International Journal of Engineering & Technology, 9 (3) (2020) 670-691

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Polynomial Exact-3-SAT-Solving Algorithm

Matthias Michael Mueller[1*]

[1]*Independent, Germany*
[*]*matthias.mueller@louis-coder.com*

Abstract

This article describes an algorithm which is supposed by the author to be capable of solving any instance of a 3-SAT CNF in maximal $O(n^{15})$, whereby n is the variable index range within the 3-SAT CNF to solve. The presented algorithm imitates the proceeding of an exponential, fail-safe solver. This exponential solver stores internal data in m-SAT clauses, with $3 \leq m \leq n$. The polynomial solver works similarly, but uses 3-SAT clauses only to save the same data. The paper explains how, and proves why this can be achieved. On 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.

Keywords: *2-SAT; 3-SAT; Complexity; P-NP-problem; SAT solver; Satisfiability.*

1. Introduction

Problems denoted as 'NP-complete' are those algorithms which need an amount of computing time or space which grows exponentially with the problem size n. If it should be accomplished to solve in general at least one of those problems in polynomial time and space, all NP-complete problems out of the complexity class NP could from then on be solvable much more efficiently. Finding the answer to the open question if such a faster computation is possible at all is called the P versus NP problem.[6, p. 148]
The present article describes an algorithm which is supposed by its author to solve the NP-complete problem 'exact-3-SAT' in polynomial time and space. This persuasion comes from the fact that the implementation of the algorithm correctly solved absolutely every 3-SAT CNF which was given as input. Those 3-SAT CNFs had a size n of up to 26, in the course of around 6 months at least one million of them have been processed with always correct output. Of course one must be skeptical, many attempts to build a polynomial 3-SAT solver failed in the early stages or the related algorithm did later turn out to be incorrect. The author of this paper does nevertheless believe the presented algorithm is relevant to the scientific community, because it did never fail and because it is extremely simple and can therefore easily be further examined by any interested reader. In case it should be really correct, the P-NP problem would be solved with the result P = NP.

2. Definitions

The following definitions will be used to describe and analyze the polynomial algorithm. Their purpose will become accessible in the course of the document. Some definitions are presented for the 3-SAT case only. The reader should be able to transfer them to 2-SAT easily, by just downsizing the figure shown in the definition (by leaving one literal off, for instance). The 2-SAT variants are occasionally used in some document passages.

2.1. 3-SAT CNF

Given is a formula in conjunctive normal form:

$$CNF = \bigwedge_{i=1}^{\gamma} (\varepsilon_{i1} x_{i1} \vee \varepsilon_{i2} x_{i2} \vee \varepsilon_{i3} x_{i3})$$

This 3-SAT CNF, in this document often just called 'CNF', consists of γ many conjugated clauses. Within each clause, $x_{i1}, x_{i2}, x_{i3} \mid i_1, i_2, i_3 \in \{1, ..., n\}$ are disjunctive Boolean variables. There are always exactly three variables in each clause. Each variable has assigned an epsilon: $\varepsilon_{i1}, \varepsilon_{i2}, \varepsilon_{i3} \in \{0, 1\}$. If this $\varepsilon_{ia} \mid a \in \{1, 2, 3\}$ has the value 0, then the value of x_{ia} is negated when evaluating the CNF. If $\varepsilon_{ia} \mid a \in \{1, 2, 3\}$ has the value 1, no negation is done. Each epsilon and its related variable form a 'literal'. The variable indices are chosen out of a set of

natural numbers $\{1,...,n\}$. The variable indices are pair-wise distinct: $i_2 \neq i_1, i_3 \neq i_1, i_3 \neq i_2$. n will be used as description of the problem size in the upcoming analysis of the solver's complexity. The task of the presented polynomial solver is to find out if there is an assignment of either true or false to each literal so that a given CNF as a whole evaluates to true. Then we say this assignment, synonymously called solution or, in literature, 'model'[6, p. 14], satisfies the CNF. If the CNF evaluates to true, we say the CNF is 'solvable'. If it is not possible to satisfy the CNF, we say the CNF is 'unsatisfiable', or synonymously 'UNSAT'. The solvability must be determined by the solver in polynomial time and space. The solver only says *if* there is a solution or not, it does not output an assignment. This is still sufficient to solve the P versus NP problem.[6, p. 149]

Examples:

- We suppose it is given:
  $CNF = (0x_{i1} \vee 0x_{i2} \vee 0x_{i3}) \wedge (1x_{i1} \vee 1x_{i2} \vee 0x_{i3}) \wedge (0x_{i1} \vee 1x_{i2} \vee 1x_{i3})$.
  This formula is solved, among others, by the solution $x_{i1} = true$ and $x_{i2} = false$ and $x_{i3} = true$ because when we insert these Boolean values, we get:
  $CNF = (\neg true \vee \neg false \vee \neg true) \wedge (true \vee false \vee \neg true) \wedge (\neg true \vee false \vee true)$.
  When evaluating this formula with the common rules of Boolean logic, we get $CNF = true$.
- A simple example of an unsatisfiable formula is:
  $CNF =$
  $(0x_{i1} \vee 0x_{i2} \vee 0x_{i3}) \wedge (0x_{i1} \vee 0x_{i2} \vee 1x_{i3}) \wedge (0x_{i1} \vee 1x_{i2} \vee 0x_{i3}) \wedge (0x_{i1} \vee 1x_{i2} \vee 1x_{i3}) \wedge$
  $(1x_{i1} \vee 0x_{i2} \vee 0x_{i3}) \wedge (1x_{i1} \vee 0x_{i2} \vee 1x_{i3}) \wedge (1x_{i1} \vee 1x_{i2} \vee 0x_{i3}) \wedge (1x_{i1} \vee 1x_{i2} \vee 1x_{i3})$.
  This formula can never evaluate to true, no matter how the value of each of the Boolean variables x_{i1}, x_{i2}, x_{i3} is chosen. The reader can verify this by just trying out all 8 possible solutions. The presented CNF is an easy example for an unsatisfiable 3-SAT clause, practically important CNFs typically are more opaque.

2.2. Notation convention

Within this document, a fixed notation convention for clauses is used:

- The index of any variable in a clause is noted in lowercase letters. For example, i_1, i_2 and i_3 are the indices of variables in the clause $(\varepsilon_{i1}x_{i1} \vee \varepsilon_{i2}x_{i2} \vee \varepsilon_{i3}x_{i3})$. The variable index as a whole, i.e. letter (here: i) plus the numeration (1, 2, 3), contains a natural number, which is out of the range $\{1,...,n\}$. This means for the example: $i_1 \in \{1,...,n\}$, $i_2 \in \{1,...,n\}$ and $i_3 \in \{1,...,n\}$.
- The related clause name is identified by a lowercase letter as well. This letter does always match the letter of the index of the variables. For example, $i = (\varepsilon_{i1}x_{i1} \vee \varepsilon_{i2}x_{i2} \vee \varepsilon_{i3}x_{i3})$. I decided to use lowercase letters for clause names as well because the source code of the exact-3-SAT solving algorithm (which is later presented) uses lowercase letters. This is done for consistency, because in the source code, all clause names are variables which contain array indices, and these variables are empirically not capitalized by most programmers.

