Polynomial Exact-3-SAT-Solving Algorithm

Matthias Michael Mueller
louis@louis-coder.com

Tue, 2015-11-03
Version D-1.0

Abstract

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

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

2 History of Polynomial Algorithm

Please notice that there are several different algorithms on the Internet that have been invented by me during the past ≈3 years. What you are currently reading is my *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 version from November 2015, named *Algorithm D*.

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

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

For all $i = 0, ..., a-1$ the variables x_{i1}, x_{i2}, x_{i3} are pairwise distinct. If ϵ_{ij} is placed before x_{ij}, it shall mean that x_{ij} is negated. a is used in this document as the number of AND-ed terms. The literal 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.

One way to decide if the 3-SAT instance is solvable would be to try out all possible solutions. The problem about this proceeding is that, if there is no solution at all, you had to test as many as 2^n solutions to finally find out that you cannot solve the CNF. I strongly suppose the algorithm described in this document can determine the 3-SAT CNF's solvability, even in the worst-case, after maximal n^{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 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 paper, for the 3-SAT CNF that is to be solved by the polynomial solver, as explained in the previous topic. As already mentioned, n is the literal 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 "1" chars and exactly* $n-3$ *"-" chars.*

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

Example for $n = 5$:

$$(x_1 \vee x_3 \vee \neg x_5) \quad = \quad \texttt{1-1-0}$$
$$(x_2 \vee \neg x_4 \vee \neg x_5) \quad = \quad \texttt{-1-00}$$

A CNF is, within this document and related program code, notated 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 paper I'll use the expressions 'char(s)' and 'digit(s)' synonymously.

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.*

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

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

Example:

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

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

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

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

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

Proof: Char S_i is, for any $i = 1...n$, either "0" or "1". Those two states do, as defined in 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_i means that the literal i did not exist in the original, mathematical clause notation and thus doesn't decide if the clause is satisfied or not. So $S_i = C_i = \{0, 1\}$ states C as satisfied, because it corresponds to the mathematical notation $S_i = C_i = \{false, true\}$. □

Observation 4.1.9 *It is not possible to satisfy all* $2^3 = 8$ *Filled Clause Types 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 4.1.10 *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** ($c = \{0, ..., PossibleClauseNumber - 1\}$). *Clause indices are mostly implemented in form of for-loop-variables, pointing to an element out of the set of Possible Clauses.*

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

Definition 4.1.11

- *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 includes 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 includes 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 4.1.12 *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 4.1.9, it is *not* possible to satisfy *all* of those clauses using some Possible Solution. If at least one CT Line is not to be satisfied as all its clauses are absent from the CNF, then in each CT Column there's at least one Filled Possible Clause missing, so the Clause Table DNF can be satisfied as *every* CT Column can be satisfied. Please notice that either *all* instances of the same clause are absent from the CNF, or none. For instance, either both `000-` clauses in the first two CT Lines, first column, are absent or none of them. This means that all copies of a PC within the Clause Table are either all replaced by true or are all replaced by false. It is not possible that some are replaced by true and others by false. □

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 a further Windows-console

[2] For education, please look up the mentioned absent PCs in the CT example of definition 4.1.11.

program which generates and displays the Clause Table DNF for a choose-able 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 paper 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 my 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 4.1.12. Please notice: Although the Clause Table has exponentially many CT Lines, the upcoming algorithm can evaluate it in polynomial time and space! So although the Clause Table is actually 'too large', this fact does *not* mean that building up a polynomial algorithm upon the idea of the CT needs to fail. The reason therefore will be explained later on.

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 4.1.12 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.
- I is not in conflict with J or K.
- All `0/1` digits of I appear in J or/and K.

It is valid that (J and I), or (K and I), or (J and K and I) 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.
- I is not in conflict with J or K.
- H is not in conflict with J or K.
- I and H are either equal, or they have exactly one conflict at a well-defined position p within I and H. The position p is in this paper counted 1-based, i.e. the range of p is $\{1, ..., n\}$, 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. At this position p, J and K must both have a '`-`' digit.

