Polynomial Exact-3-SAT-Solving Algorithm

Matthias Michael Mueller
louis@louis-coder.com

Thu, 2016-02-25
Version D-1.1

Abstract

This article describes an algorithm that is capable of solving any instance of a 3-SAT CNF in maximal $O(n^{18})$, whereby n is the literal index 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 unsolved problems within its discipline, 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. The arising enormous computation effort entails 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 article supposes 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

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 *fourth* attempt to build a polynomial 3-SAT solver. I have developed four 3-SAT solving algorithms, which I call *Algorithm A* (published December 2012), *Algorithm B* (published December

2013), *Algorithm C* (put on-line May 2015) and the current major version from November 2015, named *Algorithm D* (the updated version D-1.1, which explains the algorithm more clearly and more detailed, was published in February 2016).

While Algorithm A and C turned out to be faulty, version B and D could not be falsified yet. With 'not falsified' I mean Algorithm B and D are of polynomial complexity for sure, as one can see from the way it is programmed, i.e. the implementation consists of nested loops only and uses no power-of-function(s) or recursive procedure calling, and they never returned wrong results in over a million test runs. The problem about Algorithm B was that its correctness could not (yet) be mathematically proven, and although it never returned wrong results, deeper examination of its inner state showed unexpected behavior (see topic 4.5.2). For this reason I created Algorithm C, which was a simplified version of version B. Unfortunately, the simplification done on the code made it incorrect, for specially constructed 3-SAT CNFs, Algorithm C fails, what I unfortunately discovered not before it was put online. Therefore, I combined the main algorithm (2-dimension Boolean array, as we'll see later on in this document) of version B and added version C's additional error testing and code parts that make the solver 'more robust'. The result of this merging, Algorithm D, is described within the article you are currently reading.

You can view all four algorithms in "pdf" format by visiting:

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

You can also download the C++ implementations of Algorithms B and D, which do in each case consist of extensively commented source code and a compiled binary for Windows and one for Linux within a "zip" file. The download-URL of that zip-file is:

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

Finally there's the Polynomial SAT Solver page on my private homepage. To view it, click on the following link:

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

Please prefer downloading material related to Algorithm D (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 article 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 index 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. If the CNF has no solution, we call it 'unsolvable' or 'unsatisfiable' (both expressions are used synonymously). Otherwise, if the CNF has one or more solutions, we call it 'solvable' or, interchangeably, 'satisfiable'.

One way to decide if the 3-SAT instance is solvable would be to try out all possible solutions. The matter with 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. Practical tests and theoretical considerations give reason to expect that the algorithm described in this document can reliably determine the 3-SAT CNF's solvability, even in the worst-case, after maximal n^{18} 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

To explain how the polynomial solving algorithm works, first a set of expressions and conventions will be declared. After that, we turn towards the actual algorithm.

4.1 General Definitions

The following definitions are crucial for the understanding of the algorithm. You will notice that I continuously capitalize many of them. This shall express the definitions are important and (some of them) can be seen as proper names exclusively used in this document and not in classic literature.

Definition 4.1.1 *The expression* **CNF** *does consistently stand, within this article, 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 index range of the CNF, i.e. the number of different literal indices. n will later be seen as the polynomial solver's input size for the definition of 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/and* [1] *"1" chars and exactly* $(n-3)$ *"-" chars.*

A classical, mathematical clause known from general literature can be converted into a Clause Line as follows: The Clause Line is initially a string $\{\texttt{-}\}^n$. Then we regard the three literals of the mathematic clause one after another. If a literal is negated, place "0" at the location within the Clause Line indicated by the literal index (the initial "-" chars are overridden). If the literal of the mathematical clause is not negated, act equally but place a "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, noted in Clause Line Notation. I decided to do this to get 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 defines 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. However, in the running text of this article I'll use the expressions 'char(s)' and 'digit(s)' synonymously.

[1] In this document, "or/and" does always mean that both "or" as well as "and" is applicable. In particular, "or/and" has nothing to do with the exclusive or.

Definition 4.1.3 *Two clauses C_1 and C_2 are said to be* **in conflict** *if C_1 has a "0" at a position where C_2 has a "1", 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. We use the notation $C_1 \not\equiv C_2$ to express C_1 and C_2 are in conflict, and we use the notation $C_1 \equiv C_2$ to express C_1 and C_2 are not in conflict.*

Mathematically we can describe 'two clauses J and K are in conflict' as follows: Let c_x be the x-th digit within an s-SAT clause C that is given in Clause Line Notation, what means that $c_x \in \{\texttt{0,1,-}\}$ with $1 \leq x \leq n$ and $s \geq 3$, n is the highest literal index in the SAT CNF to solve. Two clauses J and K, whose digits are accessible via j_x and k_x, are in conflict ($J \not\equiv K$) if $\exists x \mid (j_x = \texttt{0} \wedge k_x = \texttt{1}) \vee (j_x = \texttt{1} \wedge k_x = \texttt{0})$.

Definition 4.1.4 *We define the linguistic expression '***all 0/1 digits of** C **appear in** J **or/and** K*' mathematically as follows:*

The basic assumptions are: We regard some clause C, which is an s-SAT clause in Clause Line Notation, $s \geq 3$. When C is given in Clause Line Notation, then C is a string of n chars (synonymously called digits) $\in \{\texttt{0,1,-}\}$, whereby n is the highest literal index in the SAT CNF to solve. Each of the n digits of C shall be accessible via c_x, $1 \leq x \leq n$. The analogous applies to two further clauses J (digits accessible via j_x) and K (digits accessible via k_x). C, J and K must not be in conflict pair-wise ($C \equiv J$, $C \equiv K$, $J \equiv K$).

Finally we define 'all 0/1 digits of C appear in J or/and K' as follows:

$$\forall x \in \{1, 2, ..., n\} : (c_x = \texttt{-}) \vee (c_x = j_x) \vee (c_x = k_x)$$

This means that any digit of C is either '-', or it is '0' or '1' and then equal to the digit(s) at the same location (digit index) x within J, or K, or both.

Definition 4.1.5 *We define the linguistic expression '***a digit of** C **appears outside** J **and** K**, at position** p*' mathematically as follows:*

The basic assumptions as in the previous definition (4.1.4) are taken as granted.

If a digit c_p appears 'outside' J and K, then:

$$((c_p = \texttt{0}) \vee (c_p = \texttt{1})) \wedge (j_p = \texttt{-}) \wedge (k_p = \texttt{-})$$

Definition 4.1.6 *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 4.1.7 *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 can be replaced by "x", for notation purposes.*

Example:

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

Definition 4.1.8 *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 with "0" or/and "1" chars, the result is to be called* **Filled Clause Type(s)**.

Definition 4.1.9 *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-`, `00-0`, `00-1`, `01-0`, ..., `-111`}

Definition 4.1.10 *We say a Possible Solution S '***satisfies***' a Possible Clause C if S has at least one "0" or "1" at the same position as within C. Mathematically: $\exists x \in \{1, 2, ..., n\} \mid s_x = c_x$. Example: S =*`000000` *satisfies C =*`1-01--` *as both have a "0" at 3rd position.*

Proof: Char s_i is, for any $i \in \{1, 2, ..., n\}$, either "0" or "1". Those two states do, as defined in definition 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 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 4.1.11 *It is not possible to satisfy all $2^3 = 8$ Filled Clause Types with 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 with some Possible Solution, as can be tried out as well. □

Definition 4.1.12 *When referring to one or more Possible Clauses, we use* **capital letters** *to indicate clause strings ($C \in \{0,1,-\}^n$). Possible Clause indices are expressed by* **non-capital letters** *($c \in \{0, 1, ..., PossibleClauseNumber - 1\}$). Clause indices are mostly implemented in form of for-loop-variables, pointing to an element out of the set of Possible Clauses.*

Definition 4.1.13

- *The* **Clause Table**, *abbreviated 'CT', is a formula in disjunctive normal form.*
- *Every Clause Table has* 2^n *OR-ed terms and* $\binom{n}{3}$ *AND-ed Possible Clauses within each OR-term.* n *is the literal index range of the corresponding SAT CNF.*
- *The* j*-th OR-ed term of the CT shall be called the* j*-th* **CT Line**. *The accumulated set of all* i*-th Possible Clauses out of the AND-terms shall be called the* i*-th* **CT Column**.
- *Each column of the CT comprises the same Clause Type, but not necessarily the same clauses.*
- *The clauses in each CT Line are all pair-wise not in conflict.*
- *Each CT Line has an associated* $\{0,1\}^n$ *string called* **Underlying Solution** *and comprises all clauses that are not in conflict with that Underlying Solution. Each of the* 2^n *Underlying Solutions contributes one CT Line with all clauses not in conflict to that Underlying 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 SAT CNF to solve. If a clause exists in the SAT 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 Underlying 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.

To make you understand the construction of the Clause Table better, another example for $n = 5$:

(000-- ∧ 00-0- ∧ 00--0 ∧ 0-00- ∧ 0-0-0 ∧ 0--00 ∧ -000- ∧ -00-0 ∧ -0-00 ∧ --000)∨
(000-- ∧ 00-0- ∧ 00--1 ∧ 0-00- ∧ 0-0-1 ∧ 0--01 ∧ -000- ∧ -00-1 ∧ -0-01 ∧ --001)∨
(000-- ∧ 00-1- ∧ 00--0 ∧ 0-01- ∧ 0-0-0 ∧ 0--10 ∧ -001- ∧ -00-0 ∧ -0-10 ∧ --010)∨
(000-- ∧ 00-1- ∧ 00--1 ∧ 0-01- ∧ 0-0-1 ∧ 0--11 ∧ -001- ∧ -00-1 ∧ -0-11 ∧ --011)∨
(001-- ∧ 00-0- ∧ 00--0 ∧ 0-10- ∧ 0-1-0 ∧ 0--00 ∧ -010- ∧ -01-0 ∧ -0-00 ∧ --100)∨
(001-- ∧ 00-0- ∧ 00--1 ∧ 0-10- ∧ 0-1-1 ∧ 0--01 ∧ -010- ∧ -01-1 ∧ -0-01 ∧ --101)∨
(001-- ∧ 00-1- ∧ 00--0 ∧ 0-11- ∧ 0-1-0 ∧ 0--10 ∧ -011- ∧ -01-0 ∧ -0-10 ∧ --110)∨
(001-- ∧ 00-1- ∧ 00--1 ∧ 0-11- ∧ 0-1-1 ∧ 0--11 ∧ -011- ∧ -01-1 ∧ -0-11 ∧ --111)∨
(010-- ∧ 01-0- ∧ 01--0 ∧ 0-00- ∧ 0-0-0 ∧ 0--00 ∧ -100- ∧ -10-0 ∧ -1-00 ∧ --000)∨
(010-- ∧ 01-0- ∧ 01--1 ∧ 0-00- ∧ 0-0-1 ∧ 0--01 ∧ -100- ∧ -10-1 ∧ -1-01 ∧ --001)∨
(010-- ∧ 01-1- ∧ 01--0 ∧ 0-01- ∧ 0-0-0 ∧ 0--10 ∧ -101- ∧ -10-0 ∧ -1-10 ∧ --010)∨
(010-- ∧ 01-1- ∧ 01--1 ∧ 0-01- ∧ 0-0-1 ∧ 0--11 ∧ -101- ∧ -10-1 ∧ -1-11 ∧ --011)∨
(011-- ∧ 01-0- ∧ 01--0 ∧ 0-10- ∧ 0-1-0 ∧ 0--00 ∧ -110- ∧ -11-0 ∧ -1-00 ∧ --100)∨
(011-- ∧ 01-0- ∧ 01--1 ∧ 0-10- ∧ 0-1-1 ∧ 0--01 ∧ -110- ∧ -11-1 ∧ -1-01 ∧ --101)∨
(011-- ∧ 01-1- ∧ 01--0 ∧ 0-11- ∧ 0-1-0 ∧ 0--10 ∧ -111- ∧ -11-0 ∧ -1-10 ∧ --110)∨
(011-- ∧ 01-1- ∧ 01--1 ∧ 0-11- ∧ 0-1-1 ∧ 0--11 ∧ -111- ∧ -11-1 ∧ -1-11 ∧ --111)∨
(100-- ∧ 10-0- ∧ 10--0 ∧ 1-00- ∧ 1-0-0 ∧ 1--00 ∧ -000- ∧ -00-0 ∧ -0-00 ∧ --000)∨
(100-- ∧ 10-0- ∧ 10--1 ∧ 1-00- ∧ 1-0-1 ∧ 1--01 ∧ -000- ∧ -00-1 ∧ -0-01 ∧ --001)∨
(100-- ∧ 10-1- ∧ 10--0 ∧ 1-01- ∧ 1-0-0 ∧ 1--10 ∧ -001- ∧ -00-0 ∧ -0-10 ∧ --010)∨
(100-- ∧ 10-1- ∧ 10--1 ∧ 1-01- ∧ 1-0-1 ∧ 1--11 ∧ -001- ∧ -00-1 ∧ -0-11 ∧ --011)∨
(101-- ∧ 10-0- ∧ 10--0 ∧ 1-10- ∧ 1-1-0 ∧ 1--00 ∧ -010- ∧ -01-0 ∧ -0-00 ∧ --100)∨
(101-- ∧ 10-0- ∧ 10--1 ∧ 1-10- ∧ 1-1-1 ∧ 1--01 ∧ -010- ∧ -01-1 ∧ -0-01 ∧ --101)∨
(101-- ∧ 10-1- ∧ 10--0 ∧ 1-11- ∧ 1-1-0 ∧ 1--10 ∧ -011- ∧ -01-0 ∧ -0-10 ∧ --110)∨
(101-- ∧ 10-1- ∧ 10--1 ∧ 1-11- ∧ 1-1-1 ∧ 1--11 ∧ -011- ∧ -01-1 ∧ -0-11 ∧ --111)∨
(110-- ∧ 11-0- ∧ 11--0 ∧ 1-00- ∧ 1-0-0 ∧ 1--00 ∧ -100- ∧ -10-0 ∧ -1-00 ∧ --000)∨
(110-- ∧ 11-0- ∧ 11--1 ∧ 1-00- ∧ 1-0-1 ∧ 1--01 ∧ -100- ∧ -10-1 ∧ -1-01 ∧ --001)∨
(110-- ∧ 11-1- ∧ 11--0 ∧ 1-01- ∧ 1-0-0 ∧ 1--10 ∧ -101- ∧ -10-0 ∧ -1-10 ∧ --010)∨
(110-- ∧ 11-1- ∧ 11--1 ∧ 1-01- ∧ 1-0-1 ∧ 1--11 ∧ -101- ∧ -10-1 ∧ -1-11 ∧ --011)∨
(111-- ∧ 11-0- ∧ 11--0 ∧ 1-10- ∧ 1-1-0 ∧ 1--00 ∧ -110- ∧ -11-0 ∧ -1-00 ∧ --100)∨
(111-- ∧ 11-0- ∧ 11--1 ∧ 1-10- ∧ 1-1-1 ∧ 1--01 ∧ -110- ∧ -11-1 ∧ -1-01 ∧ --101)∨
(111-- ∧ 11-1- ∧ 11--0 ∧ 1-11- ∧ 1-1-0 ∧ 1--10 ∧ -111- ∧ -11-0 ∧ -1-10 ∧ --110)∨
(111-- ∧ 11-1- ∧ 11--1 ∧ 1-11- ∧ 1-1-1 ∧ 1--11 ∧ -111- ∧ -11-1 ∧ -1-11 ∧ --111)

