Polynomial Exact-3-SAT-Solving Algorithm

Matthias Michael Mueller
louis@louis-coder.com

Sun, 2015-05-10
Version C-1.0

Abstract

This paper explains an algorithm that is capable of solving any instance of a 3-SAT CNF in maximal $O(n^{12})$, whereby n is the literal range within the formula to solve. The algorithm has been tested at great length in form of a C++ implementation whose download link is given. Under the supposition the algorithm is correct, the P-NP-Problem would be solved with the result that the complexity classes NP and P are equal.

1 Introduction

The P-NP-Problem is one of the most important questions, concerning both mathematicians as well as computer scientists. Easily spoken, the problem is to find out and prove if it is possible to solve, within a reasonable amount of time and space, any of the decision problems which are categorized as belonging to the complexity class 'NP'.

'Reasonable' shall mean that the number of calculating steps required to solve any of those NP problems can be described by a polynomial with the input size of the problem 'n' as base and some constant exponent 'c'. The simplest example would be n^c. Until now, for problems out of the class NP there are only algorithms known whose complexity grows by a constant factor each time you increase the input size by one, so the constant is in the base and the input size is in the exponent: c^n. This has the effect that only small-sized NP-problem instances can be handled by existing computers unless some polynomial processing method is (ever) found.

The author of this paper is very sure the algorithm described in this document solves the 'NP' problem '3-SAT' in polynomial time and space, and thus would give the answer to the main question that NP problems can be solved much easier than with an exponential amount of work.

This would imply that "P = NP", what means that all algorithms which require exponentially much time and therefore belong to the complexity class 'NP', could be solved also in polynomial time. That classified all the algorithms as laying also completely in the complexity class 'P' and therefore the two complexity classes are equal. The effect would be that many planning tasks, for instance in economy, could now practically be done at all or much more precisely, what had an enormous positive effect on people's life in general.

2 History of Polynomial Algorithm

Please notice that there are several different algorithms on the Internet that have been invented by me during the past ≈3 years. What you are currently reading is my *third* attempt to build a polynomial 3-SAT solver. I have developed three 3-SAT solving algorithms, which I call *Algorithm A* (published December 2012), *Algorithm B* (published December 2013) and *Algorithm C* (put on-line May 2015).

While Algorithm A turned out to be faulty, number B could not be falsified, neither by me, nor by anyone else, as far as I know. With 'not falsified' I mean Algorithm B is of polynomial complexity for sure, as detectable by the way it is programmed, i.e. the implementation consists of nested loops only and uses no power-of-function(s) or recursion, and it never returned wrong results since over 50 million test runs. The only problem about Algorithm B is that its correctness could not (yet) be mathematically proven. All my attempts to do so were not sufficient to wipe out any doubt. For this reason I created Algorithm C, which works similar to Algorithm B, but is easier to prove, as it uses simpler mechanisms.

You can view all three algorithms in "pdf" format by visiting one URL out of:

http://www.vixra.org/author/matthias_mueller

http://www.louis-coder.com/Polynomial_3-SAT_Solver/Polynomial_3-SAT_Solver.htm

At the second URL (louis-coder.com) you can also download the C++ implementations of Algorithms B and C, which do in each case consist of pretty commented source code and one compiled Windows binary. Please notice the upcoming topic 'Implementation on the Internet'. My recommendation is to first view the code/executable of Algorithm C.

Please prefer downloading material related to Algorithm C (pdf, source code) **from my website www.louis-coder.com** (see above) instead of from third-party sites, because from my site you easily can get the newest version, which might have eventual errors corrected and might have been extended since the initial release.

3 3-SAT: The Problem to Solve

The task the algorithm explained in this paper shall do is to decide if any given instance of a 3-SAT CNF has a solution or not. '3-SAT CNF' is used here as an abbreviation that denotes an instance of a formula in conjunctive normal form, whereby each conjunction (AND-ed term) consists of a disjunction (OR-term) of exactly 3 variables, called 'literals':

$$I = \bigwedge_{i=0}^{a-1} (\epsilon_{i1} x_{i1} \vee \epsilon_{i2} x_{i2} \vee \epsilon_{i3} x_{i3})$$

For all $i = 0, ..., a-1$ the variables x_{i1}, x_{i2}, x_{i3} are pairwise distinct. If ϵ_{ij} is placed before x_{ij}, it shall mean that x_{ij} is negated. a is used in this document as the number of AND-ed terms. The literal range, i.e. the greatest index n of some literal x_n shall later be used as measurement of the input size.

Example: $I = (x_1 \vee x_3 \vee \neg x_5) \wedge (x_2 \vee x_4 \vee x_5)$

We identify $a = 2$, $n = 5$ as there are two AND-ed terms and the highest literal index is 5 (the lowest one possible is always 1, for any instance used throughout this document). One solution, also called model in literature, for the sample instance could be $x_1 = true$, $x_2 = true$, or alternatively $x_5 = false$, $x_4 = true$, for instance.

It can happen that there is no solution at all that could make the whole CNF true, what is then to be found out by the polynomial solver.

One way to decide if the 3-SAT instance is solvable would be to try out all possible solutions. The problem about this proceeding is that, if there is no solution at all, you had to test as many as 2^n solutions to finally find out that you cannot solve the CNF. I strongly suppose the algorithm described in this document can determine the

3-SAT CNF's solvability, even in the worst-case, after maximal n^{12} loop runs, whereby the practical complexity is for most CNFs considerably smaller.

As a last word I want to thank Mr. Mihai Prunescu, whose paper [8] gave me inspiration how to denote parts of this topic.

4 Main Algorithm

4.1 General Definitions

Definition 4.1.1 *The expression* **CNF** *does consistently stand, within this paper, for the 3-SAT CNF that is to be solved by the polynomial solver, as explained in the previous topic. As already mentioned, n is the literal range of the CNF, i.e. the number of different literal indices. n will later be seen as the polynomial solver's input size that defines the algorithm's complexity.*

Definition 4.1.2 *The* **Clause Line Notation** *of a clause shall be a word consisting of n chars, whereby each char can be chosen out of {0, 1, -}. For 3-SAT, any Clause Line contains exactly 3 "0" or "1" chars and exactly $n-3$ "-" chars.*

A classical, mathematical clause known from literature can be converted into a Clause Line as follows: If the literal of the mathematical clause is negated, place "0" at the location within the Clause Line denoted by the literal index. If the literal of the mathematical clause is not negated, act equally but place "1".