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

In the first example, I and H are equal, and fully contained within (J, K):

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

In the second 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 third 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 noticed that there are two definitions of 'contained', 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 paper's appendix, in the following listing only the helper procedures' prototypes are given.*

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

int        DigitNumber;
int        ClauseNumber;
char*      CNF;

int        SAT_TYPE = 3;

#define    POSSIBLE_CLAUSE_NUMBER_MAX          8192

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


// prototypes of helper functions, see this paper'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 i = 0; i < PossibleClauseNumber; i ++)
    for (int j = 0; j < PossibleClauseNumber; j ++)
        ActiveClauseTuple[i][j] = !IsInConflict(i, j) && !ExistsInCNF(i) && !ExistsInCNF(j);

// *** 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 include at least one false-valued clause. Recall that a clause is treated 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:

- I is not in conflict with J or K,
- Furthermore it is requested that there are two tuples for which (`ActiveClauseTuple[I][J] = false` or/and `ActiveClauseTuple[I][K] = false`).
- Finally (J, K) must contain I (see definition 4.2.4).

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

- I is not in conflict with J or K, and also H is not in conflict with J or K.
- Furthermore it is requested that there are four tuples for which (`ActiveClauseTuple[I][J] = false` or/and `ActiveClauseTuple[I][K] = false`) and (`ActiveClauseTuple[H][J] = false` or/and `ActiveClauseTuple[H][K] = false`).
- Finally (J, K) must contain I and H (see definition 4.2.6). Notice that I and H are either equal or 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.

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 I call 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 3 `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 I's would set `ActiveClauseTuple[J][K] := false`:

```
I ∈ {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.

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 [3].

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

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. □

4.3.2 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 can, for each CT Line, have been 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` or `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 that can be built by selecting exactly three* `0` *or/and* `1` *digits out of an l-SAT clause* C*, whereby* $l >= 3$*. Each possible selection appears exactly once in* τ [4]*.*

Example:

τ(00001-) = {000---, 00-0--, 00--1-, 0-00--, 0-0-1-, 0--01-, -000--, -00-1-, -0-01-, --001-}

[4] tau like 'three out of'.

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 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-)'
```

Observation 4.3.11 *Let S be a set of s-SAT clauses,* $s >= 3$*. Then* $\tau(S + 0(...))$ *will return the same clauses as* $\tau(\tau(S) + 0(...))$*. Different clause order or multiple copies are here not counted as difference.*

Proof: We assume each clause from S has s many `0` or/and `1` digits:

$$S_x = (d_1, d_2, d_3, ..., d_s)$$

Choosing exactly three `0/1` digits c_1, c_2, c_3 can be expressed by:

$$c_1 = (d_1 \vee d_2 \vee d_3 \vee ... \vee d_s)$$
$$c_2 = (d_1 \vee d_2 \vee d_3 \vee ... \vee d_s)$$
$$c_3 = (d_1 \vee d_2 \vee d_3 \vee ... \vee d_s)$$

We define that any c_x and d_x contains either a positive number or a negative number. The absolute value of the number is the index of the `0` or `1` digit within the clause in Clause Line Notation. If the number is positive, it represents the position of a `1` digit, if the number is negative, it represents the position of a `0` digit.

As a simplification, in this proof we allow that c_x can be equal to c_y for all $x, y \in \{1, 2, 3\}$. This will not affect the correctness of the proof that within $\tau(S + 0(...))$ there are the same clauses as within $\tau(\tau(S) + 0(...))$, because the here shown proof includes the sub case where only different c_1, c_2, c_3 are allowed.

So, $C = (c_1, c_2, c_3) = \tau(S_x + 0(...))$ can be written as:

$$c_1 = (d_1 \vee d_2 \vee d_3 \vee ... \vee d_s \vee d_0)$$
$$c_2 = (d_1 \vee d_2 \vee d_3 \vee ... \vee d_s \vee d_0)$$
$$c_3 = (d_1 \vee d_2 \vee d_3 \vee ... \vee d_s \vee d_0)$$

d_0 represents the position of the `0` digit added by $+0(...)$.

$C = (c_1, c_2, c_3) = \tau(\tau(S_x) + 0(...))$ can be written as:

$$c_1 = ((d_1 \vee d_2 \vee d_3 \vee ... \vee d_s) \vee d_0)$$
$$c_2 = ((d_1 \vee d_2 \vee d_3 \vee ... \vee d_s) \vee d_0)$$
$$c_3 = ((d_1 \vee d_2 \vee d_3 \vee ... \vee d_s) \vee d_0)$$

Because the $\vee$ operation is associative, the last two shown ways of defining C are equivalent and thus $\tau(S+0(...)) = \tau(\tau(S) + 0(...))$. $\square$

One can see that this is also true for repetitive usage of the $\tau()$ operator, because the associative law says braces can be placed in an arbitrary way, also in chained usage of the proven observation.