The order of the CT Columns and CT Lines can be chosen arbitrarily. In my solver implementation I use a kind of lexicographical order. But: If you wanted to change the CT Column order, you would have to exchange whole columns of course, and not just single clauses. This applies to exchanging CT Lines as well.

Observation 4.1.14 *A SAT 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* [2].

Proof: Within one CT Column, there are listed all $2^3 = 8$ possible Filled Clause Types. According to observation 4.1.11, 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 all replaced by true or are all replaced by false. It is not possible that some are replaced by true and others by false. □

[2] For education, please look up the mentioned absent PCs in the CT example of definition 4.1.13.

To finalize this topic, I want to give a last remark: As a help for the reader, the downloadable zip-file that contains the implementation of the polynomial solver contains, besides the actual solver code, also the code and the Windows and Linux binaries of a further console program which generates and displays the Clause Table DNF for a chooseable n (literal index range) between 3 and 8. You can find the program in the directory "Algorithm D ([...])\Tests - not part of the actual polynomial solver\ClauseTableDisplayer\(Release\)". That additional application is meant to be a help for the reader of this article to better understand the structure of the Clause Table. I recommend you to have a look at the program and run it after having read the rest of this document. For the download link of the zip-file please see the previous topic 'History of Polynomial Algorithm'.

4.2 The Polynomial 3-SAT-Algorithm

The main idea behind the polynomial SAT-solving algorithm is to detect if all Possible Clauses of at least one CT Line are all absent from the SAT CNF to solve. If all those Possible Clauses should be absent, the CNF is solvable, as mentioned in observation 4.1.14. Please notice: Although the Clause Table has exponentially many CT Lines, the upcoming algorithm can evaluate it in polynomial time and space. The reason therefore is that the polynomial algorithm regards repeating patterns, namely quadruple of clauses, as we will later see, not as often as they appear in the exponentially-sized Clause Table.

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

Definition 4.2.1 *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.*

Observation 4.2.2 *If there is at least one Active CT Line, the CNF is solvable.*

Proof: Any Active CT Line fulfills the conditions of observation 4.1.14 and definition 4.2.1, what implies the CNF is solvable. □

Definition 4.2.3 *An* **Active Clause Tuple** *shall be a tuple, i.e. an ordered list of two Possible Clauses, whereby both of those clauses appear in an Active CT Line.*

Definition 4.2.4 *We say a tuple of clauses* (J, K) **contains** *one further clause* I *if:*

- J and K are not in conflict: $J \equiv K$.
- I is not in conflict with J or K: $(I \equiv J) \wedge (I \equiv K)$.
- All `0/1` digits of I appear in J or/and K: $\forall x \in \{1, 2, ..., n\} : (i_x = \texttt{-}) \vee (i_x = j_x) \vee (i_x = k_x)$.

It is valid that $(J = I)$, or $(K = I)$, or $(J = K = I)$, i.e. clauses are equal, as long as I is still contained within J or/and K.

Example: `J=000---, K=00-0--` contain e.g. `I=0-00--`, as well as e.g. `I=-000--`.

Observation 4.2.5 *If some clause* I *is contained within a clause tuple* (J, K)*, then* I *is the only clause that appears, in* I*'s specific CT Column, in any CT Line including* J *and* K*.*

Proof: all clauses of an arbitrarily chosen CT Line are not in conflict pair-wise. When I is contained within (J, K), then *all* of I's `0/1` digits do either appear in J, or in K, or in both. This means that any other clause I_{other} would have a conflict with J or/and K, because I_{other} has at least one `0` where J or/and K has `1`, or vice versa. Thus any I_{other} different from I cannot appear in the CT Line(s) including both J and K. □

Example: `I=0--00-` is contained within `J=000---, K=---000`. As soon as we negate one digit of I, there is a conflict. Example: I_{other}`=1--00-` is *not* contained within `J=000---, K=---000`, because I_{other} has a conflict with J at the first digit.

Definition 4.2.6 *We say a tuple of clauses* (J, K) **contains** *two further clauses* I, H *when the following requirements are fulfilled:*

- J and K are not in conflict: $J \equiv K$.
- I is not in conflict with J or K: $(I \equiv J) \wedge (I \equiv K)$.
- H is not in conflict with J or K: $(H \equiv J) \wedge (H \equiv K)$.
- I and H have exactly one conflict at a well-defined position p within I and H. The position p is in this article counted 1-based, i.e. the range of p is $\{1, 2, ..., n\}$, whereby n is the SAT CNF's literal index range, respectively 'DigitNumber' in the solver implementation. Example: `I=000---` and `H=-010--` have one conflict at position $p = 3$.
- All `0` or/and `1` digits of I and H appear in J or/and K, except I's and H's conflicting `0` and `1` digits at position p, there J and K must both have a '`-`' digit.

Examples of tuples (J, K) containing I and H:

In the first example I and H have a conflict at position $p = 5$, and I and H are contained within (J, K):

`J=000---, K=00-0--, I=0-0-0-, H=0-0-1-`

In the second example, I and H have one conflict, and have their further `0` digits at different positions. Also this case is valid as long as I and H are, except the digits at p, fully contained within (J, K):

`J=000---, K=00---0, I=00--0-, H=--0-10`

You might have correctly noticed that the definition of 'contained' is ambiguous, once one clause I is contained within (J, K) and in the second definition two clauses I and H are contained within (J, K). Only the second case allows placing one digit of I and one digit of H outside J and K, both at position p.

Definition 4.2.7 *We say a Clause Table Line* **includes** *a set of clauses* $C_1, C_2, ...$ *if all those clauses appear in the Clause Table Line.*

4.2.1 Code Listing

Code Listing 4.2.8 *It follows a simplified version of the original implementation (for a downloadable, compiled and optimized version, 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 subsequent to the code listing. Please notice that the full code of helper procedures appears in this article's appendix, in the following listing only the helper procedures' prototypes are given.*

```
// ******************************************************************************************
// PLEASE NOTE: C notation of mathematic operators:
// '&&' = AND, '||' = OR, '==' = check for equality ('=' is an assignment),
// '!' = negation (true becomes false or false becomes true).
// ******************************************************************************************
```

```
// global variables used by the solver code:

int        DigitNumber;
int        ClauseNumber;
char*      CNF;

int        SAT_TYPE = 3; // 3-SAT or 2-SAT (the downloadable solver implementation can handle both)

#define    DIGIT_NUMBER_MAX                    15
#define    POSSIBLE_CLAUSE_NUMBER_MAX          3400 // DIGIT_NUMBER_MAX^3 plus some safety

int        PossibleClauseNumber;
char       PossibleClauses[POSSIBLE_CLAUSE_NUMBER_MAX][DIGIT_NUMBER_MAX + 1];


// prototypes of helper functions, see this article's appendix for complete code:

bool IsEqual(char* Clause1, char* Clause2);
bool IsEqual(int ClauseIndex1, int ClauseIndex2);
bool IsInConflict(char* Clause1, char* Clause2);
bool IsInConflict(int ClauseIndex1, int ClauseIndex2);
bool ExistsInCNF(char* Clause);
bool ExistsInCNF(int ClauseIndex);
void PossibleClauses_Create();
bool Does_ih_HaveOneConflict(char* I, char* H);
bool Does_ih_HaveOneConflict(int i, int h);
bool Does_jk_Contain_ih(char* I, char* H, char* J, char* K);


// the main part of the polynomial 3-SAT solving code:

bool DoesCNFHaveASolution(
    int     DigitNumberPassed,
    int     ClauseNumberPassed,
    char*   CNFPassed)
{
    // set global variables used by the helper functions
    DigitNumber = DigitNumberPassed;
    ClauseNumber = ClauseNumberPassed;
    CNF = CNFPassed;

    // treat CNFs whose ClauseTable does not have at least 3 columns specially
    if (SAT_TYPE == 3 && DigitNumber == 3)
    {
        return
            !ExistsInCNF("000") ||
            !ExistsInCNF("001") ||
            !ExistsInCNF("010") ||
            !ExistsInCNF("011") ||
            !ExistsInCNF("100") ||
            !ExistsInCNF("101") ||
            !ExistsInCNF("110") ||
            !ExistsInCNF("111");
    }
```

```
// *** PREPARATION ***

PossibleClauses_Create();

// initialize ActiveClauseTuple
static bool ActiveClauseTuple[POSSIBLE_CLAUSE_NUMBER_MAX][POSSIBLE_CLAUSE_NUMBER_MAX];
for (int j = 0; j < PossibleClauseNumber; j ++)
    for (int k = 0; k < PossibleClauseNumber; k ++)
        ActiveClauseTuple[j][k] = !IsInConflict(j, k) && !ExistsInCNF(j) && !ExistsInCNF(k);

// *** ENTER SOLVER MAIN LOOPS ***

bool ChangesExisting = true; // if ActiveClauseTuple[][] content has changed (a 'dirty-flag')

while (ChangesExisting)
{
    ChangesExisting = false; // reset

    for (int i = 0; i < PossibleClauseNumber; i ++) // the "main-i-loop"
    {
        for (int h = 0; h < PossibleClauseNumber; h ++) // the "main-h-loop"
        {
            if ((Does_ih_HaveOneConflict(i, h)) || (i == h)) // i == h IS allowed!!!
            {
                for (int j = 0; j < PossibleClauseNumber; j ++)
                {
                    if (IsInConflict(i, j) ||
                        IsInConflict(h, j))
                        continue;

                    for (int k = 0; k < PossibleClauseNumber; k ++)
                    {
                        if (IsInConflict(i, k) ||
                            IsInConflict(h, k) ||
                            IsInConflict(j, k))
                            continue;

                        if ((!ActiveClauseTuple[i][j] || !ActiveClauseTuple[i][k]) &&
                            (!ActiveClauseTuple[h][j] || !ActiveClauseTuple[h][k]))
                        {
                            if (ActiveClauseTuple[j][k])
                            {
                                if (Does_jk_Contain_ih(
                                    PossibleClauses[i],
                                    PossibleClauses[h],
                                    PossibleClauses[j],
                                    PossibleClauses[k]))
                                {
                                    ActiveClauseTuple[j][k] = false;
                                    ChangesExisting = true;
                                }
                            }
                        }
                    }
                }
            }
```

```
            }
        }
    }

    // *** DETERMINE SAT OR UNSAT FROM ACTIVECLAUSETUPLE ARRAY ***

    bool SAT = false; // UNSAT - CNF has no solution

    for (int i = 0; i < PossibleClauseNumber; i ++)
    {
        for (int j = 0; j < PossibleClauseNumber; j ++)
        {
            if (ActiveClauseTuple[i][j])
            {
                // printf("still true ActiveClauseTuple: %s,%s\n", PossibleClauses[i], PossibleClauses[j])
                SAT = true; // SAT - CNF has some solution (we do not know which one)
            }
        }
    }

    // *** OPTIONALLY DO CONSISTENCY CHECKS ***

#ifdef DO_CONSISTENCY_CHECKS
    // for the code of the following debugging procedure, please
    // view the downloadable implementation (code not listed here
    // due to limited space, and because DoConsistencyChecks() is
    // not required for the actual solving process):
    DoConsistencyChecks(DigitNumber, ClauseNumber, ActiveClauseTuple);
#endif

    return SAT;
}
```