Example for $n = 5$:

$$\begin{aligned} (x_1 \vee x_3 \vee \neg x_5) &= \texttt{1-1-0} \\ (x_2 \vee \neg x_4 \vee \neg x_5) &= \texttt{-1-00} \end{aligned}$$

A CNF is, within this document and related program code, notated in Clause Line Notation. I decided to do this to have a more characteristic visualization that shall be easier to understand than similar-looking, indexed 'x'-literals.

Please notice that in the solver code (implementation) n is called **DigitNumber**, as n sets the number of digits/chars a Clause Line consists of. The chars of a Clause Line are, within the solver code, called 'digits' to avoid exchanging the C++ variable type named 'char' with the Clause Line chars.

Definition 4.1.3 *Two clauses C_1 and C_2 are said to be* **in conflict** *if C_1 has a "0" char at a position where C_2 has a "1" char, or vice versa. C_1 containing "-" at a location where C_2 has "0" or "1", or vice versa, does not yet mean C_1 and C_2 are in conflict.*

Definition 4.1.4 *A* **Possible Solution** *S of the CNF shall be a word consisting of exactly n "0" or/and "1" chars, but no "-" chars. There are exactly 2^n Possible Solutions.*

Definition 4.1.5 *The* **Clause Type** *of any clause informs about the positions where, within the clause, the "0" or/and "1" chars are located. For 3-SAT, there are $\binom{n}{3}$ many different Clause Types. If wanting to distinguish between different Clause Types, the "0"/"1" chars shall be replaced by "x", for notation purposes.*

Example:

ClauseLine	*ClauseType*
0-0-0	x-x-x
10-0-	xx-x-

Definition 4.1.6 *There are, for 3-SAT, 2^3 possibilities to replace the "x" chars of a Clause Type by "0"/"1" chars, e.g.* `xxx => 000, 001, 010, 011, 100, 101, 110, 111`. *After replacing the "x" chars of a Clause Type, the result shall be called* **Filled Clause Type(s)**.

Definition 4.1.7 *Let the* **Possible Clauses**, *abbreviated 'PC(s)', be the set of all $2^3 \times \binom{n}{3}$ 3-SAT clauses that can be built at all. The* **Possible Clause Number** *is the size of that set, as just calculated. There is a special order of the clauses in the set, in such a way that all $2^3 = 8$ Filled Clause Types for a Clause Type do appear in a succession within the set of Possible Clauses.*

Example: Possible Clauses = {`000-`, `001-`, `010-`, `011-`, `100-`, `101-`, `110-`, `111-`, `00-1`, `01-0`, ..., `-111`}

Definition 4.1.8 *We say a Possible Solution S '***satisfies***' a Possible Clause C if S has at least one "0" or "1" char at the same position as within C. Example: S =*`000000` *satisfies C =*`1-01--` *as both have a "0" at 3rd position.*

Proof: Char S_i is, for any $i = 1...n$, either "0" or "1". Those two states do, as defined in 4.1.2, mean either $false$ ("0") or $true$ ("1"). The same applies to any clause in Clause Line Notation, whereby clauses might contain also "-" chars. But "-" chars can, in what concerns the satisfaction test, be ignored because a "-" char at position i within C_i means that the literal i did not exist in the original, mathematical clause notation and thus doesn't decide if the clause is satisfied or not. So $S_i = C_i = \{0, 1\}$ states C as satisfied, because it corresponds to the mathematical notation $S_i = C_i = \{false, true\}$. □

Observation 4.1.9 *It is not possible to satisfy all $2^3 = 8$ Filled Clause Types by one Possible Solution.*

Proof: This can easily be shown by trying out to satisfy e.g. `000, 001, 010, 011, 100, 101, 110, 111` – it won't work. Eventual "-" chars within the Filled Clause Types (e.g. `0-0-0, 0-0-1, 0-1-0, 0-1-1, 1-0-0, ...`) do not affect this impossibility. On the other hand, 7 or less Filled Clause Types can be satisfied by some Possible Solution, as can be tried out as well. □

Definition 4.1.10 *The* **Clause Table**, *abbreviated 'CT', is a formula in disjunctive normal form with 2^n OR-ed terms and $\binom{n}{3}$ AND-ed Possible Clauses within each OR-term. The j-th OR-ed term of the CT shall be called the j-th* **CT Line**. *The set of the i-th Possible Clauses within each of the AND-terms shall be called the i-th* **CT Column**. *Each column of the CT contains the same Clause Type, but not necessarily the same clauses. The clauses of each CT Line are all pair-wise not in conflict. Each CT Line has an associated $\{0, 1\}^n$ string called* **Underlaying (Possible) Solution** *and contains all clauses that are not in conflict with that Underlaying Solution. The clauses within the CT can be seen as Boolean variables that are, to evaluate the CT, replaced by either true or $false$. A clause within the CT is replaced by true if it is not existing in the CNF to solve. If a clause exists in the CNF, it is, within the CT, replaced by $false$.*

Example: The following DNF forms the Clause Table for 3-SAT, $n = 4$:

$$
\begin{aligned}
&(\texttt{000-} \wedge \texttt{00-0} \wedge \texttt{0-00} \wedge \texttt{-000}) \vee \\
&(\texttt{000-} \wedge \texttt{00-1} \wedge \texttt{0-01} \wedge \texttt{-001}) \vee \\
&(\texttt{001-} \wedge \texttt{00-0} \wedge \texttt{0-10} \wedge \texttt{-010}) \vee \\
&(\texttt{001-} \wedge \texttt{00-1} \wedge \texttt{0-11} \wedge \texttt{-011}) \vee \\
&(\texttt{010-} \wedge \texttt{01-0} \wedge \texttt{0-00} \wedge \texttt{-100}) \vee \\
&(\texttt{010-} \wedge \texttt{01-1} \wedge \texttt{0-01} \wedge \texttt{-101}) \vee \\
&(\texttt{011-} \wedge \texttt{01-0} \wedge \texttt{0-10} \wedge \texttt{-110}) \vee \\
&(\texttt{011-} \wedge \texttt{01-1} \wedge \texttt{0-11} \wedge \texttt{-111}) \vee \\
&(\texttt{100-} \wedge \texttt{10-0} \wedge \texttt{1-00} \wedge \texttt{-000}) \vee \\
&(\texttt{100-} \wedge \texttt{10-1} \wedge \texttt{1-01} \wedge \texttt{-001}) \vee \\
&(\texttt{101-} \wedge \texttt{10-0} \wedge \texttt{1-10} \wedge \texttt{-010}) \vee \\
&(\texttt{101-} \wedge \texttt{10-1} \wedge \texttt{1-11} \wedge \texttt{-011}) \vee \\
&(\texttt{110-} \wedge \texttt{11-0} \wedge \texttt{1-00} \wedge \texttt{-100}) \vee \\
&(\texttt{110-} \wedge \texttt{11-1} \wedge \texttt{1-01} \wedge \texttt{-101}) \vee \\
&(\texttt{111-} \wedge \texttt{11-0} \wedge \texttt{1-10} \wedge \texttt{-110}) \vee \\
&(\texttt{111-} \wedge \texttt{11-1} \wedge \texttt{1-11} \wedge \texttt{-111})
\end{aligned}
$$

We can identify $\binom{4}{3} = 4$ CT Columns and $2^4 = 16$ CT Lines. The Underlaying Solutions are `0000` for the first CT Line, `0001` for the second one, `0010` for the third one, `0011` for the fourth one, `0100` for the fifth one and so on.