Proving that '$\tau(S+1(...))$ will return the same clauses as $\tau(\tau(S)+1(...))$' works analogue with the same result that the assumption is true.

As an additional verification, I programmed a test project that tests the assumption of this proof for obvious non-correctness. You can find that test project named '**OutOfTest**' in the downloadable zip-file, in the directory "Algorithm D (newest version - never failed in tests)\Tests - not part of the actual polynomial solver\OutOfTest\(Release\)". Please read the annotations within OutOfTest's source code for more information. The OutOfTest project has been programmed and compiled for Windows only. In future versions I possibly will add a Linux version to the zip-file, if there should be request by the users. The conclusion out of the tests I did with OutOfTest is that this proof's assumption seems to be correct, the program did not find a contradiction.

Definition 4.3.12 *We define the function* $\mathbf{F}(\mathbf{S})$ *which takes a set S of 3-SAT clauses. F* [5] *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] F like "any false clause contained?"

Definition 4.3.13 *The function* $\mathbf{FT}(\mathbf{S}, \mathbf{B})$ *checks if there is an* `ActiveClauseTuple[`S_x`][B] = false` *for any* S_x *out of the set of clauses* S *and a second, single clause* B*. If there is at least one such false-stated clause tuple,* $FT(S, B)$ *returns true. If there is no false-stated clause tuple,* $F(S, B)$ *returns false.*

Definition 4.3.14 *We define the function* `AreAll01PathsFalse(`C_{in}, p`)` *as follows:*

$C_{in\_start} = \tau(J \oplus K)$
$p_{start} = 1$
ϕ `= count('-' chars in` $(J \oplus K)$`)`

```
bool AreAll01PathsFalse(C_in, p)
{
    if (p > φ)
        return false;

    if (¬ F(τ(C_in)))
        return (
            AreAll01PathsFalse(τ(C_in + 0(J ⊕ K, p)), p + 1) ∧
            AreAll01PathsFalse(τ(C_in + 1(J ⊕ K, p)), p + 1));
    else
        return true;
}
```

The behavior this function shows is the following: Initially, `AreAll01PathsFalse()` returns true if there is an initially false clause within the clause set $C_{in} = \tau(J \oplus K)$. If there is no initially false clause within, `AreAll01PathsFalse()` adds once a `0` digit and once a `1` digit to all clauses in the set $C_{in} = \tau(J \oplus K)$. Let us call $C_{in} + 0(...)$ the '0-path', and $C_{in} + 1(...)$ the '1-path'. The `0` and `1` digit will be placed at the position where $(J \oplus K)$ has its p-th '`-`' char. `AreAll01PathsFalse()` uses the τ operator on both the 0-path and then on the 1-path. This τ operator will make exact-3-SAT clauses out of the clauses of the 0- and 1-path, which do before the usage of τ all have exactly four `0/1` digits, namely the three `0/1` digits from C_{in} plus once the added `0` digit and once the added `1` digit. The τ operator will increase the number of clauses, making four 3-SAT clauses out of each 4-SAT clause. `AreAll01PathsFalse()` finally calls itself recursively, which will again check if there's any initially false clause in the 0-path, and in the 1-path. It is sufficient when at least one clause out of the 0- and one out of the 1-path make the sub call of `AreAll01PathsFalse()` return true, so figuratively we can say 'the recursive-call results of the clauses out of one τ application are OR-ed'. If `0` and `1` digits have finally been placed at *all* positions where $(J \oplus K)$ has a '`-`' char, then p is $\phi + 1$ and `AreAll01PathsFalse()` returns false. ϕ is defined as the number of '`-`' digits within $(J \oplus K)$.

Observation 4.3.15 *Under the condition that there is at least one initially false clause in each CT Line,* `AreAll01PathsFalse()` *will always return 'true' for any tuple* (J, K) *of Possible Clauses from the CT.*

Proof: We define the following variant of `AreAll01PathsFalse()`:

$C_{in\_start} = (J \oplus K)$
$p_{start} = 1$
ϕ = `count('-' chars in` $(J \oplus K)$`)`

```
bool AreAll01PathsFalseLessTau(C_in, p)
{
    if (p > φ)
        return false;

    if (¬ F(τ(C_in))
        return (
            AreAll01PathsFalseLessTau((C_in + 0(J ⊕ K, p)), p + 1) ∧
            AreAll01PathsFalseLessTau((C_in + 1(J ⊕ K, p)), p + 1));
    else
        return true;
}
```

The difference between `AreAll01PathsFalse()` and `AreAll01PathsFalseLessTau()` is that `AreAll01PathsFalseLessTau()` will not use the τ operator on the 0- and 1-paths before doing a recursive call.

