Polynomial SAT-Solver

Algorithm Explanation

by Matthias Mueller (a.k.a. Louis Coder)
louis@louis-coder.com
Explanation Version 1.0 - December 1, 2013

Abstract

This document describes an algorithm that is supposed to decide in polynomial time and space if an exact 2- or 3-SAT CNF has a solution or not. To verify the algorithm for correctness, it has been implemented as computer program which successfully determined the solvability of more than 1 million exact-3-SAT formulas.

Contained Topics:

1 Introduction

1.1 Subject of this Document

This is the manual that gives information about how the 'Polynomial SAT-Solver' by Matthias Mueller works. You can download this document you are currently reading, the solver program (source code and Windows binary) and instructions from:

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

1.2 State of the Art

To solve Satisfiability Problem instances, there were only deterministic algorithms existing that need, in the worst case, exponential time. The algorithm explained in this document is guaranteed to require merely polynomial time to solve any exact 2- or 3-SAT formula. The author of the algorithm does furthermore suppose that the algorithm does always return correct results.

1.3 Requirements of the Reader

The Polynomial SAT-Solver algorithm should be of use to computer scientists and mathematicians who are familiar with the P-NP-problem. I assume that the reader has some experience in theoretical computer science, in using Windows programs and in programming.

2 Definitions

In the following topics I want to define some expressions used throughout the solver program, its source code and all regarding documents.

2.1 Task of the Solver

The solver is meant to be used to find out if a given problem instance of the such called 'Satisfiability Problem' has a solution or not. The output of the solver is either 'solvable' or 'UNSAT' (abbreviation for unsatisfiable, that means not solvable).

2.2 CNF

The solver processes 'SAT(isfiability)'-formulas, which are mathematically called 'CNFs'. CNF stands for 'Conjunctive Normal Form', which is, in common literature, notated as follows (example):

$$CNF = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$

The OR-ed 'x's are called literals, the AND-ed terms are called the clauses.

The task is to decide if there is a solution, i.e. if it is possible to assign each literal a value of true or false so that the whole CNF formula becomes true. If there is a solution, the CNF is "satisfiable" (or also called "solvable"), otherwise it is "unsatisfiable" (or also called "UNSAT"). For the example CNF above, this is possible, and a concrete solution would be:

$$x_1 = true$$
$$x_2 = true$$
$$x_3 = true$$

because when we insert this into the CNF we get:

$$CNF = (true \vee true \vee true) \wedge (true \vee false \vee false) \wedge (false \vee true \vee true)$$

which can be evaluated to 'true'.

An exact 3-SAT CNF is a CNF with always exactly three literals per clause, and an exact 2-SAT CNF is one with always exactly two literals per clause. For our needs, the same literal shall appear maximal once per clause.

There were already algorithms existing that solve 2-SAT CNFs in polynomial time, e.g. by using a logical resolution. It is possible to use the resolution also on 3-SAT CNFs, but the problem is that special CNFs can be constructed that lead to an exponential amount of time when being resolved. An example for such a class of 'hard to solve' 3-SAT CNFs is the such-called 'Pigeon Hole Problem'. Professor Haken proved in 1985 that resolution-based solvers need exponential time for solving large-enough pigeon hole problems [1]. Note that my (current) algorithm is not resolution based, so the exponential lower bound of resolution-based solvers does not apply to my algorithm.

2.3 DNF

'DNF' is the abbreviation of 'disjunctive normal form'. While a CNF is a conjunction of disjunctions, a DNF is the opposite, a disjunction of conjunctions. Example:

$$DNF = (x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \overline{x_2} \wedge \overline{x_3}) \vee (\overline{x_1} \wedge x_2 \wedge x_3)$$

As you surely know a conjunction is an AND-interrelation and a disjunction is an OR-interrelation. So a DNF consists of OR-ed AND-terms.

2.4 ClauseLine Notation

In documents, solver output and its source code, I use a special self-invented notation for CNFs. I call it 'the ClauseLine notation'.

Traditionally, a clause is written like this:

$$(x_1 \vee \overline{x_3} \vee x_5)$$

I would write that clause like this:

```
1-0-1
```

There's a maximal index of the literals in the CNFs to solve. I call this maximal possible index the DigitNumber. The example above has DigitNumber = 5, as the highest index of a literal is 5.

A mathematical clause can be converted to a ClauseLine as follows:

1. Write DigitNumber minus signs, for example (DigitNumber = 5): `-----`. A minus sign at position p means that the literal p is not existing in the clause.
2. Loop through the literals of the clause. Each literal has an index. If the literal is negated, write a 0 at the location denominated by the literal index, if the literal is not negated, place a 1 (in both cases replace the minus sign).

The whole CNF is written as a list of ClauseLines, i.e. one ClauseLine after another, each in a single document line. I decided to use the ClauseLine notation as I find it is a much better visualization of clauses and CNFs than 'tons of' indexed x variables.

2.5 PossibleClauses

I call the 'PossibleClauses' the set of all clauses that can appear in a CNF with highest literal index DigitNumber.

Example for 2-SAT, DigitNumber = 3; PossibleClauses are:	Example for 2-SAT, DigitNumber = 5; PossibleClauses are:	Example for 3-SAT, DigitNumber = 4; PossibleClauses are:
00-	00---	000-
01-	01---	001-
10-	10---	010-
11-	11---	011-
0-0	0-0--	100-
0-1	0-1--	101-
1-0	1-0--	110-
1-1	1-1--	111-
-00	0--0-	00-0
-01	0--1-	00-1
-10	...	01-0
-11	---00	01-1
	---01	...
	---10	-111
	---11	

Note that the order of the PossibleClauses follows a defined pattern:

For an exact 3-SAT formula, there are three 0 or 1 digits (respectively, for exact 2-SAT, two). The first PossibleClause has those digits at the very left. Then the 3rd digit (for exact 3-SAT) goes to the right, step by step. Then the 2nd digit moves one place to the right and the 3rd digit goes to the right again, and so on. Before a digit changes its place, all possible 0/1 combinations are run through. For exact 3-SAT, there are 8 possibilities of 0/1 combinations and for exact 2-SAT, there are 4.

It might be helpful to view the code of the implementation:

```
PossibleClauseNumber = 0; // reset

for (int D1 = 0; D1 < DigitNumber - 2; D1 ++)
// position of first 0 or 1 digit in PossibleClause
{
	for (int D2 = D1 + 1; D2 < DigitNumber - 1; D2 ++)
	// position of second 0 or 1 digit in PossibleClause
	{
		for (int D3 = D2 + 1; D3 < DigitNumber; D3 ++)
		// position of third 0 or 1 digit in PossibleClause
		{
			for (int C = 0; C < 8; C ++) // combination index
			// 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');

				PossibleClauseNumber ++;
			}
		}
	}
}
```

The PossibleClauses are always enumerated in this well-defined order. The PossibleClause indices begin, especially in the implementation, with 0.

2.6 In Conflict

Two clauses are 'in conflict' if they have at least one literal once negated and once non-negated at the same position within the two ClauseLines. Examples:

In conflict:

```
00---
-1--0
```

or

```
1--0-
0--1-
```

Not in conflict:

```
00---
-0--0
```

or

```
1--0-
-01--
```

3 The Polynomial SAT-Solving Algorithm

3.1 The Idea of the 'ClauseTable'

I allege that we can determine the solvability of any exact X-SAT Satisfiability Problem CNF with X >= 2 in the following way:

Loop through all possible solutions the SAT CNF could have. If the DigitNumber was 5 for example, the first possible solution is 00000, then 00001, then 00010, then 00011, then 00100, and so on. For each possible solution, write down a term that contains in AND-ed form all possible clauses that are not in conflict with the current solution. ('Possible Clauses' are explained in topic 2.5 and 'not in conflict' is explained in topic 2.6 of this document. The notation of clauses and 'DigitNumber' is explained in topic 2.5.)

Example of how to create a ClauseTable:

Imagine we have a 2-SAT CNF with DigitNumber = 3.

The first possible solution is: 000
So the first AND-term is:
`(00- & 0-0 & -00).`

The second possible solution is: 001
The second AND-term is:
`(00- & 0-1 & -01).`

...