The order of the CT Columns and CT Lines can be chosen arbitrarily. In my solver implementation I used a kind of lexicographical order.

Observation 4.1.11 *A CNF is solvable if, and only if, all Possible Clauses of at least one CT Line are <u>all</u> absent from the CNF to solve, i.e. those PCs do not exist in the CNF. For instance, a 3-SAT CNF for $n = 4$ would have a solution if at least `011-, 01-1, 0-11, -111` would not appear in that CNF*[1]*.*

Proof: Within one CT Column, there are notated *all* $2^3 = 8$ possible Filled Clause Types. According to 4.1.9, it is *not* possible to satisfy *all* of those clauses using some Possible Solution. If at least one CT Line is not to be satisfied as all its clauses are absent from the CNF, then in each CT Column there's at least one Filled Possible Clause missing, so the Clause Table DNF can be satisfied as *every* CT Column can be satisfied. Please notice that either *all* instances of the same clause are absent from the CNF, or none. For instance, either both `000-` clauses in the first two CT Lines, first column, are absent or none of them. This means that all copies of a PC within the Clause Table are either replaced by *true* or are all replaced by $false$. It is *not* possible that some are replaced by *true* and others by $false$. □

4.2 The Polynomial 3-SAT-Algorithm

The main idea is to detect if all Possible Clauses of at least one CT Line are all absent from the CNF to solve. If all those Possible Clauses should be absent, the CNF is solvable, as mentioned in 4.1.11. Please notice: Although the Clause Table has exponentially many CT Lines, the upcoming algorithm can evaluate it in polynomial time and space! So although the Clause Table is actually 'too large', this fact does *not* mean that building up a polynomial algorithm upon the idea of the CT needs to fail.

But first, some more definitions that will later be necessary to describe the working steps of the polynomial algorithm:

[1] Please look up the mentioned absent PCs in the CT example of 4.1.10

Definition 4.2.1 *A* **Valid Clause Triple** *shall be the set of three Possible Clauses i, j and k, which must be all not in conflict pair-wise. Notice that the notation order of the PCs within the triple is important, k shall always be written at last.*

Definition 4.2.2 *Let an* **Active Clause Table Line** *be a CT Line that consists only of Possible Clauses that are absent from the CNF to solve, i.e. those PCs do all not exist in the CNF.*

In the upcoming code listing, we will see that the solver implementation uses a Boolean array `InActiveCTLine[x]` that tells if a clause x is marked as within any Active CT Line.

Observation 4.2.3 *If there is at least one Active CT Line, the CNF is solvable.*

Proof: Any Active CT Line fulfills the conditions of 4.1.11 and 4.2.2, what implies the CNF is solvable. □

Definition 4.2.4 *We name a Valid Clause Triple an* **Active Clause Triple** *if all clauses i, j and k are marked as within an Active CT Line, i.e.* `InActiveCTLine[i]==true` *and* `InActiveCTLine[j]==true` *and* `InActiveCTLine[k]==true`*.*

Observation 4.2.5 *If three Possible Clauses are not in conflict pair-wise, there is at least one CT Line that contains all those three PCs.*

Proof: If three clauses i, j, k are not in conflict, they are all not in conflict with at least one Possible Solution. We know that for *each* Possible Solution there is exactly one CT Line that contains all clauses not in conflict with that Possible Solution. So there is at least one CT Line that contains the clauses i, j, and k. □

4.2.1 Code Listing

Code Listing 4.2.6 *It follows a simplified version of the original implementation (see also the upcoming topic "Implementation on the Internet"). The programming language is C++. You might first have a brief look at the code and study it in detail later, after having read the explanations behind the code listing. Please notice that the code of helper procedures whose functionality is clear or explained precisely within this paper is not shown explicitly.*

```
int         PossibleClauseNumber = 0;
char        PossibleClauses[POSSIBLE_CLAUSE_NUMBER_MAX][DIGIT_NUMBER_MAX];

int         ClauseTypeNumber = 0;
char        ClauseTypes[POSSIBLE_CLAUSE_TYPE_NUMBER_MAX][DIGIT_NUMBER_MAX];

#define     SATISFIABLE     true
#define     UNSATISFIABLE   false

bool CNF_Solve_Polynomial_Core()
{
    InitializePossibleClauses(PossibleClauseNumber, PossibleClauses); // [content not listed i. d.]
    InitializeClauseTypes(ClauseTypeNumber, ClauseTypes); // [content not listed in detail]

    bool    InActiveCTLine[POSSIBLE_CLAUSE_NUMBER_MAX];
```

```
for (int i = 0; i < PossibleClauseNumber; i ++)
    InActiveCTLine[i] = !ExistsInCNF(PossibleClauses[i]); // [ExistsInCNF() not listed in detail,
                                                          //  returns true if passed clause in
                                                          //  CNF to solve; '!': negation]
bool    ChangesExisting = true;

while (ChangesExisting)
{
    ChangesExisting = false;

    for (int k = 0; k < PossibleClauseNumber; k ++)
    {
        for (int i = 0; i < ClauseTypeNumber; i ++)
        {
            int     FilledClauseTypeNumber = 0; // becomes 8 in GetFilledClauseTypes() (3-SAT)
            char    FilledClauseTypesi[FILLED_CLAUSE_TYPE_NUMBER_MAX][DIGIT_NUMBER_MAX];

            // convert a Clause Type into Filled Clause Types [not listed explicitly]
            GetFilledClauseTypes(ClauseTypes[i], FilledClauseTypeNumber, FilledClauseTypesi);

            for (int j = 0; j < i; j ++)
            {
                FilledClauseTypeNumber = 0;
                char    FilledClauseTypesj[FILLED_CLAUSE_TYPE_NUMBER_MAX][DIGIT_NUMBER_MAX];

                GetFilledClauseTypes(ClauseTypes[j], FilledClauseTypeNumber, FilledClauseTypesj);

                bool    ijk_InSameActiveCTLine = false;

                for (int FCTi = 0; FCTi < FilledClauseTypeNumber; FCTi ++)
                {
                    // InSameCTLine = !IsInConflict(clause1, clause2) [not listed explicitly]
                    if (!InSameCTLine(FilledClauseTypesi[FCTi], PossibleClauses[k]))
                        continue;

                    for (int FCTj = 0; FCTj < FilledClauseTypeNumber; FCTj ++)
                    {
                        if (!InSameCTLine(FilledClauseTypesj[FCTj], PossibleClauses[k]) ||
                            !InSameCTLine(FilledClauseTypesj[FCTj], FilledClauseTypesi[FCTi]))
                            continue;

                        // we can calculate the PossibleClauseIndices of the Filled Clause Types
                        // FCTi and FCTj as follows, because the PCs have a well-defined order:
                        int PCIndexFCTi = i * CombinationCount + FCTi; // CombinationCount=8...
                        int PCIndexFCTj = j * CombinationCount + FCTj; // ...for 3-SAT.

                        if (InActiveCTLine[PCIndexFCTi] &&
                            InActiveCTLine[PCIndexFCTj])
                        {
                            ijk_InSameActiveCTLine = true; // (if also InActiveCTLine[k]==true,)
                            goto LeaveFCTLoops;            // there's any Active CT Line...
                        }                                  // ...containing k!
                    }
                }
```

```
LeaveFCTLoops:;
                    if (!ijk_InSameActiveCTLine)
                    {
                        if (InActiveCTLine[k] == true)
                        {
                            InActiveCTLine[k] = false; // k is not in any Active CT Line;
                            ChangesExisting = true; // (later) repeat whole k,i,j,FCTi,FCTj loops;
                        }
                    }
                }
            }
        }
    }

    for (int i = 0; i < PossibleClauseNumber; i ++)
    {
        if (InActiveCTLine[i])
        {
            return SATISFIABLE;
        }
    }

    return UNSATISFIABLE;
}
```