2.3. Possible clauses

The set PC of possible clauses is defined as:

$$PC = \{(\varepsilon_{i1}x_{i1} \vee \varepsilon_{i2}x_{i2} \vee \varepsilon_{i3}x_{i3})\}$$

with $\varepsilon_{i1}, \varepsilon_{i2}, \varepsilon_{i3} \in \{0,1\}, i_1, i_2, i_3 \in \{1,...,n\}, i_2 \neq i_1, i_3 \neq i_1, i_3 \neq i_2$. All possible choices of these epsilon and i values appear once in the set PC. $PCNum = |PC| = (2^3 \times \binom{n}{3})$, because there are $2^3 = 8$ epsilon combinations and $\binom{n}{3}$ possibilities to choose 3 distinct variable indices out of $\{1,...,n\}$.

Example:

- For $n = 3$, the set of possible clauses is:
  $PC = \{$
  $(0x_1 \vee 0x_2 \vee 0x_3)$,
  $(0x_1 \vee 0x_2 \vee 1x_3)$,
  $(0x_1 \vee 1x_2 \vee 0x_3)$,
  $(0x_1 \vee 1x_2 \vee 1x_3)$,
  $(1x_1 \vee 0x_2 \vee 0x_3)$,
  $(1x_1 \vee 0x_2 \vee 1x_3)$,
  $(1x_1 \vee 1x_2 \vee 0x_3)$,
  $(1x_1 \vee 1x_2 \vee 1x_3)$
  $\}$.

2.4. Active solutions

An active solutions is defined as a string of n many characters (abbreviated 'chars'), whereby each char can be chosen out of $\{\texttt{0},\texttt{1},\texttt{-}\}$:

$$u = \{\texttt{0},\texttt{1},\texttt{-}\}^n$$

Examples:

- $u = \texttt{0000}$, $u = \texttt{0001}$, $u = \texttt{000-}$, $u = \texttt{00----}$, $u = \texttt{10--10}$, and so on.

2.5. In conflict

Two clauses $j,k \in PC$ are said to be 'in conflict' if they have at least one variable index in common and the epsilons concerned are not equal:

$$(j \not\equiv k) \Leftrightarrow \exists (h \in \{1,2,3\}, i \in \{1,2,3\} \mid ((\varepsilon_{j_h} = 0 \wedge \varepsilon_{ki} = 1) \vee (\varepsilon_{j_h} = 1 \wedge \varepsilon_{ki} = 0)) \wedge (j_h = k_i))$$

Similar is defined for a clause $j \in PC$ and an active solution u:

$$(j \not\equiv u) \Leftrightarrow \exists (i \in \{1,2,3\} \mid ((\varepsilon_{j_i} = 0 \wedge u_{j_i} = 1) \vee (\varepsilon_{j_i} = 1 \wedge u_{j_i} = 0)))$$

We define 'not in conflict' as $(j \equiv k) \Leftrightarrow \neg(j \not\equiv k)$ resp. $(j \equiv u) \Leftrightarrow \neg(j \not\equiv u)$. Furthermore it is defined for a clause $c \in PC$ and an active solution u: "The active solution u contains c", "c appears in the active solution u", "c is in the active solution u" or "c from the active solution u" $\Rightarrow c \equiv u$ (i.e. c is not in conflict with u).

Examples:

- $j = (0x_1 \vee 0x_2 \vee 0x_3)$ and $k = (0x_2 \vee 1x_3 \vee 0x_5)$ are in conflict,
  because x_3 has an epsilon value ε_{x3} of 0 in j and an epsilon value ε_{x3} of 1 in k.
  So it applies: $j \not\equiv k$.
- $j = (0x_1 \vee 1x_2 \vee 0x_3)$ and $k = (0x_2 \vee 1x_3 \vee 0x_5)$ are in conflict,
  because there are even two conflicts between both clauses' ε_{x2} and ε_{x3}.
  So it applies: $j \not\equiv k$.
- $j = (0x_1 \vee 0x_2 \vee 0x_3)$ and $k = (0x_2 \vee 0x_3 \vee 0x_5)$ are not in conflict.
  So it applies: $j \equiv k$.
- $j = (0x_1 \vee 0x_2 \vee 0x_3)$ and $k = (0x_4 \vee 0x_5 \vee 0x_6)$ are also not in conflict.
  So it applies: $j \equiv k$.
- $j = (0x_1 \vee 0x_2 \vee 0x_3)$ and $u = \texttt{0100}$ are in conflict,
  because the char number 2 in u differs from the epsilon value of x_2 in j.
  So it applies: $j \not\equiv u$.
- $j = (0x_1 \vee 0x_2 \vee 0x_3)$ and $u = \texttt{0000}$ are not in conflict.
  So it applies: $j \equiv u$.

2.6. Clause line notation

Definition 2.6.1. *The clause line notation of a clause shall be a string consisting of n chars, whereby each char can be chosen out of* $\{\texttt{0},\texttt{1},\texttt{-}\}$. *For x-SAT, any clause line contains exactly x many 0 or/and 1 chars and exactly* $(n-x)$ *many '-' chars. x is here any natural number, in this document mostly 2 or 3.*

A classical, mathematical clause known from general literature can be converted into a clause line as follows: The clause line is initially a string $\{\texttt{-}\}^n$. Then regard the three literals of the mathematic clause one after another. If a literal is negated, place '0' at the location[1] within the clause line indicated by the literal index (the initial '-' chars are overridden). If the literal of the mathematical clause is not negated, act equally but place a '1'.

Examples for $n = 5$:

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

The reverse direction, clause line notation to mathematical disjunction notation, works accordingly: The position of a 0 or 1 char in the clause line is the variable index in the disjunction, only if the char in the clause line is a 0, the variable in the disjunction is negated.
Clauses are, within this document, partially noted in clause line notation. I decided to do this to get a more characteristic visualization which is easier to understand than similar-looking, indexed 'x'-variables. Remark: It can easily be determined if two clauses j,k in clause line notation are in conflict. They are in conflict if j has a 0 at some position where k has a 1, or vice versa.

Examples:

- $j = \texttt{000---}$ is in conflict with $k = \texttt{111---}$
- $j = \texttt{--000-}$ is in conflict with $k = \texttt{101---}$
- $j = \texttt{000---}$ is in conflict with $k = \texttt{--100-}$
- $j = \texttt{010---}$ is in conflict with $k = \texttt{00--0-}$

But:

- $j = \texttt{000---}$ is not in conflict with $k = \texttt{--000-}$
- $j = \texttt{--100-}$ is not in conflict with $k = \texttt{1-1-0-}$
- $j = \texttt{111---}$ is not in conflict with $k = \texttt{---000}$
- $j = \texttt{111---}$ is not in conflict with $k = \texttt{111---}$

[1] The x-th location within a clause in clause line notation is always defined as the x-th char from left.