The 8th possible solution is: 111
The 8th AND-term is:
`(11- & 1-1 & -11).`

When you OR all AND-terms, you get the 'ClauseTable' formula.

For the just shown example, we get the following ClauseTable formula:

```
(00- & 0-0 & -00) | <- "ClauseTable line 1"
(00- & 0-1 & -01) | <- "ClauseTable line 2"
(01- & 0-0 & -10) | <-...
(01- & 0-1 & -11) |
(10- & 1-0 & -00) |
(10- & 1-1 & -01) |
(11- & 1-0 & -10) |
(11- & 1-1 & -11)

 ^ "ClauseTable column 1"
       ^ "ClauseTable column 2"
             ...
```

It will be from use to have defined the expressions 'ClauseTable lines' and '-columns', as follows:

- All clauses within the same one AND-term shall build a 'ClauseTable line'. For example, 00-, 0-0 and -00 build ClauseTable line 1.
- All clauses at the same position within each AND-term shall build a 'ClauseTable column'. For example, 00-, 01-, 10- and 11- build ClauseTable column 1.

The ClauseTable formula is evaluated in the following way:

- You treat the clauses within the ClauseTable formula as Boolean variables.
- '&' denominates a Boolean AND and '|' denominates a Boolean OR.
- If a clause from the ClauseTable is *not* existing in the SAT CNF to solve, replace that clause by true.
- If a clause from the ClauseTable is existing in the SAT CNF to solve, replace that clause by false.

If the ClauseTable formula, as a whole, is true then the SAT CNF has any solution. If the ClauseTable formula, as a whole, is false then the SAT CNF is unsatisfiable, that means it has no solution. That means that the ClauseTable formula is then true if all clauses from at least one ClauseTable line are all *not* existing in the SAT CNF to solve. For example, a 2-SAT CNF with DigitNumber = 3 is solvable if it contains neither 00-, nor 0-0, nor -00.

3.2 Example ClauseTables

I want to show you three example-ClauseTables for three different CNF sizes. "|" shall denominate a logical OR and "&" shall denominate a logical AND. The examples are used to help to explain the idea, of course it is not from practical meaning to solve CNFs with only 3 or 4 different literals. But the ClauseTable can be created for arbitrary large CNFs as well.

Here are some example ClauseTables:

2-SAT, DigitNumber = 3:

```
(00- & 0-0 & -00) |
(00- & 0-1 & -01) |
(01- & 0-0 & -10) |
(01- & 0-1 & -11) |
(10- & 1-0 & -00) |
(10- & 1-1 & -01) |
(11- & 1-0 & -10) |
(11- & 1-1 & -11)
```

2-SAT, DigitNumber = 4:

```
(00-- & 0-0- & 0--0 & -00- & -0-0 & --00) |
(00-- & 0-0- & 0--1 & -00- & -0-1 & --01) |
(00-- & 0-1- & 0--0 & -01- & -0-0 & --10) |
(00-- & 0-1- & 0--1 & -01- & -0-1 & --11) |
(01-- & 0-0- & 0--0 & -10- & -1-0 & --00) |
(01-- & 0-0- & 0--1 & -10- & -1-1 & --01) |
(01-- & 0-1- & 0--0 & -11- & -1-0 & --10) |
(01-- & 0-1- & 0--1 & -11- & -1-1 & --11) |
(10-- & 1-0- & 1--0 & -00- & -0-0 & --00) |
(10-- & 1-0- & 1--1 & -00- & -0-1 & --01) |
(10-- & 1-1- & 1--0 & -01- & -0-0 & --10) |
(10-- & 1-1- & 1--1 & -01- & -0-1 & --11) |
```

```
(11-- & 1-0- & 1--0 & -10- & -1-0 & --00) |
(11-- & 1-0- & 1--1 & -10- & -1-1 & --01) |
(11-- & 1-1- & 1--0 & -11- & -1-0 & --10) |
(11-- & 1-1- & 1--1 & -11- & -1-1 & --11)
```

3-SAT, DigitNumber = 4:

```
(000- & 00-0 & 0-00 & -000) |
(000- & 00-1 & 0-01 & -001) |
(001- & 00-0 & 0-10 & -010) |
(001- & 00-1 & 0-11 & -011) |
(010- & 01-0 & 0-00 & -100) |
(010- & 01-1 & 0-01 & -101) |
(011- & 01-0 & 0-10 & -110) |
(011- & 01-1 & 0-11 & -111) |
(100- & 10-0 & 1-00 & -000) |
(100- & 10-1 & 1-01 & -001) |
(101- & 10-0 & 1-10 & -010) |
(101- & 10-1 & 1-11 & -011) |
(110- & 11-0 & 1-00 & -100) |
(110- & 11-1 & 1-01 & -101) |
(111- & 11-0 & 1-10 & -110) |
(111- & 11-1 & 1-11 & -111)
```

As already told, you can imagine the clauses '000-', '00-0', etc. as Boolean variables that are replaced by true if that clause is *not* existing in the SAT CNF. Any clause in the ClauseTable is replaced by false if that clause is existing in the SAT CNF.

In the following I want to show a sample evaluation. If a clause is still notated it shall be true and 'f' means the clause at the location of 'f' is false.

2-SAT, DigitNumber = 3:

```
(00- & 0-0 & -00) |
(00- & 0-1 & -01) |
(01- & 0-0 & -10) |
(01- & 0-1 & -11) |
(10- & 1-0 & -00) |
(10- & 1-1 & -01) |
(11- & 1-0 & -10) |
(11- & 1-1 & -11)
```

The following CNF:

```
00-
0-0
-11
1-0
11-
1-1
```

is not satisfiable. You know this by replacing clauses in the ClauseTable by false if they do appear in the CNF:

```
(f   & f   & -00) |
(f   & 0-1 & -01) |
(01- & f   & -10) |
(01- & 0-1 & f  ) |
(10- & f   & -00) |
(10- & f   & -01) |
(f   & f   & -10) |
(f   & f   & f  )
```

As each AND term contains at least once false ('f') the whole ClauseTable is false. So we know that the CNF is unsatisfiable, there is no solution.

Now imagine the CNF is shrunken:

```
      // 00- has been removed
0-0
-11
1-0
11-
1-1
```

the ClauseTable therefore:

```
(00- & f   & -00) | <- ClauseTable line 1
(00- & 0-1 & -01) | <- ClauseTable line 2
(01- & f   & -10) |
(01- & 0-1 & f  ) |
(10- & f   & -00) |
(10- & f   & -01) |
(f   & f   & -10) |
(f   & f   & f  )
```

We see that the second ClauseTable line does not contain false but only true. So the CNF is solvable.

As a last note I want to mention that 'ClauseTable' is sometimes abbreviated with 'CT', mainly in the source code.

3.3 Mathematic Notation of a ClauseTable

The following ClauseTable for 2-SAT, DigitNumber = 4:

```
(00-- & 0-0- & 0--0 & -00- & -0-0 & --00) |
(00-- & 0-0- & 0--1 & -00- & -0-1 & --01) |
(00-- & 0-1- & 0--0 & -01- & -0-0 & --10) |
(00-- & 0-1- & 0--1 & -01- & -0-1 & --11) |
(01-- & 0-0- & 0--0 & -10- & -1-0 & --00) |
(01-- & 0-0- & 0--1 & -10- & -1-1 & --01) |
(01-- & 0-1- & 0--0 & -11- & -1-0 & --10) |
(01-- & 0-1- & 0--1 & -11- & -1-1 & --11) |
(10-- & 1-0- & 1--0 & -00- & -0-0 & --00) |
(10-- & 1-0- & 1--1 & -00- & -0-1 & --01) |
(10-- & 1-1- & 1--0 & -01- & -0-0 & --10) |
(10-- & 1-1- & 1--1 & -01- & -0-1 & --11) |
(11-- & 1-0- & 1--0 & -10- & -1-0 & --00) |
(11-- & 1-0- & 1--1 & -10- & -1-1 & --01) |
(11-- & 1-1- & 1--0 & -11- & -1-0 & --10) |
(11-- & 1-1- & 1--1 & -11- & -1-1 & --11)
```

would be notated mathematically as the ClauseTable DNF:

$$\left(\left(\overline{x_1} \vee \overline{x_2}\right) \wedge \left(\overline{x_1} \vee \overline{x_3}\right) \wedge \left(\overline{x_1} \vee \overline{x_4}\right) \wedge \left(\overline{x_2} \vee \overline{x_3}\right) \wedge \left(\overline{x_2} \vee \overline{x_4}\right) \wedge \left(\overline{x_3} \vee \overline{x_4}\right)\right) \vee$$
$$\left(\left(\overline{x_1} \vee \overline{x_2}\right) \wedge \left(\overline{x_1} \vee \overline{x_3}\right) \wedge \left(\overline{x_1} \vee x_4\right) \wedge \left(\overline{x_2} \vee \overline{x_3}\right) \wedge \left(\overline{x_2} \vee x_4\right) \wedge \left(\overline{x_3} \vee x_4\right)\right) \vee$$
...