4.2.2 Implementation Details

First, the solver initializes the Boolean array `InActiveCTLine[PossibleClauseIndex]`. This array tells the solver if any copy of a Possible Clause (identified by the PC index) from the Clause Table is within an Active CT Line. At the beginning, a PC is seen as within an Active CT Line if that clause is absent from the CT. This means that the content of `InActiveCTLine[]` does actually not yet tell if the clause is really in such an Active CT Line, not until the solving procedure has finally finished. (Notice that an Active CT Line must consist of clauses that are *all* missing in the CNF. It is clear that this is not yet determinable from the initial state of `InActiveCTLine[]` as each PC has been, until now, processed independently.)

After having initialized `InActiveCTLine[]`, the solver enters a while-loop containing further five for-loops. The first three for-loops use the index variables i, j and k. k points to a Possible Clause whose `InActiveCTLine[]` value is to be updated. i and j do each point to a Possible Clause Type. Within the i, j and k loops there are further iterations done: The $FCTi$ loop runs through the Filled Clause Types created out of the Possible Clause Type i. The purpose of the $FCTj$ loop is analogue, it iterates through the Filled Clause Types created out of the Possible Clause Type j.

If a Valid Clause Triple can be built out of the clauses corresponding to $FCTi$, $FCTj$ and k, the solver checks if any of those Valid Clause Triples contains Possible Clauses that are all *marked* as being within any Active CT Line. If this should *not* be the case for *all* $2^3 \times 2^3 = 64$ possible $FCTi/FCTj$ combinations, Possible Clause k is marked as *not (any more)* in any Active CT Line.

We repeat the i, j and k loops within the while-loop until `InActiveCTLine[]` did not change any more. If now any array element of `InActiveCTLine[]` should (still) be *true*, we finally know that there is a complete Active CT Line within the Clause Table belonging to the CNF to solve, so the CNF is solvable. We do not know which concrete solution (model) solves the CNF, but we know that there is at least one solution. If `InActiveCTLine[]` is completely $false$, there is no solution.

So a clause k within the Clause Table keeps its 'in an Active CT Line' state only in the following case:

- Starting-point is that i and j do each point to two different CT Columns.
- Furthermore, $FCTi$ and $FCTj$ point to one Possible Clause within CT Column i respectively j.
- Now, if there is at least one clause within each of those two CT Columns that could build a Valid Triple with k, and the two clauses from those two CT Columns are both marked as within an Active CT Line, then clause k keeps its eventual marking as within some Active CT Line.
- If it would not be possible to build a Valid Triple out of k, $FCTi$ and $FCTj$, then the value of `InActiveCTLine[k]` stays unchanged.
- If it is possible to build one or more Valid Triple(s) but not enough involved clauses, i.e. not at least one $FCTi/FCTj$ pair, do have the 'in any Active CT Line' marking, then `InActiveCTLine[k]` is set to $false$.

Observation 4.2.7 *By studying the code of code listing 4.2.6, we can observe that the value of `InActiveCTLine[k]` can, after a one-time initialization, only change from true to false. Figuratively, `InActiveCTLine[k]` changes from true to false if there are not enough other clauses existing which would be needed to form an Active CT Line with k. In some kind of 'domino-effect' `InActiveCTLine[k]` flags can get unset.*

4.2.3 Proof of Correctness

I want to split up the proof of correctness into several parts. First, we'll see that the polynomial solver does never say 'unsatisfiable' when the CNF is satisfiable. Afterwards we examine the opposite case, whereby two 'sub-cases' are proven separately.

4.2.3.1 Solvable Detection

Observation 4.2.8 *The polynomial algorithm shown in 4.2.6 detects reliably when the CNF to solve is satisfiable, that means when the CNF does have a solution.*

Proof: According to 4.2.3, the CNF is solvable if there's at least one Clause Table Line whose clauses are *all* absent from the CNF. This CT Line is called 'Active CT Line'. By regarding 4.2.6 carefully, you can notice that `InActiveCTLine[k]` is initialized to *true* for all clauses k that appear in the Active CT Line.

Within the $FCTi$ and $FCTj$ loops, we check if there are, for each clause k out of the Active CT Line, two other clauses C_{FCTi} and C_{FCTj} that are within the same CT Line as k, and that have both set `InActiveCTLine[FCTi/FCTj]` to *true*. We will, for all those clauses k, find *at least* C_{FCTi} and C_{FCTj} out of exactly the Active CT Line.