4.2.2 Implementation Details

In this section I will give more information on key-parts of the code you just saw in the Code Listing.

The polynomial solver contains a 2-dimensional Boolean array called `ActiveClauseTuple[J][K]`. This array tells if a tuple of clauses (J, K) from the (imagined) Clause Table is still in an Active CT Line, or if that tuple is known to be exclusively within CT Lines that do all include at least one false-valued clause. Recall that a clause is trodden as false if it appears in the SAT CNF to solve. If a clause is not present in the SAT CNF, that clause is imagined to be true.

At the beginning of the solving process, the array `ActiveClauseTuple[J][K]` is initialized. Only those array elements will be true that correspond to two clauses J, K which are not in conflict and that are both absent from the SAT CNF. Of course, this initialization does not yet reliably tell if a clause tuple (J, K) is within an Active CT Line, because a CT Line is already *not* an Active CT Line if it includes one false clause. The initialization regards only two clauses from a CT Line at a time, and because a CT Line has always more than two clauses, it is likely to happen that also tuples from a non-Active CT Line get an initial `ActiveClauseTuple[J][K]`-value of true.

The state the polynomial solver must reach is the following one: at the end of the solving process, exactly those tuples (J, K) that appear in at least one Active CT Line must have a 'true' `ActiveClauseTuple[J][K]`-value. By contrast, those tuples (J, K) that appear only in non-Active CT Lines must have a 'false' `ActiveClauseTuple[J][K]`-value. This means that, initially, there are too many `ActiveClauseTuple`-values 'true'. What the polynomial solver now does is that it regards once two and once four tuples to 'conclude' that a third, respectively fifth tuple *must* appear exclusively in non-Active CT Lines as well.

The conclusion which tuples must be in non-Active CT Lines is done according to the following two rules:

"Rule 1": Any tuple (J, K) is disabled if there is a third clause I, whereby:

- (J, K) must contain I (see definition 4.2.4). This implies that I is not in conflict with J or K.
- Furthermore it is requested that:
  (`ActiveClauseTuple[I][J] = false` or/and `ActiveClauseTuple[I][K] = false`).

"Rule 2": Any tuple (J, K) is disabled if there is a third clause I and a fourth clause H, whereby:

- (J, K) must contain I and H (see definition 4.2.6). This implies that I is not in conflict with J or K, and also H is not in conflict with J or K. Notice that I and H have *always exactly one* conflict at some position p, as also defined in the just mentioned definition. The conflict digits (once `0` and once `1`) lie 'outside' J and K, meaning J and K both have a '`-`' char at position p.
- Furthermore it is requested that:
  (`ActiveClauseTuple[I][J] = false` or/and `ActiveClauseTuple[I][K] = false`) and
  (`ActiveClauseTuple[H][J] = false` or/and `ActiveClauseTuple[H][K] = false`).

These two rules are applied until all `ActiveClauseTuple`-values have been checked and none of them has been changed any more. This repetition is implemented by a while-loop, which is called the 'while-ChangesExisting-loop'. More details about this loop are to come later on.

On first sight, it is probably not clear why the repetitive usage of those two rules always leads to the final state that only tuples from Active CT Lines are still true. But the purpose of the first rule, "Rule 1", can be understood easily: We disable the tuple (J, K) if there is a contained I, and `ActiveClauseTuple[I][J] = false` or `ActiveClauseTuple[I][K] = false`. What is important to realize is that if I is contained within (J, K), then I is the only clause that appears in I's CT Column within *all* CT Lines including J, K (see observation 4.2.5). This means that if I is false, there is at least one false clause (namely I) in *all* CT Lines including J, K. So it is guaranteed to be valid to set the tuple (J, K) to false.

Figuratively one can understand "Rule 1" like this: Take clauses J and K and write the `0/1` digits of both clauses into a temporary third clause L. L can have, for this consideration, more than three `0/1` digits. If now a false clause I is found whose `0/1` digits do all appear 'within' L, then `ActiveClauseTuple[J][K]` is set to false by Rule 1.

Example:

`J=000---, K=--000- => L=00000-`

So, any of the following false I's would set `ActiveClauseTuple[J][K] :=` [3] `false`:

`I` $\in$ {`000---, 00-0--, 00--0-, 0-00--, 0-0-0-, 0--00-, -000--, -00-0-, -0-00-, --000-`}

Understanding the purpose and influence of "Rule 2" needs more detailed explanations, which will soon follow in the upcoming Proof of Correctness. Summarized, "Rule 2" allows setting `ActiveClauseTuple[J][K] := false` through false clauses I and H which have, in contrast to the requirements of Rule 1, not all its `0` and `1` digits 'within' J and K.

[3] 'A := B' means variable A gets assigned the value of B.

4.3 Proof of Correctness

The proof of correctness is split up into two parts. First, we'll see that the polynomial solver does never say 'unsatisfiable' when the CNF is satisfiable. Afterwards we examine the opposite case, i.e. we show the solver does never detect 'satisfiable' although the CNF is UNSAT [4].

Within the proof we'll make use of the following linguistic terms:

Definition 4.3.1 *The expression* **'disabling a tuple** (J, K)**'** *shall mean setting* `ActiveClauseTuple[J][K]` *to false. A* **disabled tuple** (J, K) *is, by implication, a tuple of clauses* J, K *for which* `ActiveClauseTuple[J][K]` *is (already) false. An* **enabled tuple** *is one whose corresponding* `ActiveClauseTuple` *value is (still) true.*

4.3.1 Why Satisfiable Detection is Reliable

Observation 4.3.2 *The polynomial algorithm shown in code listing 4.2.8 detects reliably when the SAT CNF to solve is satisfiable, that means when the CNF does have a solution.*

Proof: When the SAT CNF is solvable, its corresponding Clause Table has at least one CT Line that includes exclusively true clauses.

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.

We realize that all tuples (J, K) with J, K out of the true CT Line are initialized to `ActiveClauseTuple[J][K] := true`, because J and K are not in conflict (the property of any clause tuple from one and the same CT Line), and are imagined to be true because they do not appear in the SAT CNF ("`!ExistsInCNF(J) && !ExistsInCNF(K)`", see code listing 4.2.8).

During the polynomial solving process' main part, which is the i/h/j/k loop iterations within the while-ChangesExisting-loop, the solver tries, for each clause tuple (J, K), to find clauses I, or I and H that are 'contained' by (J, K). Please recall definitions 4.2.4 and 4.2.6. So `ActiveClauseTuple[J][K]` is set to false if there is either at least one 'contained' I for which `ActiveClauseTuple[I][J] = false` or `ActiveClauseTuple[I][K] = false`. But such an I will not be found for all tuples out of the true CT Line. This is because I must be contained by (J, K). Such a clause I is the only clause that appears in any CT Line including (J, K), as shown in observation 4.2.5. But those I are all imagined as true, defined by the starting conditions of this proof. When all I's are imagined to be true, then also all `ActiveClauseTuple[I][J]` and `ActiveClauseTuple[I][K]` are true. This does eventually also apply to clause H: Depending on how many CT Lines include only true clauses, we will never find such a just mentioned I, or we do never find some H, or eventually both, if many CT Lines are completely true.

The algorithm will finally return 'satisfiable' as at least all `ActiveClauseTuple[J][K]` are true for all clauses out of the Active CT Line. □

[4] 'UNSAT' is, in this document and the solver code, the abbreviation for 'unsatisfiable'.

4.3.2 Why UNSAT Detection is Reliable

The initial situation of this proof is that in each CT Line, there is at least one clause stated false, because it is existing in the SAT CNF to solve. The CT Column containing the false clause was, for each CT Line, selected absolutely arbitrarily.

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 includes at least one false.

Observation 4.3.3 *The polynomial algorithm shown in code listing 4.2.8 detects reliably when the CNF to solve is unsatisfiable, that means when the CNF has no solution.*

Proof: The goal in this proof is to show that the polynomial solver will never say the CNF would be satisfiable although it is in reality unsatisfiable. Please keep in mind that there is, as just told, at least one false clause in each CT Line. Because the proof is expansive, it is subdivided into several sections, beginning with some notation conventions, followed by the derivation of the actual proof.

4.3.2.1 General Definitions for Proof

Definition 4.3.4 *An* **initially false clause** *shall be a Possible Clause that has, in the context of the Clause Table theory, the value false because it appears in the SAT CNF to solve. An initially false clause might cause the disabling of clause tuples from the very beginning of the polynomial solving process on, in contrast to false-switched clause tuples which might disable further clause tuples not before having been disabled themselves.*

Definition 4.3.5 *We use the notation* $\mathbf{C_3 = C_1 \oplus C_2}$ *to denote the* **overlay** *of two clauses* C_1 *and* C_2 *to a third clause* C_3*. This operation can be applied to any set of two clauses if these clauses are not in conflict.* C_3 *is initially a* $\{\texttt{-}\}^n$ *string into which all* `0` *and* `1` *digits of* C_1 *and* C_2 *are copied, each* `0/1` *digit to the location where it appears in* C_1 *respectively* C_2*.*

Examples:

`0000-- = 000--- ⊕ -000--`

`-10000 = -100-- ⊕ ---000`

`111--- = 111--- ⊕ 111---`

It is possible and valid that C_3 might have more `0/1` digits than C_1 or C_2.

Definition 4.3.6 *Let the notation $\tau(\mathbf{C})$ describe the whole set of 3-SAT clauses which have been built by selecting exactly three `0` or/and `1` digits out of an s-SAT clause C, whereby $s \geq 3$. Each possible selection appears exactly once in τ* [5].