Remember: a clause is replaced by true if it is missing in the SAT CNF. If the whole ClauseTable is evaluated to true, the SAT CNF has any solution. You do not need to regard single literals, a clause in the ClauseTable is always replaced by true as a whole. For example, the SAT CNF:

$$Sample\_2\_SAT\_CNF = \left(\overline{x_1} \vee \overline{x_2}\right) \wedge \left(\overline{x_1} \vee \overline{x_3}\right) \wedge \left(\overline{x_1} \vee \overline{x_4}\right)$$

would replace the first three clauses in the ClauseTable DNF by false and the resting ones by true, because the first three clauses in the sample 2-SAT CNF appear 1-to-1 in the ClauseTable DNF.

3.4 ClauseTable Line Clauses are not in Conflict

The ClauseTable has a special structure:

- If a set of n many clauses are all not in conflict with each other, it is guaranteed that there is a ClauseTable line that contains all those n many clauses.

Example:

We want to regard the following ClauseTable of a 2-SAT CNF with DigitNumber = 4:

```
(00-- & 0-0- & 0--0 & -00- & -0-0 & --00) |
(00-- & 0-0- & 0--1 & -00- & -0-1 & --01) |
(00-- & 0-1- & 0--0 & -01- & -0-0 & --10) |
(00-- & 0-1- & 0--1 & -01- & -0-1 & --11) |
(01-- & 0-0- & 0--0 & -10- & -1-0 & --00) |
(01-- & 0-0- & 0--1 & -10- & -1-1 & --01) |
(01-- & 0-1- & 0--0 & -11- & -1-0 & --10) |
(01-- & 0-1- & 0--1 & -11- & -1-1 & --11) |
(10-- & 1-0- & 1--0 & -00- & -0-0 & --00) |
(10-- & 1-0- & 1--1 & -00- & -0-1 & --01) |
(10-- & 1-1- & 1--0 & -01- & -0-0 & --10) |
(10-- & 1-1- & 1--1 & -01- & -0-1 & --11) |
(11-- & 1-0- & 1--0 & -10- & -1-0 & --00) |
(11-- & 1-0- & 1--1 & -10- & -1-1 & --01) |
(11-- & 1-1- & 1--0 & -11- & -1-0 & --10) |
(11-- & 1-1- & 1--1 & -11- & -1-1 & --11)
```

Now let's invent a set of 6 clauses:

We begin with {00--}. Then we must add another clause that is not in conflict with any of the clauses that are already in the set. We cannot add for example 1-1- as the first digit of 1-1- is in conflict with that of 00--. But we could add 0-1-. So we have: {00--, 0-1-}. Let's extend this set until we have for example {00--, 0-1-, 0--1, -01-, -0-1, --11}.

This set of clauses is completely included in the 4th ClauseTable line:

```
(00-- & 0-0- & 0--0 & -00- & -0-0 & --00) |
(00-- & 0-0- & 0--1 & -00- & -0-1 & --01) |
(00-- & 0-1- & 0--0 & -01- & -0-0 & --10) |
(00-- & 0-1- & 0--1 & -01- & -0-1 & --11) | <- ClauseTable line 4
(01-- & 0-0- & 0--0 & -10- & -1-0 & --00) |
(01-- & 0-0- & 0--1 & -10- & -1-1 & --01) |
(01-- & 0-1- & 0--0 & -11- & -1-0 & --10) |
(01-- & 0-1- & 0--1 & -11- & -1-1 & --11) |
(10-- & 1-0- & 1--0 & -00- & -0-0 & --00) |
(10-- & 1-0- & 1--1 & -00- & -0-1 & --01) |
(10-- & 1-1- & 1--0 & -01- & -0-0 & --10) |
(10-- & 1-1- & 1--1 & -01- & -0-1 & --11) |
(11-- & 1-0- & 1--0 & -10- & -1-0 & --00) |
(11-- & 1-0- & 1--1 & -10- & -1-1 & --01) |
(11-- & 1-1- & 1--0 & -11- & -1-0 & --10) |
(11-- & 1-1- & 1--1 & -11- & -1-1 & --11)
```

When we want to describe exactly one ClauseTable line by a set of n many clauses then it is ingenious that n is the count of ClauseTable columns (n = 6 for the just shown example). I will later prove that any set of n clauses that are all not in conflict among each other does *always* describe at least one ClauseTable line. Please note that all clauses within the same ClauseTable column are always in conflict with each other, so we do *not* need to claim that the n clauses are from n different ClauseTable columns, as this is the case anyway if n clauses are not in conflict.

3.5 Main Idea of Polynomial SAT Algorithm

As you were told in the last topic, any set of n clauses that are all not in conflict among each other are guaranteed to describe a ClauseTable line, if n is the count of ClauseTable columns.

If all of those n clauses are not existing in the SAT CNF to solve, this would mean that we found a complete ClauseTable line that contains only clauses that do not appear in the SAT CNF. This would mean that the SAT CNF has a solution, as I explained in the previous topic 3.1.

3.6 Polynomial SAT Algorithm in Words

The actual task of my solver is to find out if there is a set of possible clauses that do all *not* appear in the SAT CNF to solve and that 'build' at least one ClauseTable line. The solver does this by regarding each clause 'triple' built out of three possible clauses. Each of those triples is further dissected into 'tuples'. How the algorithm does this is still to be explained and will be shown in the form of pseudo-code.

Remember the example set of possible clauses shown in topic 3.4, this set was: {00--, 0-1-, 0--1, -01-, -0-1, --11}.

The clauses {00--, 0-1-, 0--1, -01-, -0-1, --11} shall now all be absent from the SAT CNF to solve and are all within one ClauseTable line.

```
(00-- & 0-0- & 0--0 & -00- & -0-0 & --00) |
(00-- & 0-0- & 0--1 & -00- & -0-1 & --01) |
(00-- & 0-1- & 0--0 & -01- & -0-0 & --10) |
(00-- & 0-1- & 0--1 & -01- & -0-1 & --11) | <- in this ClauseTable line there are
                                               all clauses from the example set
(01-- & 0-0- & 0--0 & -10- & -1-0 & --00) |
(01-- & 0-0- & 0--1 & -10- & -1-1 & --01) |
(01-- & 0-1- & 0--0 & -11- & -1-0 & --10) |
(01-- & 0-1- & 0--1 & -11- & -1-1 & --11) |
(10-- & 1-0- & 1--0 & -00- & -0-0 & --00) |
(10-- & 1-0- & 1--1 & -00- & -0-1 & --01) |
(10-- & 1-1- & 1--0 & -01- & -0-0 & --10) |
(10-- & 1-1- & 1--1 & -01- & -0-1 & --11) |
(11-- & 1-0- & 1--0 & -10- & -1-0 & --00) |
(11-- & 1-0- & 1--1 & -10- & -1-1 & --01) |
(11-- & 1-1- & 1--0 & -11- & -1-0 & --10) |
(11-- & 1-1- & 1--1 & -11- & -1-1 & --11)
```

We can find out if a set of clauses build at least one ClauseTable line in the following way:

- There is a loop that loops through all possible clauses. Let the possible clause that loop is pointing to be named X.
- There is a second loop that loops through all possible clauses. Let the possible clause that loop is pointing to be named Y.
- There is a third loop that loops through all possible clauses. Let the possible clause that loop is pointing to be named Z.