The algorithm will finally return 'SATISFIABLE' as at least all `InActiveCTLine[x]` are *true* for all clauses x out of the Active CT Line. □

Here's a sketch how the Clause Table could look like in the described case:

$$
\begin{array}{l}
(\ldots \wedge \ldots \wedge \ldots \wedge \ldots \wedge \ldots)\vee \\
(\texttt{true} \wedge \texttt{true} \wedge \texttt{true} \wedge \texttt{true} \wedge \texttt{true})\vee \\
(\ldots \wedge \ldots \wedge \ldots \wedge \ldots \wedge \ldots)\vee \\
\ldots
\end{array}
$$

Notice that '...' within the braces can be either *true* or $false$. The size of the Clause Table depends on the dimensions of the CNF to solve.

4.2.3.2 Non-solvable Detection

Observation 4.2.9 *The polynomial algorithm shown in 4.2.6 detects reliably when the CNF to solve is unsatisfiable, that means when the CNF has no solution.*

Proof Part 1: We regard the case that there are *all* clauses from at least one CT Column within the CNF to solve. That means that *all* clauses of this CT Column are replaced by $false$, what makes the CT DNF as a whole be evaluated to $false$, and so we know the CNF to solve is unsatisfiable. Let the clauses out of this CT Column, as well as the CT Column itself, be called 'missing'.

In such a case `InActiveCTLine[k]` will get $false$ as soon as i or j points to the Filled Clause Types of the missing CT Column. Furthermore, for any k, `InActiveCTLine[k]` will become $false$ after having checked all Valid Triples that are built out of k and at least one out of $FCTi$ or $FCTj$ from the missing CT Column.

That means that the clauses x of more and more CT Columns get `InActiveCTLine[x]:=false` resulting in the state that finally there's no `InActiveCTLine[x]` set to *true* any more. So the algorithm will return 'UNSATISFIABLE', as expected. □

One possible Clause Table corresponding to the proof conditions could be:

(... ∧ false ∧ ... ∧ ... ∧ ...)∨
(... ∧ false ∧ ... ∧ ... ∧ ...)∨
(... ∧ false ∧ ... ∧ ... ∧ ...)∨
...

Notice that the index of the CT Column that contains exclusively $false$-values is not fixed, it can be chosen arbitrarily.

Proof Part 2: The following case is resting: In each CT Line, there is at least one clause replaced by $false$, because it is existing in the CNF to solve. The difference to the second case (the previous 'Proof Part 1') is that the CT does not have one whole CT Column set to $false$ but this value appears in an arbitrary CT Column for every CT Line.

The Clause Table could look like as in the following schematic example:

(... ∧ false ∧ ... ∧ ... ∧ ...)∨
(... ∧ ... ∧ false ∧ ... ∧ ...)∨
(... ∧ ... ∧ ... ∧ false ∧ ...)∨
(false ∧ ... ∧ ... ∧ ... ∧ ...)∨
...

The $false$ values are 'spread' over the whole CT, the only rule is that every CT Line contains at least one $false$.

To prove that the polynomial solver will return 'UNSATISFIABLE' also in the case that in each CT Line, there's at least one $false$ in an arbitrary CT Column, I want to go far back:

We know that we can determine the solvability of the 3-SAT CNF by replacing the clauses within the corresponding CT DNF by *true* or $false$ and then just evaluating the DNF.

In the case of interest that each CT Line contains at least one $false$, we can easily see that the whole CT DNF gets $false$, as the AND-ed Boolean values need to be *all true* in at least one CT Line, what is not the case. Although we can easily show that the Boolean result of the CT DNF can be computed, this does not yet help in what concerns the polynomial solver's algorithm, as the CT DNF has exponential size and such this computation cannot be done 1-to-1 by the polynomial solver.

Nevertheless the CT DNF is the key to the proof of correctness, because we can transform the CT DNF into a special CNF, to be called the Deployed CNF, for which we will see that it describes the proceeding of the polynomial solver. So when the Deployed CNF always returns the same results as the Clause Table DNF, and the polynomial solver always returns the same results as the Deployed CNF, then we can say the polynomial solver is correct as it always returns the same result as the Clause Table DNF.

Definition 4.2.10 *The* **Deployed CNF** *is a formula in conjunctive normal form that is created out of the Clause Table DNF according to the following scheme:*

The CT DNF consists of clauses c_{ol} that are organized in o many CT Columns and l many CT Lines:

$$\begin{array}{l}(c_{11} \wedge c_{21} \wedge c_{31} \wedge c_{41} \wedge ... \wedge c_{o1}) \vee \\ (c_{12} \wedge c_{22} \wedge c_{32} \wedge c_{42} \wedge ... \wedge c_{o2}) \vee \\ (c_{13} \wedge c_{23} \wedge c_{33} \wedge c_{43} \wedge ... \wedge c_{o3}) \vee \\ ... \\ (c_{1l} \wedge c_{2l} \wedge c_{3l} \wedge c_{4l} \wedge ... \wedge c_{ol})\end{array}$$

The Deployed CNF therefore is[2]:

$$\begin{array}{l}((c_{11} \wedge c_{21} \wedge c_{31}) \vee (c_{12} \wedge c_{22} \wedge c_{32}) \vee (c_{13} \wedge c_{23} \wedge c_{33}) \vee ... \vee (c_{1l} \wedge c_{2l} \wedge c_{3l})) \wedge \\ ((c_{11} \wedge c_{21} \wedge \underline{c_{4}}_{1}) \vee (c_{12} \wedge c_{22} \wedge c_{32}) \vee (c_{13} \wedge c_{23} \wedge c_{33}) \vee ... \vee (c_{1l} \wedge c_{2l} \wedge c_{3l})) \wedge \\ ((c_{11} \wedge \underline{c_{3}}_{1} \wedge \underline{c_{4}}_{1}) \vee (c_{12} \wedge c_{22} \wedge c_{32}) \vee (c_{13} \wedge c_{23} \wedge c_{33}) \vee ... \vee (c_{1l} \wedge c_{2l} \wedge c_{3l})) \wedge \\ ((\underline{c_{2}}_{1} \wedge \underline{c_{3}}_{1} \wedge \underline{c_{4}}_{1}) \vee (c_{12} \wedge c_{22} \wedge c_{32}) \vee (c_{13} \wedge c_{23} \wedge c_{33}) \vee ... \vee (c_{1l} \wedge c_{2l} \wedge c_{3l})) \wedge \\ ((c_{11} \wedge c_{21} \wedge c_{31}) \vee (c_{12} \wedge c_{22} \wedge \underline{c_{4}}_{2}) \vee (c_{13} \wedge c_{23} \wedge c_{33}) \vee ... \vee (c_{1l} \wedge c_{2l} \wedge c_{3l})) \wedge \\ ... \\ ((c_{(o-2)_1} \wedge c_{(o-1)_1} \wedge c_{(o)_1}) \vee (c_{(o-2)_2} \wedge c_{(o-1)_2} \wedge c_{(o)_2}) \vee (c_{(o-2)_3} \wedge c_{(o-1)_3} \wedge c_{(o)_3}) \vee ... \vee (c_{(o-2)_l} \wedge c_{(o-1)_l} \wedge c_{(o)_l}))\end{array}$$

On first sight, it might not be obvious what was done. But it is easy to explain: The CT DNF became the Deployed CNF by using the distributive law for Boolean expressions. You know the distributive law already from school. For example, $(1*2*3)+(4*5*6)$ can be transformed to $(1+4)*(1+5)*(1+6)*(2+4)*(2+5)*(2+6)*(3+4)*(3+5)*(3+6)$. Instead of doing this with multiplication ($*$) and addition ($+$), we apply the distributive law to AND ($\wedge$) and OR ($\vee$) connections, which is also valid, see e.g. [7].