`AreAll01PathsFalseLessTau()` practically probes:

$$
\begin{array}{l}
F(\tau(((((J \oplus K) + 0(L,1)) + 0(L,2)) + ...) + 0(L,\phi)))\wedge \\
F(\tau(((((J \oplus K) + 0(L,1)) + 0(L,2)) + ...) + 1(L,\phi)))\wedge \\
F(\tau(((((J \oplus K) + 0(L,1)) + 1(L,2)) + ...) + 0(L,\phi)))\wedge \\
F(\tau(((((J \oplus K) + 0(L,1)) + 1(L,2)) + ...) + 1(L,\phi)))\wedge \\
F(\tau(((((J \oplus K) + 1(L,1)) + 0(L,2)) + ...) + 0(L,\phi)))\wedge \\
F(\tau(((((J \oplus K) + 1(L,1)) + 0(L,2)) + ...) + 1(L,\phi)))\wedge \\
F(\tau(((((J \oplus K) + 1(L,1)) + 1(L,2)) + ...) + 0(L,\phi)))\wedge \\
F(\tau(((((J \oplus K) + 1(L,1)) + 1(L,2)) + ...) + 1(L,\phi)))
\end{array}
$$

`AreAll01PathsFalse()` practically probes:

$$
\begin{array}{l}
F(\tau(\tau(\tau(\tau(J \oplus K) + 0(L,1)) + 0(L,2)) + ...) + 0(L,\phi)))\wedge \\
F(\tau(\tau(\tau(\tau(J \oplus K) + 0(L,1)) + 0(L,2)) + ...) + 1(L,\phi)))\wedge \\
F(\tau(\tau(\tau(\tau(J \oplus K) + 0(L,1)) + 1(L,2)) + ...) + 0(L,\phi)))\wedge \\
F(\tau(\tau(\tau(\tau(J \oplus K) + 0(L,1)) + 1(L,2)) + ...) + 1(L,\phi)))\wedge \\
F(\tau(\tau(\tau(\tau(J \oplus K) + 1(L,1)) + 0(L,2)) + ...) + 0(L,\phi)))\wedge \\
F(\tau(\tau(\tau(\tau(J \oplus K) + 1(L,1)) + 0(L,2)) + ...) + 1(L,\phi)))\wedge \\
F(\tau(\tau(\tau(\tau(J \oplus K) + 1(L,1)) + 1(L,2)) + ...) + 0(L,\phi)))\wedge \\
F(\tau(\tau(\tau(\tau(J \oplus K) + 1(L,1)) + 1(L,2)) + ...) + 1(L,\phi)))
\end{array}
$$

$L = (J \oplus K); \phi =$ `number of '-' chars within L`

According to observation 4.3.11, $\tau(S + 0(L,p))$ is the same as $\tau(\tau(S) + 0(L,p))$, so we can be sure `AreAll01PathsFalse()` and `AreAll01PathsFalseLessTau()` do always return the same result (true or false) for the same clause tuple (J, K)!

We showed `AreAll01PathsFalse()` returns the same as `AreAll01PathsFalseLessTau()`, now we must show `AreAll01PathsFalseLessTau()` always returns true if there is an initially false clause in each CT Line including the clauses J and K.

C_{in} within `AreAll01PathsFalseLessTau()` will at full recursion depth contain all Underlying Solution strings of the CT Lines including J and K within `AreAll01PathsFalseLessTau()`'s C_{in} set. Colloquially spoken this is because the Underlying Solutions of the regarded CT Lines are to be built by adding 'all `0/1` combinations around $(J \oplus K)$', and that is what `AreAll01PathsFalseLessTau()` does. Finally it should be clear that using the τ operator on any Underlying Solution means getting all Possible 3-SAT Clauses of the CT Line related to the Underlying Solution. So this set of clauses $C_{US_x} = \tau(US_x)$ must contain some initially false clause because at least one clause from each CT Line is initially false, as basically assumed in this chapter. □

