow b$ is equivalent to $a <= b$.

Resulting algorithm. Let us use the above idea to come up with an algorithm. Let us select one of the propositional variables, e.g., v_1, and let us show how we can eliminate the variable v_1 from a 2-CNF formula F.

The formula F may contain clauses of the type $v_1 \vee b$ for some b, or of the type $\sim v_1 \vee c$ for some c.

- A clause of the first type is equivalent to $\sim b <= v_1$.
- A clause of the second type is equivalent to $v_1 <= c$.

By considering all such clauses, we get some lower bounds for v_1 and some upper bounds.

- If such a v_1 exists, then each lower bound must be less than or equal to each upper bound.
- Vice versa, if each lower bound is smaller than or equal to each upper bound, then as v_1, we can take, e.g., the largest of the lower bounds.

Each inequality of the type $\sim b <= c$ between a lower bound and an upper bound can be described in the equivalent form of a clause $b \vee c$. By combing the resulting clauses with original clauses that did not contain v_1, we get a new formula which:

- Has one fewer variable and

- whose satisfiability is equivalent to the satisfiability of the original formula.

From this formula, we can now eliminate the next variable, etc. In n steps, we reduce it to the formula with no variable s at all – and thus, we can decide whether the formula is satisfiable or not. If the formula is satisfiable, then, going back, we can find the values of the variables one by one.

This algorithm is feasible. For n variables, we have 2n possible literals, and thus, no more than $(2n)^2$ possible clauses. So, at each of n stages of the algorithm, we have a formula with a feasible (quadratic) number of clauses – and therefore, the overall time is at most cubic (and hence, feasible).

Example. Let take a formula (a ∨ b) & (~a ∨ ~c) & (b ∨ c) & (~b ∨ ~c).

Let us first eliminate the variable a. The clause a ∨ b is equivalent to ~b <= a, and the clause ~a ∨ ~c is equivalent to a <= ~c. So, in this case, we have:

- one lower bound for a, and
- one upper bound for a.

The requirement that every lower bound should be smaller than equal to every upper bound has the form ~b <= ~c. In terms of a clause, this means b ∨ ~c.

Combining this clause with the original clauses that do not contain a, we get a new formula with one fewer variable: (b ∨ ~c) & (b ∨ c) & (~b ∨ ~c). Let us now eliminate the variable b from this formula. The three clauses containing b take the form c <= b, ~c <= b, and b <= ~c. Here, we have:

- two lower bounds for b and
- one upper bound.

Thus, the requirement that every lower bound be smaller than or equal to every upper bound takes the form c <= ~c and ~c <= ~c.

- The inequality ~c <= ~c is, of course, always true.
- The condition c <= ~c is not satisfied when c = 1, but it is satisfied when c = 0.

Thus, we conclude that c = 0.

These inequalities are satisfiable – and thus, the original formula is satisfaible. Let us now go back one variable at a time and find the values of the variables that make the original formula true.

- Let us first find b. For c = 0 , the inequalities c <= b, ~c <= b, and b <= ~c that contain b take the form 0 <= b, 1 <= b, and b <= 1. Thus, 1 <= b <= 1, so b = 1.
- Let us now find a. Substituting c = 0 and b = 1 into the inequalities ~b <= a and a <= ~c containing a, we conclude that 0 <= a <= 1. Thus, in principle, we can have both values a = 0 and a = 1.

One can easily check that for each of these two values a, we have a satisfying vector – i.e., a set of values that make the original formula true:

- a = 0, b = 1, c = 0 is a satisfying vector, and
- a = 1, b = 1, and c = 0 is also a satisfying vector.

Acknowledgments. This work was supported by the National Science Foundation grant HRD-1242122 (Cyber-ShARE Center of Excellence).

The authors are thankful to Dr. Mourat Tchoshanov for his encouragement.

References.

1. M. Sipser, *Introduction to the Theory of Computation*, Thomson Course Technology, Boston, Massachusetts, 2012.