When there is an (active) solution given in clause line notation, then this notation can be transformed to mathematical notation by replacing each 0 at position p by $x_p = false$ and each 1 at position p by $x_p = true$.

Example:

- $u = 0011$ satisfies a clause or CNF $\rightarrow x_1 = false, x_2 = false, x_3 = true, x_4 = true$ satisfies the clause or CNF.

2.7. 2oo()

Definition 2.7.1. *2oo(n-SAT clause c) shall be defined as the set of all 2-SAT clauses which are not in conflict with c.*

Examples:

- $2oo(000) = \{00-,0-0,-00\}$
- $2oo(010) = \{01-,0-0,-10\}$
- $2oo(0000) = \{00--,0-0-,0--0,-00-,-0-0,--00\}$
- $2oo(0101) = \{01--,0-0-,0--1,-10-,-1-1,--01\}$

2.8. Clause path

The clause path is a visualization of the set of possible clauses. One gets the clause path by just writing all possible clauses one below the other.

Examples:

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

This is the clause path for 2-SAT, *n*=4. The same for 3-SAT, *n*=4 is:

```
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
```

A block within any clause path shall be any subset of 4 (for 2-SAT clause paths) or 8 (for 3-SAT clause paths) possible clauses which have all their 0/1 chars at the same positions.
Examples:

```
00--    block 1
01--    block 1
10--    block 1
11--    block 1
0-0-    block 2
0-1-    block 2
1-0-    block 2
1-1-    block 2
0--0    block 3
0--1    block 3
1--0    block 3
1--1    block 3
-00-    block 4
-01-    block 4
-10-    block 4
-11-    block 4
-0-0    block 5
-0-1    block 5
-1-0    block 5
-1-1    block 5
--00    block 6
--01    block 6
--10    block 6
--11    block 6
```

The same for 3-SAT, n=4:

```
000-    block 1
001-    block 1
010-    block 1
011-    block 1
100-    block 1
101-    block 1
110-    block 1
111-    block 1
00-0    block 2
00-1    block 2
01-0    block 2
01-1    block 2
10-0    block 2
10-1    block 2
11-0    block 2
11-1    block 2
0-00    block 3
0-01    block 3
0-10    block 3
0-11    block 3
```

```
1-00    block 3
1-01    block 3
1-10    block 3
1-11    block 3
-000    block 4
-001    block 4
-010    block 4
-011    block 4
-100    block 4
-101    block 4
-110    block 4
-111    block 4
```

The SAT solving algorithms presented in this paper do all work for any block order. This means it is valid to shuffle whole blocks, but the clauses in each block must stay in one and the same block.
Example:

```
00--    block 1
01--    block 1
10--    block 1
11--    block 1
0--0    block 2
0--1    block 2
1--0    block 2
1--1    block 2
0-0-    block 3
0-1-    block 3
1-0-    block 3
1-1-    block 3
-00-    block 4
-01-    block 4
-10-    block 4
-11-    block 4
--00    block 5
--01    block 5
--10    block 5
--11    block 5
-0-0    block 6
-0-1    block 6
-1-0    block 6
-1-1    block 6
```

is valid, but

```
00--    block 1
01--    block 1
0-0-    block 2
0-1-    block 2
0--0    block 3
0--1    block 3
10--    block 1
11--    block 1
1-0-    block 2
1-1-    block 2
1--0    block 3
1--1    block 3
...
```

is not valid. Shuffling clauses within a block would not confuse the presented solvers, but does not make sense because then the clause path is more ‘chaotic’ and thus harder to understand (for the human examiner). Although there is this theoretical support for different block orders, this shuffling is not of use for the polynomial algorithm and will not be further examined in this document. It is just mentioned here for the sake of completeness.

2.9. Clause path column

The 'x-th clause path column' shall be similarly defined like a matrix column: It just denotes the set of chars in the x-th column (i.e. all chars at the x-th position from left) of the clause path.

Example:

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

↑↑↑↑
1234 ← index of the clause path column (1: first column, 2: second column etc.)
```

3. Derivation of the polynomial algorithm

An exponential solver will be introduced. This exponential solver is easy to understand and easy to verify. It will be shown that the exponential solver detects the solvability of any 2- or 3-SAT CNF reliably. Subsequently, it will be pointed out that the idea of the exponential solver (extending/removing active solutions) can be implemented similarly in polynomial time and space as well. Although this document's title announces a 3-SAT solver, many examples show a 2-SAT clause path, a 2-SAT CNF or 2-SAT clauses. This is done for simplicity, 2-SAT examples are mostly easier to understand and need much less document space. It should be easily achievable for the reader to transfer the 2-SAT examples to 3-SAT ones. When the 3-SAT case of an explanation or example is not obvious, it is declared directly.

3.1. Exponential Solver

The exponential solver works as follows:

- Add one initial active solution u_{init} with no 0/1 chars (for the succeeding example u_{init} = `----`).
- Walk the clause path, clause by clause (top-down).
- For each current clause i:
  - If clause i does not appear in the CNF:
    - Loop through the active solutions.
    - If i is not in conflict with the current active solution u, then 'extend' u:
      - Create a copy of u,
      - add all 0/1 chars of i to u. If u does already consist of n many 0/1 chars, keep it untouched (u still counts as extended in this case).
- If i is the last clause in the current block: Delete all active solutions which were already existing when walking arrived at the end of the prior block. This means, only keep extended active solutions.
- If at the end of the last block there's an active solution left, then the CNF is solvable, else it is unsatisfiable.

Example (2-SAT, n=4 for reasons of space and for convenience; 3-SAT and any n analogous):

Clause path:

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

Clause path with clauses marked which appear in the example CNF:

CNF:

```
00--
01--
1-0-
1--1
-0-1
-1-1
```

Clause path:

```
00--            <- CNF clause
01--            <- CNF clause
10--
11--
0-0-
0-1-
1-0-            <- CNF clause
1-1-
0--0
0--1
1--0
1--1            <- CNF clause
-00-
-01-
-10-
-11-
-0-0
-0-1            <- CNF clause
-1-0
-1-1            <- CNF clause
--00
--01
--10
--11
```

Clause path with active solutions at end of each block:

```
00--            <- CNF clause
01--            <- CNF clause
10--
11-- 10--,11--
0-0-
0-1-
1-0-            <- CNF clause
1-1- 101-,111-
0--0
0--1
1--0
1--1 1010,1110  <- CNF clause
-00-
-01-
-10-
-11- 1010,1110
-0-0
-0-1            <- CNF clause
-1-0            <- CNF clause
-1-1 1010       <- CNF clause
--00
--01
--10
--11 1010
```

The exponential solver found out that the CNF has a solution, because the active solution `1010` remains. We must invert the active solution and get the final solution ('model') for the CNF. We get the inverse solution by just toggling all 0 to 1 and all 1 to 0: `1010` $\rightarrow$ `0101`. The reason for the inversion is that any remaining active solution does exactly *not* satisfy the CNF. This will be explained in the proof of an upcoming claim (see 3.1.1). We verify `0101`: Each clause from the CNF and each inverted solution must have at least one 0/1 char in common, at the same position. This applies to `0101`:

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

The 0/1 chars marked by underlining appear in the inverted solution `0101`, at the same positions.

Claim 3.1.1. *Given the proceeding of the exponential solver as just introduced. If at least one active solution rests at the end of clause path walking, then the CNF is solvable. Else, it is unsatisfiable.*

Proof:

1) In case the CNF is solvable, it has some solution (i.e. some 'model'). Then all those possible clauses will not appear in the CNF which are satisfied by the inverse solution.

Example:

- Assumed the solution ('model') of the CNF is `0000` then the following clauses do *not* appear in the CNF:

  ```
  11--,
  1-1-,
  1--1,
  -11-,
  -1-1,
  --11.
  ```

  They do not appear in the CNF because none of them would be satisfied by the solution `0000`. That none of the six clauses is satisfied by the solution is seeable when transforming them back into mathematical notation: The clause `11--` is in mathematical notation: $(x_1 \vee x_2)$. The solution `0000` is in mathematical notation: $x_1 = false, x_2 = false, x_3 = false, x_4 = false$. When inserting the solution into the clause: $(false \vee false) = false$. So the solution does not satisfy the clause `11--`.
  In clause line notation, one sees that a solution satisfies a clause if the solution and the clause both have a 0 char at the same position, or if they have a 1 char at the same position. If the solution is completely (at all 0/1 chars) in conflict with the clause, then the solution does not satisfy the clause. Again, this applies to the clause line notation.

So we know that when the CNF is solvable, all those clauses $C = \{c_1, c_2, ...\}$ from the clause path do not appear in the CNF which are not in conflict with the inverse solution. This means that the clauses from C are not in conflict pair-wise, because they are all also not in conflict

with the inverse solution. This means in return that in each block of the clause path, there is one clause (out of C) which is not in conflict with at least one clause (also out of C) from each other block, and those clauses do all not appear in the CNF. Therefore, the exponential solver will add all clauses from C to an active solution. At the end of the clause path walking, this active solution is exactly the inverse solution. The exponential solver will output 'solvable', because there's an active solution left at the end of the clause path walking. This is the result we want.
2) In case the CNF is unsatisfiable: Proof by contradiction: When the CNF is unsatisfiable, then there must be for *each* thinkable inverse solution at least one clause in the CNF which is not in conflict with that inverse solution. Else the CNF is solvable, as priorly explained in 1). This means the exponential solver cannot keep any active solution active until the end of the clause path.
The reason therefore is that the inverse of any active solution can be seen as a solution ('model') for the CNF which is extended with each newly crossed clause path block. When the exponential solver arrives (in the clause path) at this clause which appears in the CNF and which is not in conflict with an active solution, then this active solution will be removed. This will happen for *all* thinkable active solutions, because for each of those active solutions there is a non-conflicting clause in the CNF. This has been pointed out at the beginning of this proof part 2). So no active solution will have remained at the end of the solving process and thus the exponential solver will output 'UNSAT'. □

3.2. Polynomial solver

We cannot save the active solutions in memory, for the following reason: Active solutions are m-SAT clauses. In practice (in tests done by me), very large m appeared, in extreme cases the 0/1 char count in some active solutions were close or equal to the variable index range n of the CNF to solve. In this case, the exponential solver might store exponentially many active solutions. Therefore, the idea of the polynomial solver is to check if there 'would be' an active solution existing (in an imagined parallel run of the exponential solver) which contains (i.e. is not in conflict with) the current clause i, whereby i is the currently regarded clause in the clause path. The polynomial algorithm works as follows:

- There is a 1-d, Boolean array "`bool Active[PCNum];`" with one true/false value for each possible clause.
  - Active[c] tells if clause c would still be in at least one active solution.
  - We say a clause c is 'active' if and only if it applies: Active[c] = true.
- Initially, all array values are set to true.
- Walk the clause path, clause by clause (top-down).
- For each current clause i:
  - If i appears in the CNF:
    - Set Active[i] := false.
  - Else regard each combination of three blocks from the clause path:
    - If in at least one combination of three blocks there's *not* in *each* of the three blocks one clause which is CNF-absent, active and is not in conflict with i, then set Active[i] := false.
      - The clauses from the three blocks are named j,k,l.
      - j,k,l must all not be in conflict with i, and also not in conflict pair-wise.
- Repeat the complete walking procedure until no Active[] value has been changed any more.
- If in the last block there's at least one Active[] = true left, then the CNF is solvable, else it is unsatisfiable.

To clarify the linguistic description, regard the source code of the polynomial solver:

```
bool Active[POSSIBLE_CLAUSE_NUMBER_MAX];

for (int i = 0; i < PossibleClauseNumber; i ++)
    Active[i] = true;

bool ChangesExisting = true;

while (ChangesExisting)
{
    const int BlockSize = 8; // The count of clauses in one block: 8 for 3-SAT (4 for 2-SAT)

    ChangesExisting = false;

    for (int i = 0; i < PossibleClauseNumber; i ++)
    {
        if (Active[i])
        {
            if (ExistsInCNF(i))
            {
                Active[i] = false;
                ChangesExisting = true;
                goto Next_i;
            }

            bool Found_j = false;
```

```
            for (int j = 0; j < PossibleClauseNumber; j ++)
            {
                if ((j % BlockSize) == 0) // first clause in block?
                {
                    Found_j = false;
                }

                if (Active[j] && !ExistsInCNF(j) && // ! means 'not' (i.e. negation)
                    !IsInConflict(i, j))
                {
                    bool Found_k = false;

                    for (int k = 0; k < PossibleClauseNumber; k ++)
                    {
                        if ((k % BlockSize) == 0)
                        {
                            Found_k = false;
                        }

                        if (Active[k] && !ExistsInCNF(k) &&
                            !IsInConflict(i, k) &&
                            !IsInConflict(j, k))
                        {
                            bool Found_l = false;

                            for (int l = 0; l < PossibleClauseNumber; l ++)
                            {
                                if ((l % BlockSize) == 0)
                                {
                                    Found_l = false;
                                }

                                if (Active[l] && !ExistsInCNF(l) &&
                                    !IsInConflict(i, l) &&
                                    !IsInConflict(j, l) &&
                                    !IsInConflict(k, l))
                                {
                                    Found_l = true;
                                }

                                if ((l % BlockSize) == BlockSize - 1)
                                {
                                    if (!Found_l)
                                    {
                                        goto Dont_Set_Found_k_true;
                                    }
                                }
                            }

                            Found_k = true;
                        }
Dont_Set_Found_k_true:;
                        if ((k % BlockSize) == BlockSize - 1)
                        {
                            if (!Found_k)
                            {
                                goto Dont_Set_Found_j_true;
                            }
                        }
                    }