The correctness of the statements done in this chapter, and the function of `AreAll01PathsFalse()` and `AreAll01PathsFalseLessTau()` have been tested for contradiction using the test project '**Exponential Recursive Solver**'. You can find the Exponential Recursive Solver's source code plus the compiled Windows and Linux binary in the downloadable zip-file, in the directory "Algorithm D (newest version - never failed in tests)\Tests - not part of the actual polynomial solver\ExponentialRecursiveSolver\(Release\). The Exponential Recursive Solver uses its procedure `AreAll01PathsFalse()` to initialize the `ActiveClauseTuple` array-elements. After that initialization, the ActiveClauseTuples should all have the same value (true or false) as they would have at the end of the polynomial solver's solving process. This is verified by the procedure `DoConsistencyChecks()`, exactly like done in the polynomial solver. Please view the source code of the Exponential Recursive Solver for more information and details. Please notice there the compile flag `#define USE_TAU_SPARINGLY`, which decides if the Exponential Recursive Solver tests the algorithm of `AreAll01PathsFalseLessTau()` or alternatively, if `#define USE_TAU_SPARINGLY` is commented-out, of `AreAll01PathsFalse()`.

4.3.2.2 Definition of Practical Recursion

Although the polynomial solver, as shown in Code Listing 4.2.8, does not implement a 'classical' recursion in the sense that a code procedure calls itself, there is some kind of recursion observable. It is namely the case that `ActiveClauseTuple[J][K]` might get disabled after further ActiveClauseTuples 'required' by `ActiveClauseTuple[J][K]` got disabled. Each of those 'further' `ActiveClauseTuple`-values might, in in the same way, depend from a third, fourth, fifth, and so on set of `ActiveClauseTuple` values, which can, in return, each get disabled by even further ActiveClauseTuples. This recursion, meaning using the rules to disable ActiveClauseTuples on the disabling ActiveClauseTuples themselves, is to be called the '**Practical Recursion**'. I chose the linguistic expression 'practical' to express that the recursion appears in practice, meaning in real life, although not explicitly implemented in form of the mentioned recursive procedure-calling. The reason why the solver reliably detects UNSAT can only be understood when regarding the recursive dependencies of `ActiveClauseTuple` values.

Example:

`ActiveClauseTuple[J][K]` is set to false if there are two Possible Clauses I and H being contained in (J, K) for which:

$$(\neg\texttt{ActiveClauseTuple[I][J]} \vee \neg\texttt{ActiveClauseTuple[I][K]}) \wedge$$
$$(\neg\texttt{ActiveClauseTuple[H][J]} \vee \neg\texttt{ActiveClauseTuple[H][K]})$$

Now the important observation: Each of these four checked `ActiveClauseTuples` $(I, J), (I, K), (H, J), (H, K)$ could have been disabled according to the same scheme as `ActiveClauseTuple[J][K]`, i.e. by further I and H clauses.

Example: As already mentioned,

1) `ActiveClauseTuple[J][K]` is set to false if, for some contained I and H,

$$(\neg\texttt{ActiveClauseTuple[I][J]} \vee \neg\texttt{ActiveClauseTuple[I][K]}) \wedge$$
$$(\neg\texttt{ActiveClauseTuple[H][J]} \vee \neg\texttt{ActiveClauseTuple[H][K]})$$

Furthermore,

2) `ActiveClauseTuple[I][J]` is set to false if, for some contained $I2$ and $H2$,

$$(\neg\texttt{ActiveClauseTuple[I2][I]} \vee \neg\texttt{ActiveClauseTuple[I2][J]}) \wedge$$
$$(\neg\texttt{ActiveClauseTuple[H2][I]} \vee \neg\texttt{ActiveClauseTuple[H2][J]})$$

3) `ActiveClauseTuple[I][K]` is set to false if, for some contained $I3$ and $H3$,

$$(\neg\texttt{ActiveClauseTuple[I3][I]} \vee \neg\texttt{ActiveClauseTuple[I3][K]}) \wedge$$
$$(\neg\texttt{ActiveClauseTuple[H3][I]} \vee \neg\texttt{ActiveClauseTuple[H3][K]})$$

4) `ActiveClauseTuple[H][J]` is set to false if, for some contained $I4$ and $H4$,

$$(\neg\texttt{ActiveClauseTuple[I4][H]} \vee \neg\texttt{ActiveClauseTuple[I4][J]}) \wedge$$
$$(\neg\texttt{ActiveClauseTuple[H4][H]} \vee \neg\texttt{ActiveClauseTuple[H4][J]})$$

5) `ActiveClauseTuple[H][K]` is set to false if, for some contained $I5$ and $H5$,

$$(\neg\texttt{ActiveClauseTuple[I5][H]} \vee \neg\texttt{ActiveClauseTuple[I5][K]}) \wedge$$
$$(\neg\texttt{ActiveClauseTuple[H5][H]} \vee \neg\texttt{ActiveClauseTuple[H5][K]})$$