Example:

$$\tau(\texttt{00001-}) = \{\texttt{000---, 00-0--, 00--1-, 0-00--, 0-0-1-, 0--01-, -000--, -00-1-, -0-01-, --001-}\}$$

Definition 4.3.7 *We say an Underlying Solution US* **contains** *some 3-SAT clause C if $C \in \tau(US)$.*

Definition 4.3.8 *The expression $C_{out} = \mathbf{C_{in}} + \mathbf{0}(\mathbf{C_{mask}}, \mathbf{p})$ has the following meaning: The clause C_{out} is a copy of clause C_{in}, with one additional `0` digit at the position where clause C_{mask} has its p-th '-' char. The same applies to $C_{out} = \mathbf{C_{in}} + \mathbf{1}(\mathbf{C_{mask}}, \mathbf{p})$, with the difference that one additional `1` digit is added. It is also defined that C_{in} and C_{out} can be sets of clauses, then the described operation is used on each of the clauses from C_{in}.*

Examples:
C_{in} is a single clause:

```
000-0- = (0-0-0- + 0(0-0-0-,1))     0-000- = (0-0-0- + 0(0-0-0-,2))
010-0- = (0-0-0- + 1(0-0-0-,1))     0-010- = (0-0-0- + 1(0-0-0-,2))
0-000- = (0-0-0- + 0(000---,1))     0-0-0- = (0-0-0- + 0(000---,2))
```

C_{in} is a set of clauses:

```
{0-000-,0-00-0,0--0-0} = ({0-0-0-,0-0--0,0--0-0} + 0(000---,1))
```

Definition 4.3.9 *Let $S(C)_{out} = \mathbf{S(C)_{in}} + \mathbf{01}(\mathbf{C_{mask}}, \mathbf{p})$ mean that we take each clause from the input set $S(C)_{in}$, create two copies of the taken clause, and add one `0` digit at position p to the first copy and one `1` digit at position p to the second copy. p is, specially in this definition, the p-th position where C_{mask} has a '-' char. The two resulting clauses are added to the output set $S(C)_{out}$. $S(C)_{out}$ has twice as many elements as $S(C)_{in}$. If the elements from $S(C)_{in}$ are l-SAT clauses, then the elements in $S(C)_{out}$ are (l+1)-SAT clauses. It is requested that the clauses in $S(C)_{in}$ do all have at least one '-' char left that can be replaced by `0` and `1`.*

Examples:

```
{0000-,0001-} = ({000--} + 01(000--,1))
```

A concatenation shall also be valid:

```
{00000,00001,00010,00011} = (({000--} + 01(000--,1))+01(000--,2))
```

Definition 4.3.10 *The expression 'S(C) =* **all possible 0/1 combinations around a clause tuple** *(A, B)' shall mean the set of clauses $S(C)$ has been created by the repetitive usage of the $+01()$ operator: $S(C) = (((((A \oplus B) + 01(A \oplus B, 1)) + 01(A \oplus B, 2)) + ...) + 01(A \oplus B, \phi))$. ϕ is the count of '-' digits within $A \oplus B$.*

Example:

```
{000000,100000,000001,100001} = 'all possible 0/1 combinations around (-000--,-0-00-)'
```

Definition 4.3.11 *We define the function $\mathbf{F(S)}$ which takes a set S of 3-SAT clauses. F* [6] *returns true if any of the clauses in S is false, meaning if at least one clause appears in the SAT CNF to solve. If there is no false clause in S, the function F returns false.*

[5] tau like 'three out of'.
[6] F like "any false clause contained?"

Observation 4.3.12 *We can describe 'one false clause in all CT Lines including clauses A and B' as follows:*

$\alpha =$

$$F(\tau(((((A \oplus B) + 0(L,1)) + 0(L,2)) + ...) + 0(L,\phi)))\wedge$$
$$F(\tau(((((A \oplus B) + 0(L,1)) + 0(L,2)) + ...) + 1(L,\phi)))\wedge$$
$$F(\tau(((((A \oplus B) + 0(L,1)) + 1(L,2)) + ...) + 0(L,\phi)))\wedge$$
$$F(\tau(((((A \oplus B) + 0(L,1)) + 1(L,2)) + ...) + 1(L,\phi)))\wedge$$
$$F(\tau(((((A \oplus B) + 1(L,1)) + 0(L,2)) + ...) + 0(L,\phi)))\wedge$$
$$F(\tau(((((A \oplus B) + 1(L,1)) + 0(L,2)) + ...) + 1(L,\phi)))\wedge$$
$$F(\tau(((((A \oplus B) + 1(L,1)) + 1(L,2)) + ...) + 0(L,\phi)))\wedge$$
$$F(\tau(((((A \oplus B) + 1(L,1)) + 1(L,2)) + ...) + 1(L,\phi)))$$

$L = (A \oplus B), \phi =$ `number of '-' chars within L`

Proof: the just shown formula α creates the Underlying Solution strings $US_x \in \{0,1\}^n$ of each CT Line including A and B, by 'adding all 0/1 combinations around A and B', as defined in definition 4.3.10. When applying the τ operator on each of those Underlying Solution strings, we get the set of all 3-SAT clauses from the CT Line corresponding to the current Underlying Solution. Finally, the $F()$ function, as recently defined, checks if at least one clause from each of those sets, i.e. from each of the regarded CT Lines, is initially false. □

4.3.2.2 Tuple Disabling at Initialization Time

With the definitions of the last topic it is now possible to set up the actual proof that the polynomial solver's UNSAT detection is reliable, i.e. that the solver, as shown in code listing 4.2.8 and as provided for downloading, detects every unsolvable SAT CNF as such. If the SAT CNF to be processed is unsolvable, then absolutely all clause tuples must be disabled by the polynomial solver. Therefore we regard a general clause tuple (J, K) and show it gets disabled when the corresponding SAT CNF is unsolvable. 'Getting disabled' means that at the very end of the polynomial solving process, `ActiveClauseTuple[J][K]` is false. J and K can be any clauses from the set of Possible Clauses (see definition 4.1.9) that are not in conflict with each other. We will identify four different main possibilities to disable the tuple (J, K), "at tuple initialization", "by Rule 1", "by Rule 2, sub case a)" and "by Rule 2, sub case b)".

As first announced possibility, we realize (J, K) is disabled if either J, or K, or both are initially false clauses. This disabling of (J, K) is done at initialization time, by the code lines:

```
for (int j = 0; j < PossibleClauseNumber; j ++)
    for (int k = 0; k < PossibleClauseNumber; k ++)
        ActiveClauseTuple[j][k] = !IsInConflict(j, k) && !ExistsInCNF(j) && !ExistsInCNF(k);
```

Recall a clause I is seen as initially false when it does appear in the 3-SAT CNF that is to be solved. So "`!ExistsInCNF(j)`" or/and "`!ExistsInCNF(k)`" is not fulfilled when $J = I$, or/and $K = I$, and `ActiveClauseTuple[J][K]` will be initialized to false. (Because code snippets are cited from the Code Listing 4.2.8, they contain non-capital letters representing clauses, but j and J, as well as k and K each represent the same clause.)

4.3.2.3 Tuple Disabling by Rule 1

If neither J nor K are initially false clauses, the tuple (J, K) can, as second possibility, be disabled by "**Rule 1**". Rule 1 disables any clause tuple (J, K) if there is *one* initially false clause I *contained* within (J, K). Please recall that "I is contained within the tuple (J, K)" means that all `0` or/and `1` digits of I appear at the same location (same digit- respectively literal index) in J, or K, or both. This has been defined in 4.2.4. It is valid to disable any tuple (J, K) if a contained clause I is false, because then I is the only clause that appears, in I's specific CT

Column, in *any* CT Line including J and K, what we know since observation 4.2.5. So we can be sure that in each CT Line including J and K and I, there is at least one false clause, namely I. And we also said we can disable any tuple (J, K) that appears exclusively in CT Lines that include at least one false clause, what is the case now.

Mathematically we can express "I is contained within [e.g.] `(J=000-----,K=---000--)`" through:

$$I \in \tau(J \oplus K)$$

$$\Leftrightarrow$$

$$I \in \tau(\texttt{000-----} \oplus \texttt{---000--})$$

This means that when we overlay clauses J and K ($J \oplus K$), then I is equal to one clause resulting from the usage of the τ operator on the overlay ($J \oplus K$).

For our example, the set of I contained in (J, K) and not equal to J or K is: $\tau(J \oplus K) = \tau(\texttt{000-----} \oplus \texttt{---000--})$ = {`00-0----`, `00--0---`, `00---0--`, `0-00----`, `0-0-0---`, `0-0--0--`, `0--00---`, `0--0-0--`, `0---00--`, `-000----`, `-00-0---`, `-00--0--`, `-0-00---`, `-0-0-0--`, `-0--00--`, `--000---`, `--00-0--`, `--0-00--`}. If any of these just shown clauses is false, because it appears in the SAT CNF to be solved, then `ActiveClauseTuple[J][K]` is set to false by Rule 1.

Up to this point, we collected information what Rule 1 shall theoretically do. Now, in the following paragraphs we will see how Rule 1 is practically implemented within the code shown in the 'Code Listing' (4.2.8), and also in the downloadable example implementation.

So, how does the provided code assign `ActiveClaueTuple[J][K] := false` when there is a contained, initially false clause I? First, we notice that because I is an initially false clause, all `ActiveClauseTuple[I][X]` and all `ActiveClauseTuple[X][I]` values will be initialized to false for any clause X, because either "`!ExistsInCNF(I)`" or "`!IsInConflict(I, X)`" is not given. This circumstance has just been explained in the previous topic "Tuple Disabling at Initialization Time".

Furthermore, within the solver implementation, as shown in code listing 4.2.8, it is allowed that $I = H$ (I is equal to H). Because the i- and h- main loops will point, within every complete run, exactly once to each thinkable I/H tuple, it is for sure that the both conditions I is equal to H, and I and H are contained within (J, K) will be fulfilled once. Then the helper function `Does_jk_Contain_ih()` will return true as the current I and H are contained in (J, K) in the way requested by 4.2.6, or, what is the currently more important case, as $I = H$ and I and H are contained in (J, K) as requested by 4.2.4. For these equal I and H, which are contained within (J, K), we regard the crucial excerpt from the polynomial solver's source code (4.2.8) and see that the condition:

```
if ((!ActiveClauseTuple[I][J] || !ActiveClauseTuple[I][K]) &&
    (!ActiveClauseTuple[H][J] || !ActiveClauseTuple[H][K]))
{
    if (Does_jk_Contain_ih(I, H, J, K)) [7]
    {
        ActiveClauseTuple[J][K] = false; // this code line disables tuple (J,K)
    }
}
```

will be true. This is because (`!ActiveClauseTuple[I][J] || !ActiveClauseTuple[I][K]`) && (`!ActiveClauseTuple[H(=I)][J] || !ActiveClauseTuple[H(=I)][K]`) is given, due to the fact that I and H are initially false clauses and thus *any* `ActiveClauseTuple[I][X]` and *any* `ActiveClauseTuple[H][X]` is initialized to false for an arbitrary clause X.

Concluded, we saw that also some initially false clause I that is contained within (J, K) will disable (J, K).

[7] While the function name states J and K first, those two clauses are passed as 3rd and 4th argument. This has historical reasons, please don't get confused because of that.

4.3.2.4 Tuple Disabling by Rule 2

The following (long) topic will give information about how the polynomial solver can decide on disabling some tuple which does not consist of initially false clauses, and which does not completely contain one initially false clause.

The goal is the following: We will show that the polynomial solver disables any tuple (J, K) if there is at least one initially false clause contained in *each* Possible Solution (an n-SAT clause) built by adding 'all 0/1 combinations around (J, K)'.

An example shall illustrate that: if e.g. `J=000-----` and `K=---000--`, then we need to show that (J, K) is disabled if:

$$(\exists C_1 = false \in \tau(\texttt{00000000}))\wedge$$
$$(\exists C_2 = false \in \tau(\texttt{00000001}))\wedge$$
$$(\exists C_3 = false \in \tau(\texttt{00000010}))\wedge$$
$$(\exists C_4 = false \in \tau(\texttt{00000011}))$$

What is very important to show is that it is ensured that the polynomial solver will disable (J, K) for <u>any</u> C_1, C_2, C_3 and C_4 combination out of the $\tau(...)$ sets, because this is equivalent to allow choosing <u>any</u> initially false clause from each CT Line including J and K.

To prove the polynomial solver will disable any tuple (J, K) if there are initially false clauses contained in each of the Possible Solutions created by adding all 0/1 combinations around (J, K), let me start with a concrete example. Imagine we have, for this consideration, the following tuple (J, K) which is to be disabled:

```
J = 000--
K = 000--
```

We suppose neither J nor K are initially false clauses. What the solver now must do is to check if there is at least one initially false clause in each of the sets τ`(0000-)` and τ`(0001-)`.