                    Found_j = true;
                }
```

```
Dont_Set_Found_j_true:;
                if ((j % BlockSize) == BlockSize - 1) // last clause in block?
                {
                    if (!Found_j)
                    {
                        Active[i] = false;
                        ChangesExisting = true;
                        goto Next_i;
                    }
                }
            }
Next_i:;
        }
    }
}

bool SAT = false; // SAT contains the solving result: true for CNF is solvable,
                  // false for CNF is UNSAT

for (int i = PossibleClauseNumber - BlockSize; i < PossibleClauseNumber; i ++)
{
    if (Active[i])
    {
        SAT = true;
        break;
    }
}
```

Why does this code detect the solvability of any 3-SAT CNF? This question is to be answered in the following chapters step-by-step. As already explained, the polynomial solver must determine for each clause c if that clause would appear in an active solution. This information is stored in the array Active[].

Example:

Clause path with active solutions at end of each block:

```
00--            <- CNF clause
01--            <- CNF clause
10--
11-- 10--,11--
0-0-
0-1-
1-0-            <- CNF clause
1-1- 101-,111-
0--0
0--1
1--0
1--1 1010,1110  <- CNF clause
-00-
-01-
-10-
-11- 1010,1110
-0-0
-0-1            <- CNF clause
-1-0            <- CNF clause
-1-1 1010       <- CNF clause
--00
--01
--10
--11 1010
```

We perform the clause path walking as introduced in 3.1. When we have arrived at the end of the last block, we know that the exponential solver would have the active solution 1010 left. This has already been explained. Then we know that the following clauses c do still appear in an active solution, because they are not in conflict with that active solution:

```
10--,
1-1-,
1--0,
```

```
-01-,
-0-0,
--10.
```

Therefore, the polynomial solver must contain the following active clauses, i.e. clauses with a related Active[] value of true:

```
Active[10--] = true;
Active[1-1-] = true;
Active[1--0] = true;
Active[-01-] = true;
Active[-0-0] = true;
Active[--10] = true;
```

All other Active[] values must be false:

```
Active[00--] = false;
Active[01--] = false;
Active[11--] = false;
Active[0-0-] = false;
Active[0-1-] = false;
Active[1-0-] = false;
Active[0--0] = false;
Active[0--1] = false;
Active[1--1] = false;
Active[-00-] = false;
Active[-10-] = false;
Active[-11-] = false;
Active[-0-1] = false;
Active[-1-0] = false;
Active[-1-1] = false;
Active[--00] = false;
Active[--01] = false;
Active[--11] = false;
```

So the polynomial solver shall work similar to the exponential solver, the difference is that the exponential solver saves active solutions as m-SAT clauses, with $3 \leq m \leq n$. The polynomial solver, in contrast, shall save this information in form of exact-3-SAT clauses, of which there are polynomially many. The only important question left is: How does the polynomial solver know which possible clauses would appear in an active solution? From the exponential solver's algorithm definition we can deduce: An active solution u is only kept active (i.e. is not removed) if in each block there's a CNF-absent clause which is not in conflict with u. When extending this demand to the whole clause path, it can be inferred: An active solution u is only kept active if there is in each block of the clause path at least one CNF-absent clause which is not in conflict with at least one CNF-absent clause from each other block.

Example:

- Given the basic assumptions of the previous example.
  Then the active solution u = `1010` is only kept active by the *exponential* solver because there are the CNF-absent clauses

  ```
  10-- in block 1,
  1-1- in block 2,
  1--0 in block 3,
  -01- in block 4,
  -0-0 in block 5 and
  --10 in block 6.
  ```

  These six clauses are all not in conflict with u = `1010`.

Therefore: To determine if any clause i would be in an active solution, the polynomial solver just needs to check if there is in each block of the clause path at least one clause j which is not in conflict with i, which does not appear in the CNF and whose Active[j] value is still true. This is the main action the polynomial solver performs, as one can recognize by regarding its source code carefully (3.2).

Example:

- Active[i = `10--`] is kept true by the *polynomial* solver because there are the CNF-absent clauses

  ```
  10-- in block 1,
  1-1- in block 2,
  1--0 in block 3,
  -01- in block 4,
  -0-0 in block 5 and
  --10 in block 6.
  ```

  These six clauses are all not in conflict with i = `10--`. This conforms the previously given example with the active solution u = `1010`.

By checking *all* blocks for non-conflicting, active clauses, and not only the ones before i, the polynomial solver does some kind of foreseeing if the exponential solver would keep an active solution active at all. So the polynomial solver can decide for an i in some block b_1 already if the exponential solver would remove all active solutions containing i in a future block b_2 which lies after (i.e. in the clause path visualization below) b_1. For example, the polynomial solver would not keep Active[i = `10--`] = true, with i from the first block, if there wasn't any Active[`1--0`] = true or Active[`1--1`] = true in the third block. This foreseeing does quicker set Active[i] to false, what reduces the count of Active[] = true, which in turn makes the solving process more straightforward (and easier to understand for the examiner).
By now, not all details of how and why the polynomial solver works might be clear. Succeeding chapters and especially the proof of correctness will give a deeper insight.

3.3. 0/1 Inconsistency

3.3.1. Problem description

Different from the source code, the just given explanation mentioned only two clauses i and j. It was suggested that the polynomial solver sees i as in an active solution if there is in each block one clause j which is not in conflict with i, which does not appear in the CNF and whose Active[j] value is true. This implies that the polynomial solver would work with 2 loops, i and j. But this was a preliminary simplification only, to deduce the polynomial algorithm step-by-step.
I did very comprehensive tests and noticed that the polynomial solver did not work properly with less than summarized 4 loops, when 3-SAT CNFs were solved. The polynomial solver did with less than 4 loops either return wrong results in what concerns the final satisfiable/UNSAT decision, or it did not correctly set all Active[] values. When using i, j, l loops only (without the k loop), it happened, especially for larger CNFs with $n \geq 12$, that too many Active[] values remained true. This has been found out with the help of an exponential auxiliary procedure which determines the Active[] values fail-safe (this procedure is named Debug_VerifyActive(), more on this in the upcoming topic 7.2). When using i, j, k, l loops, i.e. 4 loops, absolutely all Active[] values had the right value at end of the polynomial solving process, for all tests done (over 1,000,000), and even for large CNFs (around some thousand CNFs with $12 \leq n \leq 15$ solved). When 2-CNFs were solved, the polynomial solver did always work also with i, j, l loops, i.e. 3 loops. The fault the polynomial solver does when using too few loops is the following: The polynomial solver accepts, to keep Active[i] = true, in one block only j's which are in conflict with j's from other blocks. This means some j's have a 0 char at some location where other j's have a 1 char. This problem is to be called '0/1 inconsistency'.

Example:

```
Exponential solver  Polynomial solver
     ↓              ↓
00--                               <- CNF clause
01--                               <- CNF clause
10--                10-- = j2
11-- 10--,11--
0-0-
0-1-
1-0-                               <- CNF clause
1-1- 101-,111-
0--0
0--1
1--0
1--1 1010,1110                     <- CNF clause
-00-
-01-
-10-
-11- 1010,1110      -11- = j1
-0-0
-0-1
-1-0
-1-1 1010,1110
--00
--01
--10
--11 1010,1110      --10 = i
```

In this example, the polynomial solver would, if only seeking for *one* suitable j in one block after another, accept two j's (j_1 and j_2) in two blocks (#4 and #1) to keep Active[i = `--10`] = true. The problem is that j_1 = `-11-` appears in the active solution `1110` and j_2 = `10--` appears in an *other* active solution `1010`. Fatal is here that j_1 has a 1 char at second position where j_2 has a 0 char.

3.3.2. Preliminary supposition: no 0/1 inconsistency with enough loops

The 0/1 inconsistency just shown in the prior example:

```
10-- = j
-11- = l
--10 = i
```

can be detected reliably when using 3 loops. As soon as the polynomial solver consists of 3 loops and checks *at the same time* if the related three clauses are in conflict, it does not accept the sample j = `10--` and l = `-11-` any more to keep Active[i = `--10`] = true, because j conflicts with l. This is the behavior we want to have, because conflicting clauses shall not be the reason for keeping Active[] values true. The advantages and proficiencies of a sufficient count of loops are thematized in the following chapters.

3.3.3. Why 0/1 inconsistency does not occur

The following passages explain why the polynomial solver does not fall for 0/1 inconsistency, and which mechanisms of the polynomial solver avoid this pitfall.

No 0/1 inconsistency at positions of i's 0/1 chars

Trivial case: All j,k,l clauses accepted to keep Active[i] = true have the same property: the 0/1 chars in j,k,l at positions where i has a 0 or 1 char cannot be in conflict. This is explicitly ensured by the polynomial solver, done by the source code lines "`if (!IsInConflict(i, j) ...`". This means that all those Active[i] are kept true for which it applies: for each of the three 0/1 chars of i, there must be in *each* block at least one CNF-absent, active and non-conflicting clause which has that same 0/1 char. This visualized for one concrete example i:

```
??a??a??a?? block 1
??b??b??b?? block 2
??c??c??c?? block 3
??d??d??d?? block 4
...
--0--0--0-- i
...
??e??e??e?? block (PossibleClauseNumber / 8) - 1
??f??f??f?? block (PossibleClauseNumber / 8)
```

To keep Active[i] = true, each a,b,c,d,e,f must be either '0' (because i has a 0 char there) or '-'.
The chars (0/1/-) indicated by the placeholders '?' are not regarded in this consideration.

No 0/1 inconsistency within each set i,j,k,l

The only question left is if there can be conflicts at positions apart from the positions of i's 0/1 chars.
Here the following two cases must be distinguished:

1) Within the clauses i,j,k,l regarded all together at a time by the polynomial solver, there cannot be a conflict at all. The definition of the polynomial solver does not allow those j,k,l to keep Active[i] = true which are in conflict pair-wise.

Examples:

- The following set j,k,l is not accepted by the polynomial solver, the underlinings mark the conflicts:

```
0-0------0- = j   <- not accepted to keep Active[i] = true
-----0---10 = k   <- not accepted to keep Active[i] = true
-1------0-1 = l   <- not accepted to keep Active[i] = true

--0--0--0-- = i
```

- The following set j,k,l indeed is accepted:

```
0-0------0- = j   <- accepted to keep Active[i] = true
-----0---01 = k   <- accepted to keep Active[i] = true
-1------0-1 = l   <- accepted to keep Active[i] = true

--0--0--0-- = i
```

No 0/1 inconsistency at all

2) But can it be that the polynomial solver does accept two sets j_1,k_1,l_1 and j_2,k_2,l_2, whereby the clauses within the same set are not in conflict, but they are among the two sets?

Example:

-
```
0-0------0- = j1   <- the clauses of this set...
-----0---01 = k1   <- the clauses of this set...
-1------0-1 = l1   <- the clauses of this set...

--00-----1- = j2   <- ...are partially in conflict with clauses of this set
-----0---10 = k2   <- ...are partially in conflict with clauses of this set
----1---0-0 = l2   <- ...are partially in conflict with clauses of this set

--0--0--0-- = i
```

This is a very important question to resolve. While the previously shown problems are solved by the polynomial solver in trivial ways, it is not obvious how conflicts among accepted sets of j,k,l are avoided. Fortunately, there's a solution and that solution is even not hard to understand: The key is to recognize that the polynomial solver applies the actions to detect the 'trivial case' *recursively*. 'Recursively' because each j, k and l will once be treated like i as well. The fact that the i loop iterates through exactly the same possible clauses as j,k,l do implements this (evident from the polynomial solver's source code). This means the polynomial solver will set Active[i] := false for *every* $i \in PC$ which does not have *non-conflicting* j,k,l as described in 'trivial case'. The seek for those i is done until no Active[] changes can be done any more. The polynomial solver's 'while (ChangesExisting)' loop ensures this. Therefore, the polynomial solver will reach a state where there are only clauses active which have 'the same 0/1 chars in each clause path column'.

Definition 3.3.1. *The state 'the same 0/1 chars in each clause path column' shall be defined as follows:*

For each possible choice of two block b_1 and b_2, it applies:
b_1 contains at least one active clause c_1 and b_2 contains at least one active clause c_2 for which it applies:

Definition 1) Each char of c_1 is either '-', or equal to the char of c_2 at the same position, or the char of c_2 at the same position is '-'.
Definition 2) More compact, this just means c_1 is not in conflict with c_2.

In this state, the foundation for 0/1 inconsistency, which is that there are *exclusively* conflicting clauses among two blocks, is not given. Therefore 0/1 inconsistency cannot occur any more when the polynomial solving process has progressed.

Example:

- The following clause path visualization shows an example situation where there are 'the same 0/1 chars in each clause path column'. There are only clauses shown with an Active[] value of true, because only such clauses are important, in the sense that they have influence on the further process flow of the polynomial solver:

```
00--    block 1
01--    block 1
0-0-    block 2
0--0    block 3
0--1    block 3
-00-    block 4
-10-    block 4
-0-0    block 5
-0-1    block 5
-1-0    block 5
-1-1    block 5
--00    block 6
--01    block 6
```

  Counter example:
  In block 6,