This means, due to the distributive law, that generally:

$$\begin{array}{l}(A \wedge B \wedge C) \vee \\ (E \wedge F \wedge G)\end{array}$$

is equal to:

$$\begin{array}{l}(A \vee E) \wedge (A \vee F) \wedge (A \vee G) \wedge \\ (B \vee E) \wedge (B \vee F) \wedge (B \vee G) \wedge \\ (C \vee E) \wedge (C \vee F) \wedge (C \vee G)\end{array}$$

Now comes an important trick necessary to prove the correctness of the polynomial solver: We substitute systematically groups of three values within the Clause Table, use the distributive law to convert the (current part of the) CT to the corresponding part of the Deployed CNF, and finally re-substitute the groups of three clauses in the Deployed CNF.

To clarify this proceeding, please have a look at the following example, which uses, in contrast to the Clause Table, variables instead of clauses for better understandability.

Generally, the input DNF:

$$\begin{array}{l}(A \wedge B \wedge C \wedge D) \vee \\ (E \wedge F \wedge G \wedge H)\end{array}$$

[2]As a help for the reader, changing indices are underlined

would become by the aid of substitution and the distributive law exactly the following CNF:

$$((A \wedge B \wedge C) \vee (E \wedge F \wedge G)) \wedge ((A \wedge B \wedge D) \vee (E \wedge F \wedge G)) \wedge$$
$$((A \wedge C \wedge D) \vee (E \wedge F \wedge G)) \wedge ((B \wedge C \wedge D) \vee (E \wedge F \wedge G)) \wedge$$
$$((A \wedge B \wedge C) \vee (E \wedge F \wedge H)) \wedge ((A \wedge B \wedge D) \vee (E \wedge F \wedge H)) \wedge$$
$$((A \wedge C \wedge D) \vee (E \wedge F \wedge H)) \wedge ((B \wedge C \wedge D) \vee (E \wedge F \wedge H)) \wedge$$
$$((A \wedge B \wedge C) \vee (E \wedge G \wedge H)) \wedge ((A \wedge B \wedge D) \vee (E \wedge G \wedge H)) \wedge$$
$$((A \wedge C \wedge D) \vee (E \wedge G \wedge H)) \wedge ((B \wedge C \wedge D) \vee (E \wedge G \wedge H)) \wedge$$
$$((A \wedge B \wedge C) \vee (F \wedge G \wedge H)) \wedge ((A \wedge B \wedge D) \vee (F \wedge G \wedge H)) \wedge$$
$$((A \wedge C \wedge D) \vee (F \wedge G \wedge H)) \wedge ((B \wedge C \wedge D) \vee (F \wedge G \wedge H))$$

Now let me give you further information about the Deployed CNF.

Definition 4.2.11 *The Deployed CNF consists of exactly* $\binom{o}{3}^l$ *many AND-ed terms, whereby* o *is the count of columns in the Clause Table DNF, and* l *is the count of lines in the Clause Table DNF. These AND-ed terms are to be called the* **Deployed Groups**.

One Deployed Group in the example above would be $((A \wedge B \wedge C) \vee (E \wedge F \wedge G))$.

Definition 4.2.12 *Each Deployed Group of the Deployed CNF consists of exactly* l *many clause triples, which are to be called the* **Deployed Triples**. l *is again the number of CT Lines of the Clause Table DNF which was transformed to the Deployed CNF.*

One Deployed Triple in the example above would be $(A \wedge B \wedge C)$, or also $(E \wedge F \wedge G)$.

But, how can one describe what Deployed Groups and Deployed Triples are and how they were created? Here's a linguistic description: One Deployed Triple consists of exactly three clauses from one and the same CT Line. When we take exactly one Deployed Triple from each CT Line and we 'OR' those triples, we get one Deployed Group.

The whole Deployed CNF consists of all possible Deployed Groups. Notice that the clauses within triple T_1 of a Deployed Group need *not* to be taken from the same CT Columns as some further triple T_2 from that group. Really *all* possible triple combinations must be involved to make the Deployed CNF always return the same result as the Clause Table DNF. For better understanding, you please might want to study the general example that transformed $(A \wedge B \wedge C \wedge D) \vee (E \wedge F \wedge G \wedge H)$, as listed above.

Observation 4.2.13 *The correctness of the polynomial solver, as introduced in 4.2.6, can be shown with the help of the Deployed CNF.*

Proof: What we need to show is that the polynomial solver's working steps can be mathematically described by the Deployed CNF, as announced at the beginning of 'Proof Part 2'.

When *regarding the polynomial solver's code* (4.2.6), we see the following:

(1) `InActiveCTLine[k]` stays *true* if there is an Active Clause Triple that contains clause k. The Active Clause Triple can be located in any CT Line, but there must be at least one Active Clause Triple for each possible combination of the two CT Columns i, j and the column containing clause k.

(2) The ladder conditions are implemented as follows: k loops through all Possible Clauses, and therewith points to all CT Columns, one after another. For each k, i and j point to two further different CT Columns. *FCTi*

and $FCTj$ point to Possible Clauses within CT Columns i and j. $PCIndexFCTi$ and $PCIndexFCTj$ are the (implementation-intern) indices of the Possible Clauses that $FCTi$ and $FCTj$ point to. `InActiveCTLine[k]` stays *true* `if (InActiveCTLine[PCIndexFCTi]==true && InActiveCTLine[PCIndexFCTj]==true)`, what means that there is one Active Clause Triple consisting of k, $PCIndexFCTi$ and $PCIndexFCTj$. We see the CT Line (index) is not checked in any form at all as i, j point to CT Columns only. For each triple (i, j, k), there are $2^3 \times 2^3$ $FCTi/FCTj$ combinations that are sufficient to keep `InActiveCTLine[k]` *true*.