This is what the solver implementation trivially does, for the following reason: We suppose that there is an initially false clause within τ`(0000-)` and an initially false clause within τ`(0001-)`. Furthermore, we know the solver disables (J, K) if the following condition is true:

```
if ((!ActiveClauseTuple[I][J] || !ActiveClauseTuple[I][K]) &&
    (!ActiveClauseTuple[H][J] || !ActiveClauseTuple[H][K]))
{
    if (Does_jk_Contain_ih(I, H, J, K))
    {
        ActiveClauseTuple[J][K] = false; // this code line disables tuple (J,K)
    }
}
```

The solver does the just shown "`if (!ActiveClauseTuple[I][J] ...)`"-check for really all I and H that are contained within (J, K), what is implemented in the way that the function `Does_jk_Contain_ih()` 'filters' those I and H which are contained within J, K. The definition of 'I and H contained within J, K' allows that I has all digits within J or/and K, plus one `0` digit placed at an arbitrary position p 'outside' J and K, i.e. where J and K do both have a '`-`' digit. The same applies to H, with the only difference that H is allowed to have one additional `1` digit 'outside' J and K. This is the exact basic assumption that there is an initially false clause $F_1 \in \tau$`(0000-)`, and an initially false clause $F_2 \in \tau$`(0001-)`. The fourth digit in F_1 is the additional `0` 'outside' J and K, and the fourth digit in $F_2 \in \tau$`(0001-)` is the additional `1` 'outside' J and K.

The rule used in the just shown situation, in which the polynomial solver can disable the tuple (J, K), i.e. setting `ActicaClauseTuple[J][K] := false`, by using merely *one* contained I and H pair, is to be called **"Rule 2, sub case a)"**.

4.3.2.4.1 Practical Recursion

But, what if there is no such F_1, or/and no such F_2? The definition of 'J, K contains I and H' allows placing one additional '0' and one additional '1' only, but now, we need to figuratively place `00, 01, 10 and 11` into the two trailing '--' digits of $(J \oplus K) =$ `000--`, so that we can check if there is an initially false clause in each of the sets τ`(00000)`, τ`(00001)`, τ`(00010)` and τ`(00011)`. But this is not implemented explicitly by neither Rule 1 nor Rule 2. So how does the solver overcome that problem?

The key is to realize that there is some kind of recursion within the tuple disabling mechanism. Let us regard again the code excerpt from the polynomial solver code listing:

```
if ((!ActiveClauseTuple[I][J] || !ActiveClauseTuple[I][K]) &&
    (!ActiveClauseTuple[H][J] || !ActiveClauseTuple[H][K]))
{
    if (Does_jk_Contain_ih(I, H, J, K))
    {
        ActiveClauseTuple[J][K] = false; // this code line disables tuple (J,K)
    }
}
```

We see that `ActiveClauseTuple[J][K]` is set to false if `ActiveClauseTuple[I][J]` or `ActiveClauseTuple[I][K]`, and `ActiveClauseTuple[H][J]` or `ActiveClauseTuple[H][K]` are false.

But each of the tuples (I, J), (I, K), (H, J) and (H, K) could have been disabled in the same way as (J, K)! So it is crucial for the understanding of the polynomial solver to examine the recursion which is hidden within the solver's tuple disabling mechanism. I said 'hidden' because the polynomial solver implementation does not explicitly implement a 'classical' recursion in the way a code procedure calls itself, but the recursion appears as a kind of 'side-effect', because `ActiveClauseTuple`-values depend from each other. For this reason, that the recursion is not purposefully programmed but happens in practice, we want to call it the **Practical Recursion**.

The following code excerpt shows how (J, K) can be disabled by e.g. (I, J) and (H, J), and how thereof (I, J) could have been disabled by some further $(I2, I)$ and $(H2, I)$ one 'recursion' step earlier:

Disabling of (I, J):

```
if ((!ActiveClauseTuple[I2][I] || !ActiveClauseTuple[I2][J]) &&
    (!ActiveClauseTuple[H2][I] || !ActiveClauseTuple[H2][J]))
{
    if (Does_jk_Contain_ih(I2, H2, I, J))
    {
        ActiveClauseTuple[I][J] = false; // this code line disables tuple (I,J)
    }
}
```

Then, the just false-switched tuple (I, J) does, among some false (H, J) or/and (H, K), cause the disabling of (J, K):

```
if ((!ActiveClauseTuple[I][J] || !ActiveClauseTuple[I][K]) &&
    //                  ^ (I,J) regarded before, see code block above...
    (!ActiveClauseTuple[H][J] || !ActiveClauseTuple[H][K]))
{
    if (Does_jk_Contain_ih(I, H, J, K))
    {
        ActiveClauseTuple[J][K] = false; // this code line disables tuple (J,K)
    }
}
```

Disabling (H, J) or/and (H, K) is not explained here in detail, but works analogous to the shown disabling of (I, J).

Please recall there's no recursive procedure calling or such which would be aimed to explicitly implement the Practical Recursion. Instead, for the just shown example, first the tuple (I, J) had to be disabled before it can take part in the disabling of tuple (J, K). Notice the polynomial solver's main-loops i,h,j,k are, within the while-ChangesExisting-loop, run through completely again and again until no `ActiveClauseTuple` value has changed any more. Like this it is ensured tuples can disable each other in an arbitrary order.

4.3.2.4.2 Clause Stack Notation

For the upcoming parts of the proof, a special notation will be used to visualize sets of clauses which contribute to the disabling of some regarded tuple. This visualization is now introduced by an example: Imagine the solver needs to decide if the following tuple (J, K) is to be disabled:

```
J =    000---
K =    000---
```

We suppose neither J nor K are initially false clauses and there is no *single* clause I contained within (J, K). What does the polynomial solver do? First, it checks if there's some initially false I *and* some initially false H contained within $(J \oplus K)$:

```
I ∈  τ(0000--)
H ∈  τ(0001--)
J =    000---
K =    000---
```

The just shown list of clauses/tuples disabling each other shall be called the **Clause Stack**. When you read the 'stacked' clauses top-down, you get the 'disabling chain' which does finally result in the disabling of the tuple represented by the bottom two clauses. For instance, in the just shown Clause Stack, we see some initially false I and H would disable (J, K). If there is no such initially false clause I, we already know it is still possible that the tuple (I, J) has been disabled by some further $I2$ and $H2$, as pointed out in the previous topic 'Practical Recursion'. The same applies to (H, J), which could have been disabled by some initially false clauses $I3$ and $H3$. These two cases can be visualized in form of the Clause Stack like this:

```
I2 ∈ τ(00000-)
H2 ∈ τ(00001-)
I  ∈ τ(0000--)
J  =   000---
```

and:

```
I3 ∈ τ(00010-)
H3 ∈ τ(00011-)
H  ∈ τ(0001--)
J  =   000---
```

It is important to realize that we can examine the (I, J) and (H, J) branches independently. To understand this, let's have another look at the code which implements the disabling of (J, K):

```
if ((!ActiveClauseTuple[I][J] || !ActiveClauseTuple[I][K]) &&
    (!ActiveClauseTuple[H][J] || !ActiveClauseTuple[H][K]))
{
    if (Does_jk_Contain_ih(I, H, J, K))
    {
        ActiveClauseTuple[J][K] = false; // this code line disables tuple (J,K)
    }
}
```

We see (I, J) and (H, J) do both appear AND-ed in the same `if (!ActiveClauseTuple[I][J] ...)`-condition, so they might both be required for the disabling of (J, K). But (I, J) can be disabled independently from (H, J), because at a 'deeper' recursion level [8] (I, J) and (H, J) do not appear any more in one and the same code block, but (I, J) and (H, J) do both form autonomous tuples which are in conflict (I and H have one conflict digit at some position 'p', as defined in 4.2.6) and can be disabled by different $I2, H2$ and $I3, H3$ tuples.

Finally three further, important properties of the Clause Stack:

- The two clauses tested for being initially false are always the top-two clauses within the Clause Stack. For instance, I and H, or $I2$ and $H2$, or $I3$ and $H3$. We use the notation I/H (respectively $I2/H2$, $I3/H3$ etc.) as short form for 'I and H' (respectively '$I2$ and $H2$', '$I3$ and $H3$' etc.).
- The 3rd and 4th clause from above are the 'current J and K', that means that the tuple built out of the 3rd and 4th clause from two will be the first one disabled if the two top-most clauses are initially false.
- Clause names (I, H, J, K, ...) can be noted electively in front of or behind of a Clause Stack line without changing any meaning ('`I2 =` τ`(00000-)`' means the same as 'τ`(00000-) = I2`').

4.3.2.4.3 Final Proof

With the aid of the gained knowledge about the Practical Recursion, the Clause Stack Notation, and by having found an idea for the still open proof, namely showing that *any* theoretically acceptable combination of initially false clauses is really taken on for disabling, we can now derive the final "Why UNSAT Detection is Reliable" proof.

This derivation shall be done using a 'half-general' notation covering the large set of different clauses which could be necessary to be processed by the solver. The notation, which does partially use '`d`'-digits instead of '`0`' or '`1`' digits, is still easy to understand but powerful enough to represent all possible cases.

The '`d`'-notation will now be introduced by an example, which is also the basis for the proof.

Imagine we have two disjoint J and K and need to decide if to disable this tuple:

```
ddd------ = J
---ddd--- = K
```

We define the following rules concerning the '`d`'-notation:

- Each digit '`d`' could be replaced by either '`0`' or '`1`'. But: the choice '`0` or `1`' must be equal within each column of the Clause Stack. This is because there must be no conflict among the clauses, except eventually the one conflict within two top-most clauses I and H. The 'x-th Clause Stack column' is defined as the set of digits at the x-th position from left within all clauses in the Clause Stack.
- Each digit '`d`' can be at an arbitrary location within J and K, meaning j_x or/and k_x can be '`d`' for three or more ('more' if involving the τ operator) $x \in \{1, 2, ..., n\}$, and it shall be valid to permute the digits of J and K and of all clauses that will be regarded in the following. The only restriction for the permuting is that it is used in the same way on *all* participated clauses, so that figuratively in the Clause Stack whole columns of digits are exchanged.
- It is generally not important where within the clauses '`d`' digits stand, but how many '`d`'-digits there are and if '`d`'s overlap among the regarded clauses, i.e. if in one clause the x-th digit is '`d`' and if the same applies to one or more other clauses as well.
- The size n of the clauses in Clause Line Notation can be chosen freely as long as it is large enough to take up all '`d`' digits. With larger clauses, the number of initially false clauses required to disable (J, K) might grow, but their processing is always uniform to the proceeding soon shown in the following text passages.

[8] With 'recursion' the Practical Recursion is meant, see previous topic 4.3.2.4.1.

For the rest of the proof, we suppose J and K do not overlap, i.e. they do not share any 'd' digits. This means if j_x is 'd' then k_x is '-' and vice versa, for any $x \in \{1, 2, ..., n\}$. n is, as always, the total length of the clauses in Clause Line Notation, respectively the literal index range in the SAT CNF. We will see this is the worst case; if the solver can correctly decide if such a disjoint (J, K) tuple is to be disabled, it will decide correctly for any (J, K) tuple.

The goal we must reach now to show that the polynomial solver works, is to prove the polynomial solver, as shown in the Code Listing (4.2.8) and as served for download, does accept *any* set of clauses $S(C)$ that was built by taking exactly one initially false clause from each of the CT Lines including J and K. This is again a worst case, if we take, from one or more CT Lines including J and K, more than one clause then this case contains as sub case the situation where we take only one clause from each CT Line.

Now, let us examine if the polynomial solver fulfills the requested demands.

We suppose there's no 3-SAT clause contained within J and K. Otherwise Rule 1 would disable J and K, as pointed out in the topic "Tuple Disabling by Rule 1". This means that there's at least one digit of each I and H 'outside' J and K (where both J and K have a '-' digit), so J and K must contain *any* combination of maximal four `0` or/and `1` digits (in this example represented by 'd', as already mentioned), maximal two from I and maximal two from H. For better understanding, here are four example Clause Stacks representing the possible types of all possible situations:

All four 'd' digits of I and H appear in J or, but not and, K:

```
dd----1-- = H
dd----0-- = I
ddd------ = J
---ddd--- = K
```

or, two digits of H appear in J and two digits of I appear in K:

```
dd----1-- = H
---dd-0-- = I
ddd------ = J
---ddd--- = K
```

or, two digits of H appear in J and the two digits of I appear in J (one digit) and K (one digit):

```
dd----1-- = H
d--d--0-- = I
ddd------ = J
---ddd--- = K
```

or, one 'd' digit of I appears in J and one in K, the same applies to the two 'd' digits of H:

```
-d--d-1-- = H
d--d--0-- = I
ddd------ = J
---ddd--- = K
```

Notice the just shown cases can be set up equally with J and K swapped.