Although not required, it is ingenious to make the second loop start at the next possible clause that comes after the first loop's clause. The third loop can start one possible clause behind the second one. 'Behind' shall mean at the next possible clause in the PossibleClauses[] array. The order of the possible clauses in that array is well-defined and fixed. Please attend topic 2.5, which is explaining this.

- If at least one clause, that means X, Y or/and Z is existing in the SAT CNF to solve, skip the current possible clause triple and check the next possible clause triple.
- If at least two clauses from the possible clause triple, that means X, Y and Z, are (partially) in conflict with each other, skip the current possible clause triple.
- If all three possible clauses, that means X, Y and Z are absent from the CNF to solve, and the clauses are not in conflict pair-wise, then check the Boolean array StillToBeChecked[][] and eventually set StillToBeChecked_New[][]:

```
        if (StillToBeChecked[X][Y] == true & StillToBeChecked[X][Z] == true)
               then set StillToBeChecked_New[Y][Z] = true;
```

I want to give a brief explanation what the array StillToBeChecked[][] is good for.

Let (Y & Z) mean:

- Y and Z is a clause tuple, built out of the last two clauses from the triple X, Y and Z,
- Y and Z do both *not* appear in the SAT CNF to solve,
- Y and Z are not in conflict.

StillToBeChecked_New[Y][Z] is practically set to true if (X & Y) and (X & Z). This means that we check (Y & Z) only if (X & Y) and (X & Z) has been found as true. Notice that StillToBeChecked[][] is, at beginning of the solving procedure, initialized to true. When the clause X is the first one of a new ClauseTable column then initialize StillToBeChecked_New[][] to all false. When the clause X is the last one of a ClauseTable column, then combine StillToBeChecked[][] with StillToBeChecked_New[][] by AND-ing their corresponding Boolean values. This will be explained more detailed in the upcoming topic 3.10.

3.7 Polynomial SAT Algorithm in Pseudo Code

I allege that the following program, shown as C-like pseudo code, can determine if there is at least one ClauseTable line whose clauses do all not exist in the passed SAT CNF. If there is at least one ClauseTable line whose clauses do all not exist in the SAT CNF to solve, then the SAT CNF has a solution.

Here is the code:

```
bool Does_SAT_CNF_Have_A_Solution(argument CNF)
{
        ClauseType PossibleClauses[];                   // define (reserve memory)
        CreatePossibleClauses(PossibleClauses[]);       // fill with data (see topic 2.5!)


        bool StillToBeChecked[][];                      // define (reserve memory)
        bool StillToBeChecked_New[][];                  // define (reserve memory)

        StillToBeChecked[][] = initialize with all true;

        foreach (PossibleClause X in PossibleClauses)
        {
                if (X is the first clause in a ClauseTable column)
                        StillToBeChecked_New[][] = all false;

                foreach (PossibleClause Y in PossibleClauses)
                {
                        foreach (PossibleClause Z in PossibleClauses)
                        {
                                if (    IsInConflict(X, Y) == false & // ('==' checks for equality)
                                        IsInConflict(X, Z) == false & // ('=' would assign a value)
                                        IsInConflict(Y, Z) == false &
                                        IsInCNF(X) == false &
                                        IsInCNF(Y) == false &
                                        IsInCNF(Z) == false &
                                        StillToBeChecked[X][Y] == true &
                                        StillToBeChecked[X][Z] == true)
                                {
                                        StillToBeChecked_New[Y][Z] = true;
                                }
                        }
                }

                if (X is the last clause in a ClauseTable column)
                        StillToBeChecked[][] =
                        StillToBeChecked[][] AND StillToBeChecked_New[][];
        }

        if (there is any StillToBeChecked[A][B] == true with
                B a PossibleClause from last ClauseTable column and
                A a PossibleClause from the ClauseTable column before B)
                return CNF_Has_A_Solution;
        else
                return CNF_Has_No_Solution;
}
```

I colorized special parts of the algorithm to make you know which pseudo code lines corresponds to which example run code line in the following topic.

3.8 SAT Algorithm Example Run

Imagine we need to find out if the following CNF is solvable:

```
01--
10--
11--
0-0-
1-0-
1-1-
0--0
1--0
1--1
-00-
-10-
-11-
-0-0
-1-0
-1-1
--00
--01
--10
```

This example CNF consists of 18 possible clauses. The algorithm should find out that the CNF is solvable, as there is a complete ClauseTable line whose clauses do all not appear in the CNF, as you can see here:

```
(00-- & 0-0- & 0--0 & -00- & -0-0 & --00) |
(00-- & 0-0- & 0--1 & -00- & -0-1 & --01) |
(00-- & 0-1- & 0--0 & -01- & -0-0 & --10) |
(00-- & 0-1- & 0--1 & -01- & -0-1 & --11) | <- this ClauseTable line contains none
                                               of the clauses of the example CNF!
(01-- & 0-0- & 0--0 & -10- & -1-0 & --00) |
(01-- & 0-0- & 0--1 & -10- & -1-1 & --01) |
(01-- & 0-1- & 0--0 & -11- & -1-0 & --10) |
(01-- & 0-1- & 0--1 & -11- & -1-1 & --11) |
(10-- & 1-0- & 1--0 & -00- & -0-0 & --00) |
(10-- & 1-0- & 1--1 & -00- & -0-1 & --01) |
(10-- & 1-1- & 1--0 & -01- & -0-0 & --10) |
(10-- & 1-1- & 1--1 & -01- & -0-1 & --11) |
(11-- & 1-0- & 1--0 & -10- & -1-0 & --00) |
(11-- & 1-0- & 1--1 & -10- & -1-1 & --01) |
(11-- & 1-1- & 1--0 & -11- & -1-0 & --10) |
(11-- & 1-1- & 1--1 & -11- & -1-1 & --11)
```

USAGE OF THE TRIPLE DISSECTING ALGORITHM:

I want to list the most important steps the algorithm from topic 3.7 does. In the following paragraphs, X, Y and Z are exactly those pointers to possible clauses that are used in the pseudo code. Also the two arrays 'StillToBeChecked' and 'StillToBeChecked_New' are the arrays mentioned in the pseudo code. For better understanding, please compare the following steps, especially their color, with the code listing from topic 3.7.

Please notice that I did only list those X, Y and Z triples that pass the big condition test. All other triples will also be checked by the pseudo code but I won't list those triples in the following code listing to keep the overview as far as possible.

>>>

```
StillToBeChecked is initialized to all true.
StillToBeChecked_New is initialized to all false.

X = 00--
Y = 0-1-
Z = 0--1
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[00--][0-1-] == true & StillToBeChecked[00--][0--1] == true
       => StillToBeChecked_New[0-1-][0--1] := true;
```

```
X = 00--
Y = 0-1-
Z = -01-
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[00--][0-1-] == true & StillToBeChecked[00--][-01-] == true
       => StillToBeChecked_New[0-1-][-01-] := true;

X = 00--
Y = 0-1-
Z = -0-1
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[00--][0-1-] == true & StillToBeChecked[00--][-0-1] == true
       => StillToBeChecked_New[0-1-][-0-1] := true;

X = 00--
Y = 0-1-
Z = --11
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[00--][0-1-] == true & StillToBeChecked[00--][--11] == true
       => StillToBeChecked_New[0-1-][--11] := true;

*** Y moves on ***

X = 00--
Y = 0--1
Z = -01-
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[00--][0--1] == true & StillToBeChecked[00--][-01-] == true
       => StillToBeChecked_New[0--1][-01-] := true;

X = 00--
Y = 0--1
Z = -0-1
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[00--][0--1] == true & StillToBeChecked[00--][-0-1] == true
       => StillToBeChecked_New[0--1][-0-1] := true;

X = 00--
Y = 0--1
Z = --11
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[00--][0--1] == true & StillToBeChecked[00--][--11] == true
       => StillToBeChecked_New[0--1][--11] := true;

*** Y moves on ***

X = 00--
Y = -01-
Z = -0-1
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[00--][-01-] == true & StillToBeChecked[00--][-0-1] == true
       => StillToBeChecked_New[-01-][-0-1] := true;

X = 00--
Y = -01-
Z = --11
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[00--][-01-] == true & StillToBeChecked[00--][--11] == true
       => StillToBeChecked_New[-01-][--11] := true;

*** Y moves on ***

X = 00--
Y = -0-1
Z = --11
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[00--][-0-1] == true & StillToBeChecked[00--][--11] == true
       => StillToBeChecked_New[-0-1][--11] := true;

*** X moves on ***
```