4.3.2.3 Example Usage of Practical Recursion

For a better understanding of the mechanisms of the Practical Recursion, let me show an example. The polynomial solver shall decide if a clause tuple ($J =$ `000-----`, $K =$ `---000--`) is to be disabled. We will trace the recursion steps the solver did to find out which clauses must have been initially false to disable the example tuple (J, K). We assume neither J nor K are initially false clauses, otherwise the tuple would have been disabled already at `ActiveClauseTuple` initialization. So, because J and K are true, we check if there are some further initially false clauses I and H *contained* within (J, K). Being contained means the `0/1` digits of I and H do all appear either in J or/and in K, except one. The one excepted digit is a `0` in I, and a `1` in H (or vice versa), which do both appear at location p 'outside' J and K, where both J and K have a '`-`' digit. In the following example p is first 7 and then 8, because the two exception digits appear first at the seventh position within I and H, and then at the eighth position within $I2, I3, I4, I5$ and $H2, H3, H4, H5$.

The tuple
`J =  000---- -` [6] and
`K =  ---000- -` is set to false if at least one out of
`I ∈ τ(0000000)-` and at least one out of
`H ∈ τ(0000001)-` is initially false.

If there is also no initially false I that could have disabled (J, I) then there is still the possibility that (J, I) got disabled by some further $I2$ and $H2$ contained in (J, I).

The tuple
`J =  000---- -` and
`I ∈ τ(0000000)-` is set to false if at least one out of
`I2∈ τ(0000000 0)` and at least one out of
`H2∈ τ(0000000 1)` is initially false.

$I2$ can be any clause out of τ('J overlaid with *any* of the clauses emerged from $\tau(I)$, plus an additional `0` char'). The same applies to $H2$, with the only difference that a `1` char is added.

Suchlike happens to (K, I):
The tuple
`K =  ---000- -` and
`I ∈ τ(0000000)-` is set to false if at least one out of
`I3∈ τ(0000000 0)` and at least one out of
`H3∈ τ(0000000 1)` is initially false.

[6] each clause has 8 digits ($n = 8$). I inserted a space char in some clauses to verify digits with the same index are printed at the same location within each clause; the closing bracket ')' of $\tau(...)$ makes this spacing necessary.

Let's see how (J, H) and (K, H) could get disabled:
The tuple
`J =  000---- -` and
`H` $\in \tau$`(0000001)-` is set to false if at least one out of
`I4`$\in \tau$`(0000001 0)` and at least one out of
`H4`$\in \tau$`(0000001 1)` is initially false.

Suchlike happens to (K, H):
The tuple
`K =  ---000- -` and
`H` $\in \tau$`(0000001)-` is set to false if at least one out of
`I5`$\in \tau$`(0000001 0)` and at least one out of
`H5`$\in \tau$`(0000001 1)` is initially false.

So, we saw that the tuple (J, K) = `(000-----,---000--)` gets disabled if in each of the following four clause sets at least one initially false clause is found:

τ`(00000000) (=I2,=I3)`
τ`(00000001) (=H2,=H3)`
τ`(00000010) (=I4,=I5)`
τ`(00000011) (=H4,=H5)`

We realize that in this example, the tuple (J, K) is disabled by the polynomial solver if there is at least one initially false clause contained in each of the Underlying Solutions created by 'adding all `0/1` combinations around (J, K)'. In the following topic we will see why this observation is generally valid.

4.3.2.4 Final Conclusion

Now, with the help of the observations made in the previous topics, I will finally show that the polynomial solver, as shown in the Code Listing (4.2.8) will disable any tuple (J, K) if there is at least one initially false clause contained in each Underlying Solution of the CT Lines including J and K.