What we see is that all possible I and H are 'contained' within J and K and will therefore, because they are contained, be accepted by the polynomial solver to disable tuple (J, K). The just shown four cases are, granted, incomplete samples. But until now the polynomial solver will use "Tuple Disabling by Rule 2, sub case a)", of which we already know it works. The more interesting case comes up if there's more than one I and H contained within J and K. Then we regard the I and H branch individually (see topic 4.3.2.4.2) and search for more than two initially false clauses. The set of rules to process this case, when there are three or more initially false clauses necessary for the disabling of (J, K), is to be called **"Tuple Disabling by Rule 2, sub case b)"**.

An example: we suppose the I of the just shown example is *not* an initially false clause. H is not regarded in the following example, but its treatment is analogous to that of I:

```
???????0- = I2 // '???...'  shall mean two 'd's could be anywhere within '???...'
???????1- = H2 // the two 'd's of H2 need not to be at the same locations as in I2!
d--d--0-- = I
ddd------ = J
---ddd--- = K
```

As supposed before, we can be sure $I2$ and $H2$ have their conflict digit (`0` for I and `1` for H, at position 'p') 'outside' I, J and K, where I, J and K all have a '`-`' digit. This is because if this would not be the case, Rule 1, or Rule 2, sub case a) would have disabled (J, K).

It is important to realize we do now, when involving $I2$ and $H2$ to disable (I, J), almost have the initial situation from the beginning of this proof again:

```
  ???????0-  = I2 = I_New [9]
  ???????1-  = H2 = H_New
τ(dddddd0--) ∋ I  = J_New
  ddd------  = J  = K_New
```

or

```
  ???????0-  = I2 = I_New
  ???????1-  = H2 = H_New
τ(dddddd0--) ∋ I  = J_New
  ---ddd---  = K  = K_New
```

This is *almost* the initial situation, because: We now can choose $I = J\_New$ absolutely freely, because we already proved that *any* I (and H) out of τ`(dddddd0--)` (resp. H out of τ`(ddddd1--)`) disables (J, K). So we must show that there is, for any combination of four '`d`' digits from $I2$ and $H2$, some I and the fixed J, or/and some I and the fixed K which contain these four digits from $I2$ and $H2$. This means we must find out if we can always select some I out of τ`(dddddd0--)` which does, when being overlaid with J, contain the four '`d`' digits of $I2$ and $H2$:

```
  ???????0-  = I2 = I_New
  ???????1-  = H2 = H_New
τ(dddddd0--) ∋ I  = J_New
  ddd------  = J  = K_New
```

or, if there's no such overlay $(I \oplus J)$ containing $I2$ and $H2$, if the same condition is alternatively fulfilled by some overlay $(I \oplus K)$:

```
  ???????0-  = I2 = I_New
  ???????1-  = H2 = H_New
τ(dddddd0--) ∋ I  = J_New
  ---ddd---  = K  = K_New
```

We, respectively the polynomial solver, do the following choice: As we can select I freely, we select an I to take up three of the four '`d`' digits of $I2$ and $H2$, including an eventual '`0`' digit at position 7:

```
  ???????0- = I2 = I_New
  ???????1- = H2 = H_New
  dd----0-- = I  = J_New
  ddd------ = J
  ---ddd--- = K
```

[9] `X_New` shall mean the clause X of the next deeper recursion level of the Practical Recursion. X is hereby the corresponding name for the clause used in the majority of cited source code excerpts.

or

```
  ???????0- = I2 = I_New
  ???????1- = H2 = H_New
  ---dd-0-- = I  = J_New
  ddd------ = J
  ---ddd--- = K
```

or

```
  ???????0- = I2 = I_New
  ???????1- = H2 = H_New
  d--d--0-- = I  = J_New
  ddd------ = J
  ---ddd--- = K
```

etc.

When choosing I as specified, in the worst case merely one 'd'-digit of $I2/H2$ is left, which lies either in J or in K.

So we see, also here any $I2/H2$ is contained by either (I, J) or (I, K) or both. This means that any $I2/H2$ disables either (I, J) or (I, K) or both. The same applies to some $I3/H3$ which might disable (H, J) or/and (H, K), if those two tuples should not have been already set false by some initially false H. So *any* theoretically valid $I2/H2$ and $I3/H3$ will finally disable (J, K). Please compare that with the source code of the polynomial solver:

Disabling of (I, J):

```
if ((!ActiveClauseTuple[I2][I] || !ActiveClauseTuple[I2][J]) &&
    (!ActiveClauseTuple[H2][I] || !ActiveClauseTuple[H2][J]))
{
    if (Does_jk_Contain_ih(I2, H2, I, J))
    {
        ActiveClauseTuple[I][J] = false;
    }
}
```

Then, together with some false (H, J) or/and (H, K) processed analogously to the just shown (I, J) case, disabling of (J, K):

```
if ((!ActiveClauseTuple[I][J] || !ActiveClauseTuple[I][K]) &&
    (!ActiveClauseTuple[H][J] || !ActiveClauseTuple[H][K]))
{
    if (Does_jk_Contain_ih(I, H, J, K))
    {
        ActiveClauseTuple[J][K] = false; // this code line disables tuple (J,K)
    }
}
```

But, what if there's no initially false J or K, and also no initially false I and no initially false $I2$? Then we need some $I4$ and $H4$ which will disable a tuple built out of any of the accepted $I2$ and any of the I. Now we are in the lucky situation that I and now also $I2$ can *both* be chosen freely as shown in the following visualization:

```
τ(dddddd000) ∋ I4
τ(dddddd001) ∋ H4
τ(dddddd00-) ∋ I2
τ(dddddd0--) ∋ I
  ddd------  = J
  ---ddd---  = K
```

We see we need to show that $I2$ and I contain any two clauses, each from τ(ddddddd000) and τ(ddddddd001). This is the case, because we can choose $I2$ to contain the first three of maximal four 'd' or/and '0' digits of $I4/H4$, and we can choose I to contain the resting one digit. When $I4$ and $H4$ has disabled $(I2, I)$, then we need some $(H2, I)$ false-switched according to the same scheme, and we continue disabling (I, J) or (I, K) as described some paragraphs earlier.

If we've not yet found enough initially false clauses, we can always, like just shown, choose $I(\lambda)$ and $I(\lambda + 1)$ to contain $I(\lambda + 2)/H(\lambda + 2)$. That's the reason why n can be arbitrarily larger than in the example just shown. λ is here the recursion level depth of the Practical Recursion. For instance, in the current proof I has $\lambda = 1$, $I2$ has $\lambda = 2$ and $I4$ has $\lambda = 3$.

Ok, now we've shown every tuple of disjoint J and K is disabled by any clause set built by selecting one initially false clause from each CT Line including J and K. But what if J and K are not disjoint but *overlap* partially, or totally. For instance:

```
ddd------ = J
--ddd---- = K
```

or

```
ddd------ = J
-ddd----- = K
```

or

```
ddd------ = J
d-dd----- = K
```

or

```
ddd------ = J
ddd------ = K
```

Will the solver correctly decide if to disable these (J, K) as well? To answer this question let's go through the steps we have done for the disjoint J, K:

- For any of the overlapping J, K cases, any initially I and H will be accepted if those I and H are contained within J, K. This is no new insight but has been proven already in the "Tuple Disabling by Rule 2, sub case a)" explanation.
- The only significant part that differs from the proceeding of disjoint J, K is the selection of an eventual $I2$ and $H2$. Now, we need to show that we can always select an I so that (I, J) or/and (I, K) contains *any* $I2/H2$ combination. Let's look at a possible example with J=ddd------ and K=-ddd-----:

  ```
    ?????0---  = I2 = I_New
    ?????1---  = H2 = H_New
  τ(dddd0----) ∋ I  = J_New
    ddd------  = J  = K_New
  ```

  or

  ```
    ?????0---  = I2 = I_New
    ?????1---  = H2 = H_New
  τ(dddd0----) ∋ I  = J_New
    -ddd-----  = K  = K_New
  ```

  We see this case, and each of the other cases where J and K overlap, is even less complicated than the one of disjoint J, K. The reason therefore is the following: The important question is if (I, J) or (I, K) can take up all four 'd' digits of $I2/H2$. And we already proved this can be done for disjoint J, K. Now, when J, K

overlap, I needs to 'take up' even less digits than when J, K are disjoint, and therewith we have a sub case of the disjoint J, K. That's the reason why we know the less demanding J, K overlap case is processed correctly by the polynomial solver as well.

So we finish the last part "Tuple Disabling by Rule 2" of the "Why UNSAT Detection is Reliable" proof by summarizing:

- The polynomial solver does accept any combination of initially false clauses to disable some tuple (J, K) when there's at least one of those initially false clauses from each of the CT Lines including J and K.
- The basic assumption of the "Why UNSAT Detection is Reliable" is that there is at least one initially false clause in *every* CT Line. This means that there are also initially false clauses in all of (J, K)'s CT Lines.
- So we know, under the basic assumption of this proof, a tuple (J, K) will be disabled by the polynomial solver, for any J and K out of the set of Possible Clauses, so that absolutely no enabled tuple(s) rest.
- Concluded, because no enabled tuple rests, we know the polynomial solver will return 'UNSAT', and thus detects UNSAT reliably.

Why UNSAT Detection is Reliable: □

I verified the main statements of this "Final Proof" using a self-written test program. No error was found so far. Unluckily the test program is hard to understand and not yet in an impeccably clean state for publishing (e.g. instructions and explanations have not been written down yet), so it's not in the downloadable zip file (concerning the zip file, see upcoming topic "Implementation on the Internet"). If you want to have the test program nevertheless, send a short request to louis@louis-coder.com. I'm currently planning to add the "Final Proof verification program" to the zip file within the next weeks.

4.3.2.5 Examining Example Runs

As introduced in the previous topics, there are four mechanisms (initialization, Rule 1, Rule 2 sub case a), Rule 2 sub case b)) which the polynomial solver, as shown in code listing 4.2.8 and as served for download, can use to disable some clause tuple (J, K). To clarify the requirements and internal working steps of these mechanisms, this topic now shows four example usages, whereby the initially false clauses, clauses J and K, plus partially the contents of other involved clauses are stated. The presented clauses do not have any connection with clauses shown in other text passages but have been chosen randomly just to demonstrate the proceeding(s) in each of the four sample cases.

Case 1 of 4:

- J or/and K are initially false clauses.
  Example:
  `J = ---000--- (= initially false)`
  `K = ------000`
  ⇒ Initialization disables tuple (J, K).

Case 2 of 4:

- There is an initially false clause within (J, K).
  Example:
  `I = ---0-0-0- (= initially false)`
  `J = ---000---`
  `K = ------000`
  ⇒ Rule 1 disables tuple (J, K).

Case 3 of 4:

- There are two initially false clauses with one conflict contained within (J, K).
  Example:
  `I = --0--0-0- (= initially false)`
  `H = --1---0-0 (= initially false)`
  `J = ---000---`
  `K = ------000`
  $\Rightarrow$ Rule 2, sub case a) disables tuple (J, K).

- When we examine the polynomial solver's code and the content of the participated clause variables, we see:

```
if ((!ActiveClauseTuple[--0--0-0-][---000---] || !ActiveClauseTuple[--0--0-0-][------000]) &&
    // ^ disabled as --0--0-0- is initially false; ^ disabled as --0--0-0- is initially false
    (!ActiveClauseTuple[--1---0-0][---000---] || !ActiveClauseTuple[--1---0-0][------000]))
    // ^ disabled as --1---0-0 is initially false; ^ disabled as --1---0-0 is initially false
{
    if (Does_jk_Contain_ih(
        I = --0--0-0-,
        H = --1---0-0,
        J = ---000---,
        K = ------000))
    {
        ActiveClauseTuple[---000---][------000] = false;
    }
}
```

Case 4 of 4:

- There are three or more initially false clauses with at least two conflicts, the initially false clauses might not all be contained within (J, K).
  Example:
  `C = --0--0-0- (= initially false)`
  `D = -01---0-- (= initially false)`
  `E = -11-----0 (= initially false)`
  `J = ---000---`
  `K = ------000`

  Notice C, D, E and the upcoming F, G are no variable- or clause names used in the solver source code but are used here in this topic to be able to refer to such clauses.