As X moves to the next ClauseTable column, we need to AND the values of the arrays StillToBeChecked and StillToBeChecked_New and then set StillToBeChecked_New to all false. So we get:

StillToBeChecked all false, EXCEPT:

```
StillToBeChecked[0-1-][0--1] == true
StillToBeChecked[0-1-][-01-] == true
StillToBeChecked[0-1-][-0-1] == true
StillToBeChecked[0-1-][--11] == true
```

```
       StillToBeChecked[0--1][-01-] == true
       StillToBeChecked[0--1][-0-1] == true
       StillToBeChecked[0--1][--11] == true
       StillToBeChecked[-01-][-0-1] == true
       StillToBeChecked[-01-][--11] == true
       StillToBeChecked[-0-1][--11] == true

*** Continue ***

X = 0-1-
Y = 0--1
Z = -01-
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[0-1-][0--1] == true & StillToBeChecked[0-1-][-01-] == true
       => StillToBeChecked_New[0--1][-01-] := true;
```

(and so on.)

<<<

At the end of the algorithm, when X, Y and Z reached their 'right-most' location in the ClauseTable, we need to check if there is any StillToBeChecked[A][B] == true with B from the last ClauseTable column and A from the column before. In our example, there would be 'StillToBeChecked[-0-1][--11] == true' left at the very end of the algorithm execution. So the triple dissecting algorithm did find out that there is at least one ClauseTable line satisfied, that means all its clauses are replaced by true as they are all missing in the SAT CNF.

3.9 How to evaluate exponential ClauseTable in polynomial Time

You might rightfully have noticed that the ClauseTable has exponential size. The ClauseTable has as many 'lines' (OR-ed AND-terms) as there are possible solutions for the SAT CNF. As the number of possible solutions grows exponentially with the SAT CNF's literal range (=DigitNumber), the ClauseTable's size grows exponentially, too.

But why should the triple dissecting algorithm, as shown in topic 3.7, used on the ClauseTable, evaluate the ClauseTable to true or false in polynomial time and space?

The solution is the following: the triple dissecting algorithm needs to process each triple X, Y and Z once only, no matter how often a triple appears in the ClauseTable, which is only an imaginary construction, that means it is practically not really created in memory.

For example, have a look at the following ClauseTable for 2-SAT, DigitNumber = 4:

```
(00-- & 0-0- & 0--0 & -00- & -0-0 & --00) | <- ClauseTable line 1
(00-- & 0-0- & 0--1 & -00- & -0-1 & --01) | <- ClauseTable line 2
(00-- & 0-1- & 0--0 & -01- & -0-0 & --10) | <- ...
(00-- & 0-1- & 0--1 & -01- & -0-1 & --11) |
(01-- & 0-0- & 0--0 & -10- & -1-0 & --00) |
(01-- & 0-0- & 0--1 & -10- & -1-1 & --01) |
(01-- & 0-1- & 0--0 & -11- & -1-0 & --10) |
(01-- & 0-1- & 0--1 & -11- & -1-1 & --11) |
(10-- & 1-0- & 1--0 & -00- & -0-0 & --00) |
(10-- & 1-0- & 1--1 & -00- & -0-1 & --01) |
(10-- & 1-1- & 1--0 & -01- & -0-0 & --10) |
(10-- & 1-1- & 1--1 & -01- & -0-1 & --11) |
(11-- & 1-0- & 1--0 & -10- & -1-0 & --00) |
(11-- & 1-0- & 1--1 & -10- & -1-1 & --01) |
(11-- & 1-1- & 1--0 & -11- & -1-0 & --10) |
(11-- & 1-1- & 1--1 & -11- & -1-1 & --11)
```

You can see that the triple {00--, 0-0-, -00-} appears twice, once in ClauseTable line 1 and once in ClauseTable line 2. But nevertheless, when you use the triple dissecting algorithm, as shown in topic 3.7, that algorithm will regard the triple {00--, 0-0-, -00-} only once. **The triple dissecting algorithm is designed in such a way that it needs to regard each triple once only.**

So the complexity of the polynomial solver is mainly the work that is to be done to process once each triple built out of three possible clauses. This will be examined later on in this document.

3.10 Failed Attempt of Simplification

We said that the triple dissecting algorithm works according to the scheme:

```
if (StillToBeChecked[X][Y] & StillToBeChecked[X][Z] &

       MissingInCNF(X) &
       MissingInCNF(Y) &
       MissingInCNF(Z) &
       IsNotInConflict(X, Y) &
       IsNotInConflict(X, Z) &
       IsNotInConflict(Y, Z))

       StillToBeChecked_New[Y][Z] = true;
```

So we regard three clauses X, Y and Z at once. But why do we not process less, let's say two clauses at once?

Could we do the following?

```
if (   StillToBeChecked[A] &
       MissingInCNF(A) &
       MissingInCNF(B) &
       IsNotInConflict(A, B))

       StillToBeChecked_New[B] = true;
```

The following observation might give the answer:

if ((A is not in conflict with B) &
(B is not in conflict with C))

...does *not* have to mean that A, B and C are in the very same ClauseTable line.

I will show you an example for such a wrong conclusion. Imagine we want to solve a 2-SAT CNF with DigitNumber = 3. The related ClauseTable is:

```
(00- & 0-0 & -00) | <-
(00- & 0-1 & -01) |
(01- & 0-0 & -10) | <-
(01- & 0-1 & -11) |
(10- & 1-0 & -00) |
(10- & 1-1 & -01) |
(11- & 1-0 & -10) |
(11- & 1-1 & -11)
```

You can easily see that (00- is not in conflict with 0-0) and also (0-0 is not in conflict with -10) does *not* mean that (00- is not in conflict with -10) is valid! So, IsNotInConflict(A, B) and IsNotInConflict(B, C) does *not* need to mean that IsNotInConflict(A, C). So we *need* to check:

IsNotInConflict(A, B) &
IsNotInConflict(A, C) &
IsNotInConflict(B, C)

Only if all of those three checks return true, we can be sure that A, B and C are in one and the same ClauseTable line.

Please remind the definition from topic 3.6: Let (X & Y) mean:

- X and Y do both not appear in the SAT CNF to solve,
- X and Y are not in conflict.

In topic 3.6 the array StillToBeChecked[][] was introduced. Now you might possibly understand better what this array is used for: the array StillToBeChecked[][] practically tells us if it 'makes sense' to still process the third tuple (B & C). (B & C) can only affect the solving in some way if StillToBeChecked[B][C] is true, what is the case if (A & B) and (A & C) was found before.

3.11 SAT Algorithm needs Idea of ClauseTable

Maybe you noticed that the ClauseTable formula, which is evaluated by the triple dissecting algorithm, is a DNF (disjunctive normal form, i.e. an OR-ing of AND-terms).

An interesting question would be if the triple dissecting algorithm is suitable to solve general DNFs, that means DNFs that do not have the special structure of the ClauseTable. The special structure of the ClauseTable is that if three clauses A, B, C are *not* in conflict pair-wise, they are guaranteed to be in one ClauseTable line, that means the three clauses do all appear AND-ed in one OR-term of the ClauseTable.

I examined if the triple dissecting works for any DNF and came to the conclusion: No, the triple dissecting algorithm does *only* work reliably when being used on the ClauseTable. Have a look at the following example of a general DNF that does not have the special structure of the ClauseTable:

```
(F & B & C & D) |
(A & F & C & D) |
(A & B & F & D) |
(A & B & C & F)
```

'F' shall be the only Boolean variable that is false, A, B, C and D shall be true. This means that we must treat them, in the 'triple dissecting algorithm' from topic 3.7, as if they were *not* in the SAT CNF.