(3) So the implementation will keep any `InActiveCTLine[k]` *true* if there is, for each CT Column triple (i, j, k) at least one CT Line that contains clauses from each of those CT Columns, being marked as 'within any Active CT Line'. If such a triple is found for each possible CT Column triple, the solver will return 'SATISFIABLE', otherwise `InActiveCTLine`-values will get $false$ one after another, in the already mentioned 'domino-effect', and the solver returns 'UNSATISFIABLE'.

(4) When regarding the Deployed CNF, we see that *all* Deployed Groups must be *true* to get the whole Deployed CNF *true*, what means 'SATISFIABLE'. If at least one Deployed Group is not *true*, the Deployed CNF is $false$, what means 'UNSATISFIABLE'.

(5) One Deployed Group consists of l many clause triples, the Deployed Triples, whereby there is exactly one clause triple for each CT Line. As the Deployed Triples are OR-ed, this means there must be merely one Deployed Triple *true* within each Deployed Group, meaning there must be only one Active CT Line for each CT Column triple. This describes exactly the polynomial solver's proceeding as mentioned in point (3). □

To come back to the initial question of Proof Part 2: Yes, we can finally be sure the polynomial solver will detect a CNF whose Clause Table contains 'one $false$ in every CT Line' as 'UNSATISFIABLE' as there is at least one Deployed Group that consists of Deployed Triples that do each contain at least one of the assumed $false$ clauses. As the polynomial solver always returns the same result as the Deployed CNF, the polynomial solver will return 'UNSATISFIABLE', as expected. (Proof Part 2: □)

Another, important question comes up: We know the CT DNF has exponential size, as there are 2^n many CT Lines. When we deploy the CT DNF, we get an even larger formula. How can we evaluate this enormously large deployed formula in polynomial time?

The answer is the following: Luckily, the CT DNF has, by design, such a special pattern so that clause triples repeat among the CT Lines. This repetition appears more and more the larger the CT is. We can be sure, by examining the CT structure, that there are never more than $O(n^9)$ clause triples, as there are $O(n^3)$ many Possible Clauses and thus $O((n^3)^3)$ Possible Clause triples.

While for arbitrary formulas in disjunctive normal form (DNF), there might be an exponentially growing count v of variables if the DNF grows exponentially with the input size and if the DNF contains each variable once only. But this is *not* the case for CT DNFs, because the count of Possible Clauses (the 'variables') has a polynomial upper bound.

So we can say that the fact that the same clause triples appear very often within the CT makes it possible to evaluate the CT DNF in polynomial time and space.

Some final remarks:

- The currently given third proof can actually also be used to show that the first two proven cases (one CT Line completely *true*, or one CT Column completely $false$) are decided correctly by the polynomial solver. Please do nevertheless take the proofs given for the first two cases as an additional safety.
- Within the download-able solver code (see the upcoming topic "Implementation on the Internet"), there is a test program included that was used to check out if the conversion of a DNF to the Deployed CNF does really work. I did over two hundred million test runs, there was no error. Please download the zip file and view the code under the directory 'Tests - ...', if you are interested.

4.2.4 Further Notes about the Polynomial Algorithm

Notice that the order in which we iterate through i, j, k, $FCTi$, $FCTj$ is not important. My previous Algorithm B, as mentioned in this paper's section 2, was 'sensitive' to the running order of the nested loop it uses. It strongly seems as if I unknowingly programmed the order of those loops in a way so that the solver always returned correct results. It was the work of Mr. Prunescu who found out that Algorithm B fails for some CNFs if you permute some internal loops. You can read about this in his paper [8]. I thank Mr. Prunescu very much for finding out and informing me about this circumstance.

It might also be interesting to you that I tried to get a concrete solution ($\{0,1\}^n$ word) out of the polynomial solver (Algorithm C), in addition to the output 'CNF is solvable' or 'CNF is not solvable'. I tried to 'put together' all Possible Clauses c_x that are, at the end of the polynomial solving process, marked as being within an Active CT Line (`InActiveCTLine[`c_x`]==true`). 'Put together' shall mean writing any Active CT Line-clause c_x to a 'growing' solution word $s \in \{0,1,?\}$ if c_x is not in conflict with the already-written "0" or "1" chars within s. The initial "?" chars in s get overwritten by $c_{xj} =$ "0" or $c_{xj} =$ "1". My half-hearted attempt did not work so far, the gotten solutions did sometimes not satisfy the CNF to solve. In my opinion it is probable that getting a concrete solution could work, I just did not do it right. Improving the polynomial solver in this way could be the goal of further work and research.

As last note I want to point out that the real run-time of the polynomial solver (Algorithm C) has, in all tests I did, been much smaller than the $O(n^{12})$ that was mentioned in the abstract and that will be established in the upcoming topic. The while-loop that is repeated if `ChangesExisting==true` was in tests never iterated more than 4 to 5 times, although the theoretical upper bound of the iteration count is much greater. This is due to the fact that mostly more than the theoretical one `InActiveCTLine[]` value is set to $false$ before repeating the while-loop.

4.2.5 Open Questions

Although I tried to find out as much as possible about the polynomial SAT solving algorithm, there are at least two odd facts I cannot explain (completely) yet:

First, why do we need Active Clause Triples, and not Active Clause Tuples? When you change the algorithm in the way so that you remove the $FCTj$ loop, the solver frequently returns wrong results. By involving the concept of Active Clause Triples, the solver never failed, for millions of test runs.

To answer the question why we need triples, not tuples, I though about setting up a proof that classifies my polynomial algorithm as an equivalence relation. As you can read in math-books (e.g. [7]), you can categorize three arbitrary (mathematical) objects x, y and z as 'equivalent' if the relation that categorizes the 'equivalence' is reflective, symmetric and transitive. I thought about if this applies to the idea of the Active Clause Triples, as you need three elements (a triple) to check if the transitivity is fulfilled. But I could progress far with this idea.

Another approach to explain the need for triples is the following: Within the polynomial solver, the clause belonging to k has exactly 3 "0"/"1" chars. Clause $FCTi$ is never equal to clause k, what can be derived from the solver code. So clause k has at least one "0" or "1" char that is not 'covered' by $FCTi$. If we know that clause $FCTi$ is in an Active CT Line, but *we cannot identify that line sufficiently.*

This shall be made clear in the following example: Have a look at the following excerpt from a Clause Table:

$$(FCTi = \texttt{000-} \wedge FCTj = \texttt{?} \wedge k_1 = \texttt{-000})$$
$$(FCTi = \texttt{000-} \wedge FCTj = \texttt{?} \wedge k_2 = \texttt{-001})$$

Imagine we know that $FCTi$ is an Active CT Line. But is k_1 or k_2, or even both, in one, too?

$$(FCTi = \texttt{000-} \wedge FCTj_1 = \texttt{00-0} \wedge k_1 = \texttt{-000})$$
$$(FCTi = \texttt{000-} \wedge FCTj_2 = \texttt{00-1} \wedge k_2 = \texttt{-001})$$

Now, when regarding Active Triples, we can say without any doubt that k_1 is in an Active CT Line if $FCTi$ and $FCTj_1$ are in one, and k_2 is in an Active CT Line if $FCTi$ and $FCTj_2$ are.