This means, as already mentioned, that the polynomial solver has to disable any tuple (J, K) if there is an initially false clause contained in those Underlying Solutions built 'by adding all possible `0/1` combinations around $(J \oplus K)$'. Formally the polynomial solver must disable (J, K) if there is at least one initially false clause contained in *all* of the following clause tuples:

$$
\begin{aligned}
&FT(\tau(((((J \oplus K) + 0(L,1)) + 0(L,2)) + ...) + 0(L,\phi)), K) \wedge \\
&FT(\tau(((((J \oplus K) + 0(L,1)) + 0(L,2)) + ...) + 1(L,\phi)), K) \wedge \\
&FT(\tau(((((J \oplus K) + 0(L,1)) + 1(L,2)) + ...) + 0(L,\phi)), K) \wedge \\
&FT(\tau(((((J \oplus K) + 0(L,1)) + 1(L,2)) + ...) + 1(L,\phi)), K) \wedge \\
&FT(\tau(((((J \oplus K) + 1(L,1)) + 0(L,2)) + ...) + 0(L,\phi)), K) \wedge \\
&FT(\tau(((((J \oplus K) + 1(L,1)) + 0(L,2)) + ...) + 1(L,\phi)), K) \wedge \\
&FT(\tau(((((J \oplus K) + 1(L,1)) + 1(L,2)) + ...) + 0(L,\phi)), K) \wedge \\
&FT(\tau(((((J \oplus K) + 1(L,1)) + 1(L,2)) + ...) + 1(L,\phi)), K)
\end{aligned}
$$

$L = (J \oplus K); \phi =$ `number of '-' chars within L`

The proof shown in observation 4.3.11 lets us conclude that the just shown condition is equal to requesting that there must be an initially false clause in these clause tuples: $T =$

$$
\begin{array}{l}
FT(\tau(\tau(\tau(\tau(J \oplus K) + 0(L,1)) + 0(L,2)) + ...) + 0(L,\phi)), K) \wedge \\
FT(\tau(\tau(\tau(\tau(J \oplus K) + 0(L,1)) + 0(L,2)) + ...) + 1(L,\phi)), K) \wedge \\
FT(\tau(\tau(\tau(\tau(J \oplus K) + 0(L,1)) + 1(L,2)) + ...) + 0(L,\phi)), K) \wedge \\
FT(\tau(\tau(\tau(\tau(J \oplus K) + 0(L,1)) + 1(L,2)) + ...) + 1(L,\phi)), K) \wedge \\
FT(\tau(\tau(\tau(\tau(J \oplus K) + 1(L,1)) + 0(L,2)) + ...) + 0(L,\phi)), K) \wedge \\
FT(\tau(\tau(\tau(\tau(J \oplus K) + 1(L,1)) + 0(L,2)) + ...) + 1(L,\phi)), K) \wedge \\
FT(\tau(\tau(\tau(\tau(J \oplus K) + 1(L,1)) + 1(L,2)) + ...) + 0(L,\phi)), K) \wedge \\
FT(\tau(\tau(\tau(\tau(J \oplus K) + 1(L,1)) + 1(L,2)) + ...) + 1(L,\phi)), K)
\end{array}
$$

$L = (J \oplus K); \phi =$ `number of '-' chars within L`

The just shown formula is called '**Formula T**' [7], and is very similar to the formal description of the procedure '`AreAll01PathsFalse()`' shown in the previous topic 'General Definitions for Proof'. The only difference is that 'Formula T' uses $FT(S, K)$ where '`AreAll01PathsFalse()`' uses $F(S)$ to check if at least one clause from a set of clauses is initially false. This extension is done because the polynomial solver does always store if a clause is in an Active CT Line in clause *tuples* (`ActiveClauseTuple[][]`) rather than in single clauses, so we extend the set of clauses that must be initially false by a further clause K, that K from the tuple (J, K). This is valid as K does anyway appear in the tuple which will eventually be disabled and will thus not need 'extra `0/1` digits' that had to be found in some initially false clauses, so $F(S, K)$ and $F(S)$ will return the same results for equal input S.

We will now glean the processes that the polynomial solver shall do to disable any tuple (J, K), and we will see these requirements are fulfilled by the polynomial solver implementation as shown in this paper.

Let us start with the actions the polynomial solver shall do:

1) The solver shall disable the tuple (J, K) if either J or/and K is an initially false clause, or if an initially false clause I is contained within (J, K).

2) If (J, K) has not been disabled in point 1), the solver must regard recursively more than one tuple via the mechanism of the Practical Recursion. Generally, if a tuple (A, B) cannot be disabled by the rules just shown in point 1), the solver must instead check if there is an initially false clause within the tuples $(\tau(A{+}0(L,p))_x, B)$ and $(\tau(A{+}1(L,p))_y, B)$ for at least one x and one y [8], and with $p = \{1, ..., \phi\}$, $\phi = count\_of('-' \; within \; J \oplus K)$. The initial A_{start} is J, the initial B_{start} is K, and L is $(J \oplus K)$. Under the conditions of this proof, i.e. that there is at least one initially false clause in each CT Line, each branch of the recursion will find some initially false clause, as shown in observation 4.3.15.