The triple dissecting algorithm would do the following:

```
StillToBeChecked is initialized to all true,
StillToBeChecked_New is initialized to all false.

X = A
Y = B
Z = C
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[A][B] == true and StillToBeChecked[A][C] == true
       => StillToBeChecked_New[B][C] := true;

X = A
Y = B
Z = D
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[A][B] == true and StillToBeChecked[A][D] == true
       => StillToBeChecked_New[B][D] := true;

X = A
Y = C
Z = D
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[A][C] == true and StillToBeChecked[A][D] == true
       => StillToBeChecked_New[C][D] := true;


StillToBeChecked is AND-ed with StillToBeChecked_New.
So we get:

StillToBeChecked[B][C] == true
StillToBeChecked[B][D] == true
StillToBeChecked[C][D] == true


X = B
Y = C
Z = D
X, Y and Z are not in conflict. X, Y and Z do not appear in the SAT CNF.
Also: StillToBeChecked[B][C] == true and StillToBeChecked[B][D] == true
       => StillToBeChecked_New[C][D] := true;
```

```
StillToBeChecked is AND-ed with StillToBeChecked_New.
So we get:

StillToBeChecked[C][D] == true
```

As X, Y and Z reached their right-most position within the ClauseTable and there is a StillToBeChecked[C][D] == true with D from the last CT column (or corresponding here: the last variable in the DNF) and C from the column before, the triple dissecting algorithm says that the DNF is true. But this is wrong, the DNF is actually *not* true as it has, with the variable F, a 'false' in *each* AND-term.

I alleged the triple dissecting would *not* fail if the DNF would have the structure of a ClauseTable. But why should this be the case?

The answer is the following: As we said in topic 3.4:

"

The ClauseTable has a special structure:

- if a set of n many clauses are all not in conflict with each other, it is guaranteed that there is a ClauseTable line that contains all those n many clauses.

"

Watch carefully the DNF that made the triple dissecting algorithm fail:

```
(F & B & C & D) |
(A & F & C & D) |
(A & B & F & D) |
(A & B & C & F)
```

Within this DNF, there are the following clause tuples that are not in conflict among each other:

```
(A & B), [from DNF line 3 and 4]
(A & C), [from DNF line 2 and 4]
(A & D), [from DNF line 2 and 3]
(B & C), [from DNF line 1 and 4]
(B & D), [from DNF line 1 and 3]
(C & D). [from DNF line 1 and 2]
```

By finding all possible combinations of tuples we see that there are n many clauses that are not in conflict among each other, where n is the number of DNF 'columns'. So, in any ClauseTable, there *would* be a line that contains all those variables A to D, and the triple dissecting algorithm would *not* fail.

Due to the special structure of the ClauseTable DNF, the triple dissecting algorithm, as I strongly suppose, does always evaluate the ClauseTable DNF correctly.

3.12 Why my Algorithm might work while others failed

Why should my algorithm work while all previous attempts from other people failed?

I suppose nobody had the idea of the ClauseTable before. At least I didn't find anything similar when I briefly searched the internet using Google.de during November 2013.

As I mentioned, my 'triple dissecting' algorithm works only then when being used on a 'ClauseTable'. The 'triple dissecting' algorithm might fail when being used on general DNFs. In topic 3.11 I showed an example how you can 'fool' the triple dissecting algorithm when using it on a DNF that does not have the form of the ClauseTable. Probably other people already tried to solve general DNFs but no one handled 'ClauseTable' DNFs.

Even if anyone already implemented the ClauseTable somehow, the rest of his/her solver probably was not implemented like mine.

Notice that I did not (knowingly) extend any well-known algorithm but started from the ground up. So possibly I found an algorithm that has not been tested before.

4 Complexity

4.1 Three nested Loops

In topic 3.7 you were shown the pseudo code of the solver implementation.

The most work is done by

```
"
	foreach (PossibleClause X in PossibleClauses)
	{
		[...]

		foreach (PossibleClause Y in PossibleClauses)
		{
			[...]

			foreach (PossibleClause Z in PossibleClauses)
			{
				[...]
			}
		}
	}
"
```

what is practically implemented as three nested loops. You might have a look in the (C++) source code in the file 'ActualSolver.h', function 'Does_CNF_Have_A_Solution()'. There you see:

- The i-loop that points to the possible clause X.
  - The i-loop runs from 0 to PossibleClauseNumber - 1.
- The j-loop that points to the possible clause Y.
  - The j-loop runs from i+1 to PossibleClauseNumber - 1.
- The k-loop that points to the possible clause Z.
  - The k-loop runs from j+1 to PossibleClauseNumber - 1.

Please note that it would be possible to make all three loops, especially j and k, run the full 'range' of 0 to PossibleClauseNumber-1. So we can, for simplification, assume that each of the three loops, especially j and k, do theoretically all run PossibleClauseNumber many iterations. The practical C++ implementation doesn't do this only to speed up the solving.

4.2 PossibleClauseNumber

PossibleClauseNumber is the count of different clauses that can appear in an exact 2- or 3-SAT formula with DigitNumber possible literal indices.

As you can see in the code listing 2.5, the 3-SAT solver implementation uses 3 nested loops to create and store the possible clauses. The three nested loops define where in the current possible clause the 0/1 digits are. Furthermore there are 8 0/1 combinations. This is why the exact count of possible clauses for 3-SAT is:

$$PossibleClauseNumber\_Exact3SAT = \sum_{i=1}^{DigitNumber-2} \left(\sum_{j=i+1}^{DigitNumber-1} \left(\sum_{k=j+1}^{DigitNumber} 8 \right) \right)$$

For exact 2-SAT the solver uses two nested loops to create the possible clauses. There are 4 0/1 combinations. This is why the exact count of possible clauses for 2-SAT is:

$$PossibleClauseNumber\_Exact2SAT = \left(\sum_{i=1}^{DigitNumber-1} \left(\sum_{j=i+1}^{DigitNumber} 4 \right) \right)$$

4.3 Total Complexity

In my opinion it does not make much sense to find the formula to compute the very exact loop run count of the solver. The input has a big influence on the number of loop runs. For example, if the first X clause is absent, the j and k loops are eventually run through. If the first clause X is existing, the j and k loops are not run through at all. This makes it very hard and probably very confusion to give a formula for the very exact complexity for every possible SAT CNF.

Instead I want to specify an upper bound for the algorithm's complexity.

As mentioned in topic 4.1 there are three nested loops of which each iterates not more than PossibleClauseNumber times. Within the inner-most k loop, there are two calls to the function IsInConflict(), which might do another DigitNumber loop iterations. Within the j-loop, there is one further call to the function IsInConflict().

So in total there are never more than

$$O(PossibleClauseNumber^3 * 3 * DigitNumber)$$

loop runs, whereby PossibleClauseNumber can be denominated as $O(DigitNumber^3)$, what you can derive from the formula 'PossibleClauseNumber_Exact3SAT = ...' shown in the previous topic 4.2.

Further calculated, the worst case complexity of the polynomial solver in O-notation is:

$$O(DigitNumber^{10})$$

whereby DigitNumber is the count of different literal indices in the 3-SAT CNF to solve. Notice that the number of clauses in the SAT CNF does not have a significant influence on the total complexity as the iteration count of the main loops i, j and k depends from the DigitNumber only, not from the ClauseNumber. The same applies to the iteration count of the loop in IsInConflict().