- First, among others, the tuple (F, G):
  `F = --1---0-0`
  `G = ------000`
  is disabled by D and E, through Rule 2, sub case a).

- This is because
  `D = -01---0-- (= initially false)`
  `E = -11-----0 (= initially false)`
  have one conflict at position $p = 2$, and the resting `0/1` digits of D and E appear within $F \oplus G$.

- When we examine the polynomial solver's code and the content of the participated clause variables, we see:

```
if ((!ActiveClauseTuple[-01---0--][--1---0-0] || !ActiveClauseTuple[-01---0--][------000]) &&
    (!ActiveClauseTuple[-11-----0][--1---0-0] || !ActiveClauseTuple[-11-----0][------000]))
{
    if (Does_jk_Contain_ih(
        D = -01---0--,
        E = -11-----0,
        F = --1---0-0,
        G = ------000))
    {
        ActiveClauseTuple[--1---0-0][------000] = false;
    }
}
```

- Until now, the regarded tuple (J, K) has not been disabled yet. Instead, the solver disabled, among others, the tuple (F, G), which will now be used to finally disable (J, K):
  `J = ---000---`
  `K = ------000`
  is disabled by C and (F, G), through Rule 2, sub case a).

- This is because
  `C = --0--0-0- (= initially false)`
  `F = --1---0-0`
  have one conflict at position $p = 3$, and the resting digits of C and F appear within $J \oplus K$.

- When we examine the polynomial solver's code and the content of the participated clause variables, we see:

```
if ((!ActiveClauseTuple[--0--0-0-][---000---] || !ActiveClauseTuple[--0--0-0-][------000]) &&
    // ^ disabled as --0--0-0- is initially false; ^ disabled as --0--0-0- is initially false
    (!ActiveClauseTuple[--1---0-0][---000---] || !ActiveClauseTuple[--1---0-0][------000]))
    // ^ unimportant;                              ^ (F,G) disabled before, see above!
{
    if (Does_jk_Contain_ih(
        C =     --0--0-0-,
        F =     --1---0-0,
        J =     ---000---,
        K = G = ------000))
    {
        ActiveClauseTuple[---000---][------000] = false;
    }
}
```

4.4 Questions and Answers

This part of the document gives answers to three important questions that can come up when delving into the polynomial algorithm.

4.4.1 Why While-ChangesExisting-Loop?

In the polynomial algorithm's source code and detailed explanations given in earlier topics, you can read about a while-loop that is iterated until no `ActiveClauseTuple[][]` value has changed any more. Why do we need the such-called while-ChangesExisting-loop?

The while-ChangesExisting-loop is required because the `ActiveClauseTuple[][]` value of a clause tuple (J_1, K_1) could have been set to false after it influenced the `ActiveClauseTuple[][]` value of an other clause tuple (J_2, K_2). This means we need to re-update the second tuple (J_2, K_2) after the first tuple (J_1, K_1) has been set to false. This re-updating will be done automatically via another while-ChangesExisting-loop iteration, which is initiated for sure by setting the Boolean flag `ChangesExisting := true` if any `ActiveClauseTuple[][]` value has been set to false, e.g. in our example that value of (J_1, K_1). This mechanism is furthermore also required to have finally *all* `ActiveClauseTuple[][]` values set to false if the CNF is UNSAT, because the polynomial solver detects SAT as soon as *any* Active Clause Tuple is still true, as defined in the Code Listing and Implementation Details.

4.4.2 Why Polynomial at all?

One can question why the polynomial solver is polynomial, in what concerns time and space, at all. The answer is the following: First, we are lucky that the count of 3-SAT Possible Clauses has a polynomial upper bound, namely $O(DigitNumber^3)$. Secondly, it is fortune that we can regard quadruples of clauses i, h, j, k, respectively their `ActiveClauseTuple`-values isolated, i.e. without dependencies from further clauses or data structures. This enables us to process each quadruple *once* only per while-ChangesExisting-loop iteration, *although* those quadruples i, h, j, k do mostly appear multiple times in the Clause Table. Finally, because there are polynomially many `ActiveClauseTuple` values, definitely $O(DigitNumber^{3\times 2})$ many, we have not more than that many while-ChangesExisting-loop iterations and do altogether get an algorithm that can solve 3-SAT in polynomial time and space.

4.4.3 One 6-SAT Clause instead of two 3-SAT Clauses?

Can we replace storing the two 3-SAT clauses of any Active Clause Tuple by storing one 3-to-6-SAT clause containing all 0/1 digits of the corresponding two 3-SAT ones? In my tests this did not work as then the solver can, for example, not distinguish any more between `ActiveClauseTuple[000---][00-0--]` and `ActiveClauseTuple[0-00--][000---]`. Both would be `Active3To6SATClause[0000--]`. The problem about this is that if e.g. `ActiveClauseTuple[000---][00-0--] = true` and `ActiveClauseTuple[0-00--][000---] = false`, what shall `Active3To6SATClause[0000--]` be then? When we add additional information to know which were the 'source 3-SAT clauses' of the 3-to-6-SAT clause, then this is nothing more than storing the *two* 3-SAT clauses directly, as done by the polynomial solver described in this paper.

4.5 Further Notes about the Polynomial Algorithm

4.5.1 Possible Clause Type Order

Notice that the order of Possible Clause Types is, in solver 'D', as shown in the code listing 4.2.8, not important. This means that you can permute the Possible Clauses arbitrarily, as long as you keep those Possible Clauses in a succession (in a 'block') that have their 0/1 digits at the very same positions within the clause. My previous Algorithm 'B' is 'sensitive' to the Possible Clause Type order and seems to work reliably only for the lexicographical order. I unknowingly chose that lexicographical order and did not know that the arrangement is crucial for the correctness of the solving result. It was the work of Mr. Prunescu who found out that Algorithm B fails for some CNFs if you permute the order of PC Types [10]. You can read about this in his paper [8]. I thank Mr. Prunescu very much for finding out and informing me about this circumstance. Because of Mr. Prunescu's discovery, I made all solvers from version 'C' on working independently from the order of the Possible Clause Types. This was reached mainly by adding the "while (ChangesExisting)"-loop. The current polynomial solver versions D-1.0 and D-1.1 have block-wise Possible Clause shuffling implemented and were therewith tested intensively to verify PC Type shuffling won't obviously make the solver fail. I could not yet find out why Algorithm B works only for special PC Type orders, but this knowledge is not required any more for the understanding of Algorithm D, because Algorithm D can handle, as already mentioned, any PC Type order.

[10] Mr. Prunescu uses the expression 'Supports' for what I call 'Possible Clause Types'.

4.5.2 Consistency Verifications done

To verify the polynomial solver does internally really work as expected, I implemented a consistency check in the downloadable versions D-1.0 and D-1.1. The following supposition was tested for contradiction: If the 3-SAT CNF to solve is satisfiable, then at the end of the polynomial solving process one or more `ActiveClauseTuple[J][K]` values are true. This means that both clauses J and K are supposed to be within one or more Active CT Lines, i.e. Clause Table Lines that include *only* true clauses. So I programmed a test that collects all Active Clause Tuples using an exponential, fail-safe algorithm. This test is located in the procedure `'DoConsistencyChecks()'`. In around one million test runs solving CNFs with around 4 to 12 literals and 1 to 120 clauses, the resting tuples (J, K) found by the polynomial solver on the one hand and those detected by `'DoConsistencyChecks()'` on the other hand did invariably match.

I added the verification functionality of `'DoConsistencyChecks()'` also to my older polynomial solver version 'B' (synonymously called 'Algorithm B', see topic 2) some month after I have published that version 'B'. The result of the additional verification was that my polynomial solver version 'B' did not 'cleanly' disable exactly those tuples which appeared in non-Active CT Lines only. But the Clause Table theory requested that exactly those tuples shall be still enabled at the end of the polynomial solving process which appear in at least one Active CT Line. This requirement is not always fulfilled by solver 'B'. That's the reason why I now implemented, in polynomial solver version 'D' (the topic of this paper), *four* main-loops (i,h,j,k), where version 'B' used three. Although the current version 'D' is therefore slower than version 'B', it did always, in all test runs performed, passed the `'DoConsistencyChecks()'` test. However, although solver 'B' kept too many tuples enabled, this only happened when the SAT CNF to solve was really satisfiable. For this reason, solver 'B' did never return any wrong final result ('satisfiable' or 'UNSAT'), exactly like the current version 'D', which did also never fail up to the present day this paper you are currently reading was published.

One remark: You might have noticed that the implementation of `'DoConsistencyChecks()'` contains two arrays named `'UnderlyingSolutions[][]'` and `'NewUnderlyingSolutions[][]'` which need an amount of memory that grows exponentially with the SAT CNF size. Also the count of iterations of the inner-most loops within `'DoConsistencyChecks()'` might grow exponentially. Concerning this I want you to notice that `'DoConsistencyChecks()'` is called *only* for testing and debugging purposes, the actual polynomial solver does *not* need the exponential-sized arrays.

Within this article, I made use of the 'τ-operator' and notations like '$+01(C_{mask}, p)$'. Because this more mathematical notation deviates from the way the (original) polynomial solver is implemented I created another polynomial solver, the '**Alternative Polynomial Solver**'. This alternative version uses Possible Clause char strings instead of indices, and does an explicit placing of `0` and `1` digits, as well as applying the τ operator on those Possible Clause char strings. The purpose of this alternative solver is to be able to better test the correctness of the mathematical notation with a lower probability that there were faults made when 'converting' the mathematical notation into the solver implementation. You can find the Alternative Polynomial Solver in the downloadable zip-file, in the directory "Algorithm D (newest version - never failed in tests)\Tests - not part of the actual polynomial solver\AlternativePolynomialSolver\(Release\)". The Alternative Polynomial Solver is derived from the original polynomial solver and can therefore also be used and re-compiled on Linux, like the original version.

It might finally be good to know that most errors I experienced during the development of the polynomial 3-SAT solving algorithm appeared when solving SAT CNFs with $n \geq 9$ [11]. That means if you want to test self-made variations of the polynomial algorithm, you should make it solve SAT CNFs which are not too small, because the larger the CNFs, the higher seems the probability that `'DoConsistencyChecks()'` finds inconsistencies, or even wrong results (solvable/UNSAT) are returned. Of course the CPU time required to solve bigger CNFs grows rapidly, so $n = 9$ or $n = 10$ turned out to be a good compromise.

4.5.3 Possible Optimizations

There are options to speed up the polynomial solver code shown in code listing 4.2.8.

[11] Please note: Errors appeared during development only, the final solver, as described in this paper and as served for download did never return a wrong result and `'DoConsistencyChecks()'` did never find any inconsistent, i.e. unexpected solver state.

First, we can pre-compute the results of repetitive condition checks and store them in Boolean arrays. These checks are if two Possible Clauses are in conflict (`IsInConflict(int ClauseIndex1, int ClauseIndex2)`) and if a Possible Clause appears in the current SAT CNF to solve (`ExistsInCNF(int ClauseIndex)`). The pre-computing must be done, for `IsInConflict()`, initially and if the CNF dimensions changed. The return values of `ExistsInCNF()` must be once stored in an array each time a new CNF is to be solved. As the count of Possible Clauses grows polynomially with the CNF dimensions, we are able to do the pre-computing in a reasonable amount of time and space.

Secondly, we can exclude (J, K) pairs whose `ActiveClauseTuple[J][K]` value is false. As you can see in the Code Listing 4.2.8, the inner J and K loops are always run through from 0 to $PossibleClauseNumber - 1$. It is probable that many of those loop iterations are surplus, as either `ActiveClauseTuple[J][K]` is already false, or J and K are in conflict. So we can store a list for each J, which K need to be processed at all. This list is, in the downloadable implementation, called `kInActiveClauseTuple[][]`. For further details you might want to view the source code of the optimized solver, located within the zip-file mentioned in the topic "History of the Polynomial Solver".

Thirdly, the following idea can be used to avoid unnecessary j/k loop iterations: It might happen that, for defined I and H combinations, there are no more (J, K) tuples left at all, later on in the solving process. This is the case as soon as all `ActiveClauseTuple[J][K]` belonging to some I and H have been disabled. So it is a good idea to not start iterating the J and K loops at all. In the optimized solver, this mechanism is implemented using the array `ih_HasNojk[I][H]`.

4.6 Complexity

Observation 4.6.1 *The Polynomial 3-SAT-Solving Algorithm shown in Code Listing 4.2.8 has a worst-case complexity of maximal $O(n^{18})$, whereby n is the SAT CNF's literal index 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 $n-1$, mean a complexity of $O(n^3)$.