Now here is a summary what the polynomial solver does do, and how one can derive these actions from the source code:

1) If J or K are initially false, the code that initializes the `ActiveClauseTuple`-array will set `ActiveClauseTuple[J][K]` to false. If there is some contained, initially false clause I within (J, K), then the "main-i-loop" (within the "`while (ChangesExisting) {...}`" code block) will 'sooner or later' point to I, as well as the "main-h-loop", which will also point to I (i = h is allowed, as mentioned in the code comments).
`(!ActiveClauseTuple[I][J] || !ActiveClauseTuple[I][K]) && (!ActiveClauseTuple[H=I][J] || !ActiveClauseTuple[H=I][K])` and `Does_jk_Contain_ih(I, H=I, J, K) == true` will apply to the described situation, and the code, as listed in the Code Listing (4.2.8) will set `ActiveClauseTuple[J][K] := false`. We see the polynomial solver implementation did here in point 1) work as requested.

2) To disable any tuple (A, B) the polynomial solver checks if the following tuples are disabled: $(\neg(I, A) \vee \neg(I, B)) \wedge (\neg(H, A) \vee \neg(H, B))$. This is done by the code lines '`if (!ActiveClauseTuple[I][J] || !ActiveClauseTuple[I][K]) && (!ActiveClauseTuple[H][J] || !ActiveClauseTuple[H][K]))`

[7] T like tau, because Formula T uses a lot of τ.
[8] x and y each select one clause out of the set of clauses built by the τ operator.

{...}', whereby the clauses J and K in this code example correspond to A and B currently regarded in the actual paragraph. The polynomial solver loops through all possible I and H and 'filters', by checking the return value of the function `Does_jk_Contain_ih(I, H, A, B)`, those I and H combinations with I and H having exactly one conflicting digit, and I and H having all resting digits within $A \oplus B$. As really *all* I and H out of the set of Possible Clauses are checked, the solver will 'sooner or later' also probe the enabled state of the tuples containing the desired I and H that fulfill the condition $(I = \tau(A + 0(L = (J \oplus K), p))_x, B)$ and $(H = \tau(A + 1(L = (J \oplus K), p))_y, B)$, as requested in the previous enumeration 'Let us start with the actions the polynomial solver shall do'. Because the polynomial solver, as shown in code listing 4.2.8, sets `ActiveClauseTuple[J][K] := false` for the very first I and H that fulfill the just mentioned properties, we can conclude that at least one x and at least one y is sufficient to disable (A, B), also as requested. Please recall that the clauses J and K in the just mentioned `ActiveClauseTuple[J][K] := false` correspond to A and B currently regarded in the actual paragraph. We see the polynomial solver implementation did also here in point 2) work as planned.

As the required and implemented algorithm properties match, we proved that the polynomial solver does implement the decision algorithm of the presented 'Formula T'. We also showed Formula T can assuredly detect when some tuple appears in non-Active CT Lines only, because Formula T is derived from the function `AreAll01PathsFalse()` about which we know it has this detection-ability. Combined, we now know the polynomial solver can therefore recognize any unsolvable 3-SAT CNF reliably as unsolvable ('UNSAT').

Why UNSAT Detection is Reliable: □

4.4 Questions and Answers

This part of the document gives answers to two 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 while-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.

4.5 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 works 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 [9]. 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 version D-1.0 has block-wise Possible Clause shuffling implemented and was 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.

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 version D-1.0. 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.

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 paper, I made extensive use of the 'τ-operator' and notations like '$+0(C_{mask}, p)$' and '$+1(C_{mask}, p)$'. 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.

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

4.5.3 Possible Optimizations

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

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".

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

The implementation has been done by me using Visual Studio 2005. This means the program is designed to run on any newer Microsoft Windows operating system. 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 paper, the implementation of the described polynomial SAT-solving algorithm, and some more extras is given in the earlier topic 'History of Polynomial Algorithm'.

An optimized variation of Algorithm D, as described in this paper'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, until now (2015-November-01).

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 of too little quality. Algorithm C contains an error and is therefore useless.

6 Summary

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

7 Acknowledgments

I want to thank Mr. M. Prunescu, Simion Stoilow Institute of Mathematics of the Romanian Academy, very much for giving me precious tips on how to improve the 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'. When you copy the code from the Code Listing to a 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. 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;

    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 true;
}
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;
}
```