4.4 Possible Speed-Up

Notice that there are improvements thinkable that speed up the solver. For instance, the calls to IsInConflict() could be pre-computed:

```
for (int m = 0; m < PossibleClauseNumber; m ++)
      for (int n = 0; n < PossibleClauseNumber; n ++)
            InConflict[m][n] = IsInConflict(
                  PossibleClauses[m],
                  PossibleClauses[n])
```

This code could be placed at beginning of the solver procedure. As further permanent calls to IsInConflict() are avoided, we could decrease the total complexity to:

$$O(DigitNumber^9) + O(PossibleClauseNumber^2 * DigitNumber)$$

The latter term is the pre-computing. We can simplify the optimized total complexity to:

$$O(DigitNumber^9) + O(PossibleClauseNumber^2 * DigitNumber) =$$
$$O(DigitNumber^9) + O((DigitNumber^3)^2 * DigitNumber) =$$
$$O(DigitNumber^9) + O(DigitNumber^7)$$

5 Proofs

5.1 ClauseTable Evaluation tells SAT Solvability

The following proof shall show that a SAT CNF has at least one solution if, in the corresponding ClauseTable, there is at least one ClauseTable line whose clauses do all *not* appear in the SAT CNF.

The first observation important for the proof is the following:
It is impossible to satisfy any SAT CNF that contains 4 (2-SAT) or 8 (3-SAT) or 2^X (X-SAT) clauses consisting of all 4 or 8 or X possible 0/1 combinations. A 2-SAT example:

```
00--
01--
10--
11--
```

can not be satisfied, what you realize when you try out all possible solutions. For none of those possible solutions you will manage to satisfy all clauses of the SAT CNF at once. For 3-SAT the example would be similar, with 8 instead of 4 clauses and 3 0/1 digits instead of 2 0/1 digits.

I want to show an example for an unsatisfiable 2-SAT CNF. First I list the 4 clauses that the 2-SAT CNF to solve consists of. After that I show one thinkable solution after another, whereby we realize that none of those thinkable solutions does really satisfy *all* clauses of the CNF.

1)

```
00-- <- satisfied
01-- <- satisfied
10-- <- satisfied
11-- <- not satisfied

00xx <- solution (xx shall mean not specified as unimportant)
```

2)

```
00-- <- satisfied
01-- <- satisfied
10-- <- not satisfied
11-- <- satisfied

01xx <- solution
```

3)

```
00-- <- satisfied
01-- <- not satisfied
10-- <- satisfied
11-- <- satisfied

10xx <- solution
```

4)

```
00-- <- not satisfied
01-- <- satisfied
10-- <- satisfied
11-- <- satisfied

11xx <- solution
```

But: if there is a ClauseTable line whose clauses are *all* not existing in the CNF to solve, then those clauses must not be satisfied by the solution of the CNF. This means that there is *no* set of 4 or 8 clauses that need all be satisfied! There are maximal 3 or 7 or (2^X)-1 clauses that need to be satisfied, what is do-able in any case, what you could verify by trying out all possible solutions according to the scheme just shown. Finally, please notice that all instances of one and the same clause are all set to true *or* false *everywhere* within the ClauseTable, even if that clause appears multiple times within the ClauseTable.

Here is an example that shall visualize this:

```
"00--", "-00-", "--00", "0-0-", "-0-0", "0--0"
"00--", "-00-", "--01", "0-0-", "-0-1", "0--1"
"00--", "-01-", "--10", "0-1-", "-0-0", "0--0"
"00--", "-01-", "--11", "0-1-", "-0-1", "0--1"
"01--", "-10-", "--00", "0-0-", "-1-0", "0--0"
"01--", "-10-", "--01", "0-0-", "-1-1", "0--1"
"01--", "-11-", "--10", "0-1-", "-1-0", "0--0"
"01--", "-11-", "--11", "0-1-", "-1-1", "0--1"
"10--", "-00-", "--00", "1-0-", "-0-0", "1--0"
"10--", "-00-", "--01", "1-0-", "-0-1", "1--1" <-    imagine all these clauses of this
                                                     ClauseTable line do NOT need to be
                                                     satisfied by the CNF's solution!
"10--", "-01-", "--10", "1-1-", "-0-0", "1--0"
"10--", "-01-", "--11", "1-1-", "-0-1", "1--1"
"11--", "-10-", "--00", "1-0-", "-1-0", "1--0"
"11--", "-10-", "--01", "1-0-", "-1-1", "1--1"
"11--", "-11-", "--10", "1-1-", "-1-0", "1--0"
"11--", "-11-", "--11", "1-1-", "-1-1", "1--1"
```

This means that:
10-- does not need to be satisfied, so we can satisfy 00--, 01--, 11--
-00- does not need to be satisfied, so we can satisfy -01-, -10-, -11-
--01 does not need to be satisfied, so we can satisfy --00, --10, --11
1-0- does not need to be satisfied, so we can satisfy 0-0-, 0-1-, 1-1-
-0-1 does not need to be satisfied, so we can satisfy -0-0, -1-0, -1-1
1--1 does not need to be satisfied, so we can satisfy 0--0, 0--1, 1--0

Possible solution: 0110 (the inverse of the solution of the marked CT line that is *not* to be satisfied).

So we came to the conclusion that we can satisfy each ClauseTable column's clauses by at least one solution if there's at least one clause in each ClauseTable column that needn't to be satisfied. If in each ClauseTable column, there's at least one clause that needn't to be satisfied, there's at least one ClauseTable line whose clauses do all not need to be satisfied, as all clauses of this ClauseTable line do not appear in the SAT CNF.

5.2 Non-Conflict Clauses build a ClauseTable Line

Imagine there are n clauses that are all not in conflict with each other. I allege that there *must* be a ClauseTable line that contains all those n clauses.

Proof:

- If the n clauses are all not in conflict with each other, all literals of the three clauses are either negated or not, but not both. In particular this applies to the literals the n clauses might have in common.
- This means that there's at least one assignment that satisfies all n clauses at once. 'To satisfy' shall mean the assignment is not in conflict with the digits of all n clauses. I call that assignment 'the solution'. For instance, "0011" is one solution for the ClauseTable line containing the clauses {00--, 0-1-, 0--1, -01-, -0-1, --11}. Mathematically "there's at least one solution" can be expressed by: It_Exists(Solution S) to satisfy the n clauses.
- For each possible solution there is exactly one ClauseTable line whose contained clauses are all satisfied by that solution. Mathematically this can be expressed by: It_Exists(ClauseTable line) for each Solution S.

As, according to [2], (It_Exists(A) for all B) and (It_Exists(B) so that C) => (It_Exists(A) so that C), we can say:

"It_Exists(ClauseTable line) for each Solution S." and
"It_Exists(Solution S) to satisfy the n clauses."
=>
"It_Exists(ClauseTable line) to satisfy the n clauses."

5.3 Algorithm detects ClauseTable Lines reliably

Remember what we said in topic 3.4:

"

The ClauseTable has a special structure:

- If a set of n many clauses are all not in conflict with each other, it is guaranteed that there is a ClauseTable line that contains all those n many clauses.

"

If n clauses are not in conflict, this means that all tuples created out of those n clauses contain only clauses that are not in conflict pair-wise. Please attend also topic 3.11.

I want to show that the algorithm reliably detects if there are enough tuples existing to build a ClauseTable line. I want to do this in the form of an induction proof. But for simplification, instead of handling clauses from a ClauseTable, I will use Boolean variables from one AND-term containing all those Boolean variables. So the algorithm shall detect that the tuples built out of the Boolean variables build the whole AND-term (the AND-term stands for one line from the ClauseTable).

First, we show that the algorithm works when receiving three tuples, then we add a further Boolean variable so that we get more tuples.

Induction Basis: Let's start with a set of three Boolean variables {A, B and C} from the AND-term (A & B & C). Assume we have the tuples:

(A & B), (A & C) and (B & C).