The second important open question came up after I've realized the following issue in extended tests of the polynomial SAT solving algorithm:

If there *really* is a higher count of Active CT Lines ($>\approx 8$, tested by some exponential algorithm), it happens that clauses k keep `InActiveCTLine[k]=true` although these clauses k do actually not appear within the CT Lines that are 'active' in the strict sense that they consist of clauses all absent from the CNF to solve.

I thought about two explanations for this observation: Either the polynomial solver takes some clauses from several 'really' Active CT Lines and mixes them up as if they were in some further Active CT Line.

Or, the polynomial solver keeps clauses marked as 'within an Active CT Line' because those clauses once were marked as such and did not get unmarked, for any reason (remember the 'domino-effect' figuratively describing the step-by-step disabling of `InActiveCTLine[k]` values).

But please notice: Although one could think these two odd observations would mean the polynomial SAT solving algorithm could return wrong results, this never happened in all tests I did. Furthermore the proof of correctness given in this paper should strongly point into the direction the algorithm works. Possibly there are some easy explanations for the two open questions I did not find out yet.

4.3 Complexity

Observation 4.3.1 *The Polynomial 3-SAT-Solving Algorithm shown in 4.2.6 has a worst-case complexity of maximal $O(n^{12})$, whereby n is the CNF's literal range.*

Proof: There are $PCNum = 2^3 \times \binom{n}{3} = O(n^3)$ different Possible Clauses. Although $\binom{n}{3}$ can be calculated as $\frac{n!}{(n-3)!3!}$, which uses a factorial function, we can also gain $\binom{n}{3}$ by using three nested loop, each doing maximal n iterations. Three nested loops, each running from 0 to $n-1$, result in $O(n^3)$. Furthermore there are $TNum = 1 \times \binom{n}{3} = O(n^3)$ different Possible Clause Types.

As you can see in the Code Listing earlier on, there are the three nested loops i, j and k. We realize i and j do maximal $TNum$ many iterations, and k does maximal $PCNum$ many iterations. The block of loops i, j, k, etc. are repeated, within the while-loop, maximal $PCNum$ times, as with each repetition it is guaranteed to set at least one `InActiveCTLine[]` element to $false$, and there are only $PCNum$ many elements in use (although the array might be declared larger).

Finally there are left the two $FCTi$ and $FCTj$ loops, each of them is iterated maximal $2^3 = 8$ times, as there are such many Filled Clause Types for a given Clause Type.

While my implementation does another $O(n)$ loop iterations in the function calls to `InSameCTLine()`, I did not take into consideration this additional work as it would be possible to pre-compute and store in a Boolean array if two Possible Clauses are within the same CT Line.

Summarized, we have to multiply the run-time complexities of i, j, k, `InActiveCTLine[]` and $FCTi/FCTj$, so we get: $O((n^3) \times (n^3) \times (n^3) \times (n^3) \times 64) = O(n^{12})$. $\square$

5 Implementation on the Internet

I have implemented the algorithm explained in this paper as C++ console program. This program allows to 'invent' random 2- and 3-SAT CNFs which are first checked for solvability ('is there a solution?') using the brute-force, exponential way, and then by the polynomial algorithm. This can be done in a successive mass test that checks up to 100 million CNFs, without requiring user action in the meantime. If the two results of the exponential versus the polynomial solver should not be equal for any CNF, the program stops and the user is informed instantly via an error message.

The implementation has been done by me using Visual Studio 2005. This means the program is designed to run on any newer Microsoft Windows operating system, but it does probably also work on most Linux systems that have Wine installed.

The download URL of this paper, the implementation of the described polynomial SAT-solving algorithm, and a distribution law test program is given in the earlier topic 'History of Polynomial Algorithm'.

Algorithm C, as described in this paper's section 'The Polynomial 3-SAT-Algorithm', has been tested by me in form of the mentioned, download-able console program. Around 5 million random CNFs of different sizes (mainly literal ranges from 3 to 15, clause counts from 1 to 100), have been tested for solvability. No error was detected, until now (2015-05-08).

Within the download-able zip-file, which contains the just mentioned Algorithm C, there is also the original description and implementation of the older Algorithm B, which was published by me earlier. Some links in the Internet may still refer to Algorithm B, that's why you can find it in the zip file, too.

6 Summary

This paper introduced and explained a relatively simple algorithm that decides in polynomial time and space if any given 3-SAT CNF is solvable or not. An Internet link to a test implementation has been given as well. If the algorithm should be correct, it solved the P-NP-Problem by proving P is equal to NP.

7 Acknowledgments

I want to thank Mr. M. Prunescu, Simion Stoilow Institute of Mathematics of the Romanian Academy, very much for giving me precious tips on how to improve the document describing Algorithm B.

References

[1] Michael R. Garey and David S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, W. H. Freeman & Co., 1979.

[2] Christos H. Papadimitriou, Computational complexity, Addison-Wesley, 1994.

[3] Uwe Schöning, Theoretische Informatik - kurz gefasst, Bibl. Institut Wissenschaftsverlag, 1992, ISBN 3-411-15641-4.

[4] Ingo Wegener, Theoretische Informatik - eine algorithmenorientierte Einführung (3. Auflage), B. G. Teubner Verlag / GWV Fachverlage GmbH, Wiesbaden 2005, ISBN 3-8351-0033-5.

[5] Volker Heun, Grundlegende Algorithmen (2. Auflage), Friedr. Vieweg & Sohn Verlag / GWV Fachverlage GmbH, Wiesbaden 2003, ISBN 3-528-13140-3.

[6] Daniel Grieser, Mathematisches Problemlösen und Beweisen, Springer Fachmedien Wiesbaden 2013, ISBN 978-3-8348-2459-2.

[7] Bronstein, Semendjajew, Musiol, Mühlig, Taschenbuch der Mathematik, Verlag Harri Deutsch, Thun und Frankfurt am Main 2000, ISBN 3-8171-2015-X.

[8] Prunescu, Mihai, About a surprising computer program of Matthias Müller, http://imar.ro/~leustean/LogicSeminar/MihaiPrunescu-notes.pdf (link checked 2015-May-08, link also on my website, see this paper's section "Implementation on the Internet").