As you can see in the Code Listing earlier on, there are the four nested loops i, h, j and k. We realize they do all maximal $PCNum$ many iterations. The block of loops i, h, j, k is repeated, within the while-ChangesExisting-loop, maximal $PCNum \times PCNum$ times, as with each repetition it is guaranteed to set at least one `ActiveClauseTuple[][]` element to false.

If any Possible Clauses are in conflict with each other and if a Possible Clause appears in the SAT CNF to solve can be pre-computed in $O(PCNum^2 \times n)$ respectively $O(PCNum \times a \times n)$, whereby a is the number of clauses the SAT CNF consists of. n comes from the fact that we must, in the worst-case, loop though all n `0/1/-` digits of the clauses to decide if the clauses are in conflict or within the SAT CNF. The pre-computing is faster as, and independent from the polynomial solver's main work (the i/h/j/k loops), so the pre-computing needn't to be regarded in the final complexity declaration. Also if a clause tuple (j, k) contains some second tuple (i, h) could be pre-computed in $O(PCNum^4 \times n)$. Here $O(PCNum^4)$ is the iteration-complexity of the four loops i, h, j, k and $O(n)$ is the loop that iterates through the digits of I, H, J, K to compare single clause digits (see implementation of e.g. `Does_jk_Contain_ih()` for details).

The Polynomial Solver located in the directory "Algorithm D (newest version - never failed in tests)\Solver Application\(Release\)" within the downloadable zip-file has the first two of the three mentioned pre-computations implemented. The third computation if (j, k) contains (i, h) is done in real time to save memory (RAM). When mentioning the complexity of the polynomial algorithm, I will always assume the complexity of the fully-optimized variant, i.e. the implementation that does all three pre-computations.

Summarized, we have to multiply the run-time complexities of i, h, j, k, `ActiveClauseTuple[][]`, so we get: $O((n^3) \times (n^3) \times (n^3) \times (n^3) \times (n^3) \times (n^3)) = O(n^{18})$. □

Please notice: the complexity just calculated is a theoretical worst case in which we assume only *one* `ActiveClauseTuple[][]` value is changed per while-ChangesExisting-loop iteration. But practically, many ar-

ray values are set to false within each while-iteration. In all tests I did the while-loop was not iterated more than two to three times. This would mean we have a practical complexity of $O(n^{12})$. Nevertheless, as I have no proof for this yet, I will cite the polynomial solver's complexity as calculated in the worst-case scenario ($O(n^{18})$), which is the correct upper bound for sure.

5 Implementation on the Internet

I have implemented the algorithm explained in this article 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. Besides Windows, the solver will also work on (most) Linux systems. I could successfully compile and run solver 'D' on Ubuntu 14.04 LTS. This is possible due to "#ifdef _WIN32 ... #else ... #endif" branches within the code. Instructions about how to compile are shipped within the zip-file containing the solver code.

The download URL of this article, the implementation of the described polynomial SAT-solving algorithm, and some more extras is given in the earlier topic 'History of Polynomial Algorithm' (2).

An optimized variation of Algorithm D, as described in this article's section 'The Polynomial 3-SAT-Algorithm', has been tested by me in form of the mentioned, downloadable console program. Around one million random CNFs of different sizes (mainly literal index ranges from 4 to 12, clause counts from 1 to 120), have been processed. No error was detected, as of this writing.

Within the downloadable zip-file, which contains the just mentioned optimized Algorithm D, there are also the original descriptions and implementations of the older Algorithms B and C, which were published by me earlier. Some third-party links in the Internet may still refer to Algorithm B or C, that's why you can find them in the zip-file, too.

My recommendation is to first view and try out all material related to Algorithm D, as this is the newest and best one. Algorithm B might be of interest to you if having read Mr. Prunescu's comment ([8]). Algorithm A should be ignored as it is faulty and of too little quality. Algorithm C contains an error and is therefore useless.

6 Summary

This article 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 documents describing Algorithm B, C and D. It's also owing to Mr. Prunescu that my algorithm 'B', respectively Mr. Prunescu's variant of it, is going to be published in a scientific journal.

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 surprizing computer program of Matthias Müller, https://imar.academia.edu/MihaiPrunescu (link checked 2015-November-01, link also on my website, see this paper's section "History of Polynomial Algorithm").

A Appendix

Here is the remaining code announced in the topic 'Code Listing' (4.2.8). When you copy the code from the Code Listing to a .cpp file and add the following second part of the code behind, you get the (unoptimized) polynomial solver. To use the solver, call the function `DoesCNFHaveASolution(DigitNumber, ClauseNumber, CNF)` and check its return value; if the return value is true, the CNF is solvable, if the return value is false, the CNF has no solution. The code to generate SAT CNFs respectively read SAT CNFs from files can be taken from my downloadable sample implementation, or you program it yourself. Notice the polynomial solver does not output a 'model' (i.e. a solution string $\in \{0,1\}^{DigitNumber}$), but decides only if the CNF is solvable or not. CNF has to be a char string with all clauses of the SAT CNF in a succession, in my Clause Line Notation. This implies the CNF string has exactly `DigitNumber * ClauseNumber` chars, whereby each char is out of {`0`,`1`,`-`}. The following code, combined with the polynomial solver's main function `DoesCNFHaveASolution()`, has been tested exactly like listed, so it should work. If you have still problems, you can e-mail me. Notice that there is an optimized, already compiled version downloadable, as mentioned in this paper's topic "Implementation on the Internet".

```
bool IsEqual(char* Clause1, char* Clause2)
{
    // This function checks if the two char strings are identical.

    for (int m = 0; m < DigitNumber; m ++)
        if ((Clause1[m] != Clause2[m]))
            return false;

    return true;
}
bool IsEqual(int ClauseIndex1, int ClauseIndex2)
{
    return IsEqual(PossibleClauses[ClauseIndex1], PossibleClauses[ClauseIndex2]);
}
```

```
bool IsInConflict(char* Clause1, char* Clause2)
{
    // IsInConflict() checks if two clauses are in conflict.
    // Two clauses are in conflict if there's '0' at some position
    // in Clause1 where there's '1' in Clause2 (or vice versa).
    //
    // For example, it is in conflict:
    // "0-0-" and "1-0-".
    //
    // Not in conflict are for instance:
    // "0-0-" and "-00-", or
    // "0-0-" and "-0-0", or
    // "0-0-" and "-10-".

    for (int m = 0; m < DigitNumber; m ++)
        if ((Clause1[m] == '0' && Clause2[m] == '1') ||
            (Clause1[m] == '1' && Clause2[m] == '0'))
            return true;

    return false;
}
bool IsInConflict(int ClauseIndex1, int ClauseIndex2)
{
    return IsInConflict(PossibleClauses[ClauseIndex1], PossibleClauses[ClauseIndex2]);
}

bool ExistsInCNF(char* Clause)
{
    // Returns true if Clause appears in the CNF to solve.
    // Makes use of the global variables ClauseNumber,
    // DigitNumber and CNF.

    for (int i = 0; i < ClauseNumber; i ++)
        if (IsEqual(Clause, &CNF[i * DigitNumber]))
            return true;

    return false;
}
bool ExistsInCNF(int ClauseIndex)
{
    return ExistsInCNF(PossibleClauses[ClauseIndex]);
}

void PossibleClauses_Create()
{
    // Fills the array PossibleClauses[][] and sets PossibleClauseNumber.
    // Those variables are global; the procedure accesses further global
    // variables, namely SAT_TYPE and DigitNumber.

    PossibleClauseNumber = 0;

    if (SAT_TYPE == 3)
    {
        // NOTE: if DigitNumber == 4, the PossibleClauses for 3-SAT are:
        // 000-
        // 001-
```

```
// 010-
// 011-
// 100-
// 101-
// 110-
// 111-
// 00-0
// 00-1
// 01-0
// 01-1
// 10-0
// 10-1
// 11-0
// 11-1
// 0-00
// 0-01
// 0-10
// 0-11
// 1-00
// 1-01
// 1-10
// 1-11
// -000
// -001
// -010
// -011
// -100
// -101
// -110
// -111
// PossibleClauseNumber = (4 'blocks (i.e. Clause Types)' * 8 '0/1 combinations') = 24

// position of first '0' or '1' digit in Possible Clause:
for (int D1 = 0; D1 < DigitNumber - 2; D1 ++)
{
    // position of second '0' or '1' digit in Possible Clause:
    for (int D2 = D1 + 1; D2 < DigitNumber - 1; D2 ++)
    {
        // position of third '0' or '1' digit in Possible Clause:
        for (int D3 = D2 + 1; D3 < DigitNumber; D3 ++)
        {
            for (int C = 0; C < 8; C ++) // 000, 001, 010, 011, 100, 101, 110, 111
            {
                // We arrive here O(DigitNumber^3) times!
                // So there are O(DigitNumber^3) PossibleClauses!

                for (int m = 0; m < DigitNumber; m ++)
                    PossibleClauses[PossibleClauseNumber][m] = '-';

                PossibleClauses[PossibleClauseNumber][D1] = (C & 4 ? '1' : '0');
                PossibleClauses[PossibleClauseNumber][D2] = (C & 2 ? '1' : '0');
                PossibleClauses[PossibleClauseNumber][D3] = (C & 1 ? '1' : '0');
                PossibleClauses[PossibleClauseNumber][DigitNumber] = '\0';

                if (PossibleClauseNumber >= POSSIBLE_CLAUSE_NUMBER_MAX)
                { // if this happens, try to increase POSSIBLE_CLAUSE_NUMBER_MAX and re-compile
```

```
                            printf("Warning: PossibleClauseNumber too large in ");
                            printf("PossibleClauses_Create() !");
                            getchar();
                        }

                        PossibleClauseNumber ++;
                    }
                }
            }
        }
    }
}

bool Does_ih_HaveOneConflict(char* I, char* H)
{
    // Returns true if clause I has ONCE some 0 digit where
    // clause H has a 1, or vice versa. If I and H have two
    // or more conflicts, the procedure returns false. If I
    // and H have no conflict, false is returned as well.

    int ConflictPos = -1; // the 'p' position; -1 means not found

    for (int m = 0; m < DigitNumber; m ++)
    {
        if ((I[m] == '0' && H[m] == '1') ||
            (I[m] == '1' && H[m] == '0'))
        {
            if (ConflictPos == -1)
                ConflictPos = m;
            else
                return false;
        }
    }

    return (ConflictPos > -1); // if ConflictPos set, then one conflict was found
}
bool Does_ih_HaveOneConflict(int i, int h)
{
    return Does_ih_HaveOneConflict(PossibleClauses[i], PossibleClauses[h]);
}

bool Does_jk_Contain_ih(char* I, char* H, char* J, char* K)
{
    int ConflictPos = -1; // the 'p' position
    for (int m = 0; m < DigitNumber; m ++)
        if ((I[m] == '0' && H[m] == '1') ||
            (I[m] == '1' && H[m] == '0'))
            if (ConflictPos == -1)
                ConflictPos = m;
            else
                return false;

    bool ICoveredHere[DIGIT_NUMBER_MAX];

    for (int m = 0; m < DigitNumber; m ++)
        ICoveredHere[m] = false; // initialize
```

```
    for (int m = 0; m < DigitNumber; m ++)
        if (J[m] != '-')
            ICoveredHere[m] = true;

    for (int m = 0; m < DigitNumber; m ++)
        if (K[m] != '-')
            ICoveredHere[m] = true;

    for (int m = 0; m < DigitNumber; m ++)
        if (I[m] != '-' && ICoveredHere[m] == false)
            if (m != ConflictPos)
                return false;

    bool HCoveredHere[DIGIT_NUMBER_MAX];

    for (int m = 0; m < DigitNumber; m ++)
        HCoveredHere[m] = false; // initialize

    for (int m = 0; m < DigitNumber; m ++)
        if (J[m] != '-')
            HCoveredHere[m] = true;

    for (int m = 0; m < DigitNumber; m ++)
        if (K[m] != '-')
            HCoveredHere[m] = true;

    for (int m = 0; m < DigitNumber; m ++)
        if (H[m] != '-' && HCoveredHere[m] == false)
            if (m != ConflictPos)
                return false;

    return true;
}
```

[End of document 'Polynomial Exact-3-SAT-Solving Algorithm'.]