The algorithm was described in topic 3.7 like this:

```
bool Does_SAT_CNF_Have_A_Solution(argument CNF)
{
	ClauseType PossibleClauses[];			// define (reserve memory)
	CreatePossibleClauses(PossibleClauses[]);	// fill with data (see topic 2.5!)


	bool StillToBeChecked[][];			// define (reserve memory)
	bool StillToBeChecked_New[][];			// define (reserve memory)

	StillToBeChecked[][] = initialize with all true;

	foreach (PossibleClause X in PossibleClauses)
	{
		if (X is the first clause in a ClauseTable column)
			StillToBeChecked_New[][] = all false;

		foreach (PossibleClause Y in PossibleClauses)
		{
			foreach (PossibleClause Z in PossibleClauses)
			{
				if (	IsInConflict(X, Y) == false &
					IsInConflict(X, Z) == false &
					IsInConflict(Y, Z) == false &
					IsInCNF(X) == false &
					IsInCNF(Y) == false &
					IsInCNF(Z) == false &
					StillToBeChecked[X][Y] == true &
					StillToBeChecked[X][Z] == true)
				{
					StillToBeChecked_New[Y][Z] = true;
				}
			}
		}

		if (X is the last clause in a ClauseTable column)
			StillToBeChecked[][] =
			StillToBeChecked[][] AND StillToBeChecked_New[][];
	}
```

```
        if (there is any StillToBeChecked[A][B] == true with
                B a PossibleClause from last ClauseTable column and
                A a PossibleClause from the ClauseTable column before B)
                return CNF_Has_A_Solution;
        else
                return CNF_Has_No_Solution;
}
```

So let's use this algorithm on our three example variables:

```
X = A
Y = B
Z = C
IsInConflict(A, B) is false (by definition, as X=A and Y=B are within a tuple),
IsInConflict(A, C) is false (by definition, as X=A and Z=C are within a tuple),
IsInConflict(B, C) is false (by definition, as Y=B and Z=C are within a tuple),
IsInCNF(A) == false (by definition, see first paragraphs of this topic),
IsInCNF(B) == false (by definition, see first paragraphs of this topic),
IsInCNF(C) == false (by definition, see first paragraphs of this topic),
StillToBeChecked[A][B] is true as initialized like this,
StillToBeChecked[A][C] is true as initialized like this.

So we set StillToBeChecked_New[B][C] to true.

StillToBeChecked[B][C] becomes true as it was initialized with true and also
StillToBeChecked_New[B][C] has been set to true.
```

The algorithm realized that there's StillToBeChecked[B][C] == true with C from the last ClauseTable column and B from the ClauseTable column before (or, in this example, the last variable in the AND-term, and the one before). So the algorithm rightfully says that the SAT CNF is solvable, what does in the current case mean that the algorithm detected that all tuples creatable out of the three variables A, B and C do exist.

To prove the algorithm for correctness, I want to transfer the just shown algorithm run into a mathematical notation. This shall be done as follows:

- Let (X & Y) mean that the Boolean variables X and Y appear as a tuple, i.e.
  - X and Y do both not appear in the SAT CNF,
  - X and Y are not in conflict,
  - StillToBeChecked[X][Y] == true.
- Let (X & Y) & (X & Z) mean:
  - (X & Y) appears as tuple (see point before) AND
  - (X & Z) appears as tuple (see point before).

The just shown algorithm run with three tuples can be described mathematically like this: The algorithm first checks if `(A & B) & (A & C) & (B & C)` is true. In the current case, it is, as the existence of the tuples shall mean that the contained variables are all true and not in conflict with each other. So the algorithm sets StillToBeChecked_New[B][C], what does practically lead to the algorithm's output that A, B and C appear as tuples in any combination.

Induction Hypothesis: The algorithm still detects reliably if all tuples exist when adding another Boolean variable.

Inductive Step: Now, let's assume we add a further variable, called D. So the algorithm has to process the following tuples:

(A & B), (A & C), (A & D), (B & C), (B & D), (C & D)

from the AND-term (A & B & C & D). Now an important question is, **would the algorithm notice if at least one of those tuples was missing?** In the practical usage scenario, a tuple would be missing if at least one clause represented by a variable within the tuple would exist in the SAT CNF to solve. Will the algorithm detect this?

When we start the algorithm again, it does the following steps:
The algorithm checks if `(A & B) & (A & C) & (B & C)` is true. This is the case. So the algorithm sets StillToBeChecked_New[B][C] to true.

Next, the algorithm checks if `(A & B) & (A & D) & (B & D)` is true. Also this is the case. So the algorithm sets StillToBeChecked_New[B][D] to true.

Next, the algorithm checks if `(A & C) & (A & D) & (C & D)` is true. Again, this is the case. So the algorithm sets StillToBeChecked_New[C][D] to true.

Now, the pointer to the first variable, X=A moves one column further, that means X now points to B.

We AND StillToBeChecked[][], which is initially all true, with StillToBeChecked_New[][].

Next, the algorithm checks if `(B & C) & (B & D) & (C & D)` is true.

At this point, an important thing happens:

- Remember that the notation (B & C) means:
  - B and C do both not appear in the SAT CNF,
  - B and C are not in conflict,
  - StillToBeChecked[B][C] == true.

This means that the whole expression '`(B & C) & (B & D) & (C & D)`' can *only* be true if

- StillToBeChecked[B][C] == true and
- StillToBeChecked[B][D] == true and
- StillToBeChecked[C][D] == true.
- For instance, this means that (B & C) is only further processed if StillToBeChecked[B][C] == true.

We have set StillToBeChecked[B][C], StillToBeChecked[B][D], StillToBeChecked[C][D] when X has pointed to A, please see previous paragraphs. For instance, StillToBeChecked[B][C] could only have become true if (A & B) & (A & C) & (B & C).

So we could deploy:
`(B & C) & (B & D) & (C & D)`
to:

```
(((A & B) & (A & C) & (B & C)) & (B & C)) &
(((A & B) & (A & D) & (B & D)) & (B & D)) &
(((A & C) & (A & D) & (C & D)) & (C & D))
```

You can see that the algorithm checked all tuples
(A & B), (A & C), (A & D), (B & C), (B & D), (C & D),
as required.

Notice that all those tuples are AND-ed, that means if *only one* tuple would *not* exist the whole formula becomes false, as we expected.

When you would add further variables, e.g. E, F, G and so on, nothing changes about *the way* the algorithm checks if sufficiently many tuples are existing to build a ClauseTable line. There would be more steps necessary, but the algorithm still functions correctly.

6 Meaning of the Algorithm

If the polynomial SAT-solving algorithm described in this document should turn out to be really correct, it would solve the P-NP-Problem (one of the 'Millennium Problems') by proving P = NP.

7 Acknowledgements

I thank the reader for his/her interest in my work. If you do not understand anything about my algorithm, feel free to mail me so that I can improve the document(s) or source code. Unfortunately I could not use 'third party beta testers' before publishing as those people would possibly had stolen my ideas. It is from big interest for me to make the reader understand my algorithm, so I would really appreciate some feedback.

8 Summary

This document explained an algorithm that is supposed to decide in polynomial time and space if an exact 2- or 3-SAT formula has a solution or not.

9 References

9.1 General Literature

- Michael R. Garey and David S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, W. H. Freeman & Co., 1979.
- Christos H. Papadimitriou, Computational complexity, Addison-Wesley, 1994.
- Uwe Schöning, Theoretische Informatik - kurz gefasst, Bibl. Institut Wissenschaftsverlag, 1992, ISBN 3-411-15641-4.
- Ingo Wegener, Theoretische Informatik - eine algorithmenorientierte Einführung (3. Auflage), B. G. Teubner Verlag / GWV Fachverlage GmbH, Wiesbaden 2005, ISBN 3-8351-0033-5.
- Volker Heun, Grundlegende Algorithmen (2. Auflage), Friedr. Vieweg & Sohn Verlag / GWV Fachverlage GmbH, Wiesbaden 2003, ISBN 3-528-13140-3.
- Daniel Grieser, Mathematisches Problemlösen und Beweisen, Springer Fachmedien Wiesbaden 2013, ISBN 978-3-8348-2459-2.
- Bronstein, Semendjajew, Musiol, Mühling, Taschenbuch der Mathematik, Verlag Harri Deutsch, Thun und Frankfurt am Main 2000, ISBN 3-8171-2015-X.
- http://en.wikipedia.org/wiki/3-SAT (accessed 2013-11-23).

9.2 Concrete References

- [1] http://www.ti.inf.ethz.ch/ew/courses/extremal04/raemy.pdf (accessed 2013-12-08).
- [2] http://de.wikipedia.org/wiki/Schlussfolgerung (accessed 2013-12-08, English version also existing, see ‘other languages‘ at left edge of webpage).