```
--10
```

  cannot stay active, because there's no active clause with a 1 char at third position in block 4 and also not in block 2.
  In the shown example, all clauses are active which appear in at least one of the active solutions `0000`, `0001`, `0100`, `0101`.

Please notice that it is *not* requested that *all* active clauses are not in conflict pair-wise, but it is merely requested that there are non-conflicting clauses among any choice of two *blocks*. Therefore it is possible that the clauses from more than one active solution are active. I currently suppose also in this case the state 'the same 0/1 chars in each clause path column' is correctly established. At least the polynomial solver never ran into any problem during testing.
I verified that there is always the state 'the same 0/1 chars in each clause path column' at the end of the polynomial solving process. This verification was carried out with a variant of the polynomial solver implementation, which is also located in the Zip file (see upcoming chapter 'Solver Implementation', 7). This variant checks if there are really 'the same 0/1 chars in each clause path column' after the polynomial solving process has finished. There was never any counter-example found in all tests I did (around 100,000 CNFs solved and checked).

3.3.4. Why are multiple loops required?

Why must the polynomial solver consist of three loops i,j,l (for solving 2-SAT CNFs), respectively four loops i,j,k,l (for solving 3-SAT CNFs)? The following considerations should give an in-depth look.
Below, i chars are placeholders for the 0/1 chars of the clause i, on whose Active[i] value is to be decided. x, y and z are placeholders for further 0/1 chars. All clauses are given in clause line notation. The presented clauses are only examples. x,y,z,i can be placed elsewhere as long as those locations follow the pattern of the example.

2-SAT

Assumed
j_1 `= x--i` is in an active solution and
j_2 `= y-i-` is in an active solution, but we don't know if $x = y$, then we can *not for sure* conclude that
i `= --ii` would be in one and the same active solution, because j_1 could be in conflict with j_2.

For this reason, the polynomial solver needs for solving 2-SAT CNFs to regard i, j, l at the same time. Because then it is ensured that Active[i] stays only true if $x = y$, as i, j, l must be not in conflict pair-wise to keep Active[i] = true. This is implemented explicitly by the source code of the polynomial solver.

Example:

- Although there's the state 'the same 0/1 chars in each clause path column' established, there could appear the following situation:

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

  The listed clauses are all assumed to be CNF-absent and active. Then the clause `-00` would be kept unjustifiably active with i, j loops only, because there's j_1 = `10-` and j_2 = `0-0`. 'The same 0/1 chars in each clause path column' doesn't avoid this, because in the first clause path column, there are in all blocks the same 0/1 chars (in the first and second block 0 and 1, the third block does not contain any 0/1 chars in its first column and thus isn't regarded). As soon as the polynomial solver checks *three* clauses at once for being in conflict, it does instantly detect that there are no CNF-absent and active j and l which are not in conflict pair-wise *and* both not in conflict with i. I found this situation when testing a solver implementation with i, j loops only. So it is for sure this situation *does* appear, if implementing too few loops.

3-SAT

The scheme can be extended to 3-SAT:
Assumed
j_1 = `xx---i` is in an active solution and
j_2 = `yy--i-` is in an active solution and
l = `zz-i--` is in an active solution, but we don't know if $x = y = z$, then we can *not* conclude that
i = `---iii` would be in one and the same active solution, because j_1, j_2 and/or l could be in conflict.

For this reason, the polynomial solver needs for solving 3-SAT CNFs to regard i, j, k, l at the same time. Because then it is ensured that Active[i] stays only true if $x = y$ and $y = z$ and $x = z$, as i, j, k, l must be not in conflict pair-wise to keep Active[i] = true. This is implemented explicitly by the source code of the polynomial solver.

Why there is more than one mechanism against 0/1 inconsistency

Why does the polynomial solver need both, enough loops *and* the state 'the same 0/1 chars in each clause path column' to function correctly? Why doesn't the latter state alone suffice? I suppose the reason is the following: 'The same 0/1 chars in each clause path column' verifies that active clauses are not in conflict pair-wise. This is important for ensuring that there are non-conflicting clauses among all clause path blocks, as explained in 3.3.3. Multiple loops $i, j, (k,) l$ verify that their clauses are not in conflict pair-wise *and beyond that* are not in conflict with *both* (2-SAT) resp. with the *three* (3-SAT) 0/1 chars of i. This is an additional demand which is not ensured natively by the state 'the same 0/1 chars in each clause path column', as pointed out in the previous example. I suppose four loops i, j, k, l are finally enough, because then all of i's up to three 0/1 chars can be checked for being in conflict with j, k, l. The two shown examples (one for 2-SAT and one for 3-SAT, 'Assumed ...'), where j(, k) and l each have a distinct 0/1 char in common with i, are probably the worst cases which require the maximum loop count (three loops for 2-SAT, four loops for 3-SAT).

4. Proof of correctness

4.1. Why SAT detection is reliable

In case the CNF is solvable, then there's at least one solution ('model') which satisfies the CNF. Then all possible clauses do not appear in the CNF which are not in conflict with the inverse solution. This has been shown in 3.1.1. So in each block of the clause path there's at least one possible clause p which does not appear in the CNF (else not one active solution could rest and the CNF would always be detected as 'UNSAT' by the fail-safe exponential solver, what is a contradiction). All those p's are not in conflict pair-wise, because they are all not in conflict with the inverse solution. Finally, the Active[] value of each p is initially true, because the array Active[] has been initialized entirely to true. This means in each block there's at least one clause p which serves as j, k or l and keeps Active[i] = true, with i also equal to some p. Because then also in the last block there remains at least one Active[i] = true, the polynomial solver determines the CNF as satisfiable, as desired.

Example:

- Solution ('model'):

  0101

  Inverse solution:

  1010

  Clause path:

  00--
  01--
  10-- <- p_1, not in CNF
  11--
  0-0-
  0-1-
  1-0-
  1-1- <- p_2, not in CNF
  0--0
  0--1
  1--0 <- p_3, not in CNF
  1--1
  -00-
  -01- <- p_4, not in CNF
  -10-
  -11-
  -0-0 <- p_5, not in CNF
  -0-1
  -1-0
  -1-1
  --00
  --01
  --10 <- p_6, not in CNF
  --11

4.2. Why UNSAT detection is reliable

Foreword

It follows a proof for why the polynomial solver detects any unsatisfiable ('UNSAT') CNF as such. The presented version is the best approach I found so far. If some reader has an addition, or an alternative proof, I'd welcome him/her to publish the own variant as well.
The proof begins with an example for 2-SAT, n=4. It will be shown in which cases the polynomial solver sets Active[--00], Active[--01], Active[--10] or Active[--11] to false, and that these cases are given if the CNF to solve is UNSAT. The presented proof can be analogously applied for 3-SAT, larger n and other clauses than --00 etc. as well.

2-SAT, n=4

Claim 4.2.1.

Active[--cd] is only kept true if
Active[-b-d] is true and -b-d is CNF-absent,
Active[-b-d] is only kept true if
Active[-bc-] is true and -bc- is CNF-absent,
Active[-bc-] is only kept true if
Active[a--d] is true and a--d is CNF-absent,
Active[a--d] is only kept true if
Active[a-c-] is true and a-c- is CNF-absent,
Active[a-c-] is only kept true if
Active[ab--] is true and ab-- is CNF-absent.

Proof: The following rules apply to the construct shown in the claim:

c,d) c and d are the both 0/1 chars of a clause from the last block. There are, for 2-SAT, the following possible combinations:

```
c d clause in last block
0 0 --00
0 1 --01
1 0 --10
1 1 --11
```

I'm using the placeholders c and d here to generalize the proof. Like this, the proof shows in which cases *any* of the 4 (for 2-SAT) clauses in the last block stays active. This is of importance because the polynomial solver returns 'UNSAT' only if *every* clause from the *last* block has an associated Active[] value of false.

a,b) a is 0 or 1.
Here, it is of *great* importance: a must always be the same (0 or 1) in *all* clauses accepted by the polynomial solver to keep Active[] values true. The same applies to b: b is 0 or 1. b can differ from a (e.g. $b = 1$ and $a = 0$ or vice versa). But: Again, it is of *great* importance: b must always be the same (0 or 1) in *all* clauses accepted by the polynomial solver to keep Active[] values true. These requirements *are* ensured because the polynomial solver will always enter a state where there are 'the same 0/1 chars in each clause path column'. Therefore there will finally be either all clauses with $a = 0$ active, or none. Or there will finally be either all clauses with $a = 1$ active, or none. The same holds true for b. The polynomial solver supports also the case in which there are multiple clauses per block active, some with $a = 0$ and some with $a = 1$. But then there must be such clauses in *all* blocks. The example related to definition 3.3.1 shows such a set of active clauses whereby there is more than one clause per block active.

Dependency) The stated dependencies, like:
Active[`--cd`] is only kept true if
Active[`-b-d`] is true and `-b-d` is CNF-absent
are caused by the following rules implemented by the polynomial solver: Generally, to keep any Active[i] true, there must be CNF-absent, active clauses $j, (k,) l$ found which are not in conflict with i. In *each* block, there must be at least one such additional clause found. This means, inter alia in the block *prior* to the one containing i, there must be a CNF-absent, active and non-conflicting clause. Therefore we know that the polynomial solver needs to find at least one Active[`-b-d`] = true with `-b-d` from block 5 to keep Active[i = `--cd`] = true, with `--cd` from block 6.
The reader might have rightly noticed that only a subset of the dependencies demanded by the implementation of the polynomial solver is regarded. Nevertheless, these dependencies examined in this proof are sufficient to prove the reliable determination of the CNF's unsatisfiability.

Example of the dependencies with concrete clauses

- Active[`--00`] is only kept true if
  Active[`-0-0`] is true and `-0-0` is CNF-absent,
  Active[`-0-0`] is only kept true if
  Active[`-00-`] is true and `-00-` is CNF-absent,
  Active[`-00-`] is only kept true if
  Active[`1--0`] is true and `1--0` is CNF-absent,
  Active[`1--0`] is only kept true if
  Active[`1-0-`] is true and `1-0-` is CNF-absent,
  Active[`1-0-`] is only kept true if
  Active[`10--`] is true and `10--` is CNF-absent.

  In this example, $a = 1, b = 0, c = 0, d = 0$.

Negative example

- Because the polynomial solver will reach the state 'the same 0/1 chars in each clause path column', the following situation is not possible (the reader can assume that the shown list of active clauses is entire, i.e. there are no more active clauses):

  Active[`--00`] is only kept true if
  Active[`-0-0`] is true and `-0-0` is CNF-absent,
  Active[`-0-0`] is only kept true if
  Active[`-00-`] is true and `-00-` is CNF-absent,
  Active[`-00-`] is only kept true if
  Active[`1--0`] is true and `1--0` is CNF-absent,
  Active[`1--0`] is only kept true if
  Active[`1-0-`] is true and `1-0-` is CNF-absent,
  Active[`1-0-`] is only kept true if
  Active[`11--`] is true and `11--` is CNF-absent.

  For instance, here the assignment Active[`11--`] = true cannot appear near the end of the polynomial solving process because there's only Active[`-00-`] = true in an other block, and these two clauses `11--` and `-00-` are in conflict. Therefore, Active[i = `11--`] would not be kept true by the polynomial solver because the solver doesn't find any active, non-conflicting j or l in the block containing

`-00-`. The same applies to the two clauses `11--` and `-0-0`. Both cannot stay active, as they are in conflict and the only clauses in their respective block. As a result, all Active[] values will be set to false by the polynomial solver in some kind of domino-effect, until no active clauses rest any more. □

From all of the presented findings, we can conclude that the polynomial solver keeps Active[`--cd`] only true if

all clauses from 2oo(`00cd`) are active and CNF-absent, or if
all clauses from 2oo(`01cd`) are active and CNF-absent, or if
all clauses from 2oo(`10cd`) are active and CNF-absent, or if
all clauses from 2oo(`11cd`) are active and CNF-absent.

This means that Active[`--cd`] is *only* kept true if there are *all* clauses from at least one (inverse) solution CNF-absent. The basic assumption in this UNSAT proof is that this is *not* the case, for each solution (and therewith also for each inverse solution) there's at least one clause in the CNF which is not in conflict with that solution. Therefore, Active[`--cd`] will not stay true, for all four 0/1 combinations of c,d. As a result, no clause will rest in the last block which has an associated Active[] value of true. And this means in return that the polynomial solver will return 'UNSAT', as desired.

3-SAT

I assume that 3-SAT does not cause any additional problems, there are just 3 letters (out of a,b,c,d etc.) in each clause. In contrast to the logical resolution, there's no force to make sure clauses have enough 0/1 chars to take up information (like the resolvent of the logical resolution, which might have more than three 0/1 chars, especially when solving pigeon hole problems). The polynomial solver does only check if clauses are in conflict, if they are marked active, and if they appear in the CNF. These checks work independently from the count of 0/1 chars.

Larger n

The shown scheme can easily be extended to any n. Then clause dependencies can be requested as in the 2-SAT, n=4 case. Each Active[i], with i from block b_i, is only kept true if there is at least one CNF-absent, active and non-conflicting clause in the block prior to b_i.

Example:

- Active[`------000`] is only kept true if
  Active[`-----f-00`] is true and `-----f-00` is CNF-absent,
  Active[`-----f-00`] is only kept true if
  Active[`-----f0-0`] is true and `-----f0-0` is CNF-absent,
  Active[`-----f0-0`] is only kept true if
  Active[`-----f00-`] is true and `-----f00-` is CNF-absent,
  Active[`-----f00-`] is only kept true if
  Active[`----e--00`] is true and `----e--00` is CNF-absent,
  and so on.

  The only thing that *must* be ensured here is that the char represented by f, e, and the remaining placeholder chars not shown here, are the same (0 or 1) in *all* those clauses. This is ensured by the state 'the same 0/1 chars in each clause path column', which the polynomial solver will always reach.

5. Remarks

Some questions and answers about loops appearing in the polynomial solver's sample implementation:

- Do we need 'full range' loops (i,j,k,l iterate all from 0 to PossibleClauseNumber - 1), can't we e.g. just check $j,k,l < i$? This means can't we just check j,k,l which appear above i in the clause path? I implemented and tested this. Restricted loop ranges did not work, the polynomial solver partially returned wrong results. Probably because with restricted loop ranges not the clauses of all blocks are checked for not being in conflict with i, what could understandably lead to errors, because it's an ignoring of potentially useful information.
- Why do we need the 'while (ChangesExisting)' loop? This loop enables the polynomial solver to handle cases where some Active[] value is set to false and this Active[] value change would have influenced an Active[] value setting done *before*. So the 'while (ChangesExisting)' loop is used to support any *order* in which Active[] values are set to false. In tests, the polynomial solver mostly did only two 'while (ChangesExisting)' loop iterations, the initial one and another one in which no Active[] value changes appeared any more. This means here the 'while (ChangesExisting)' loop would not have been required. But in some tests, the polynomial solver did more than 2, namely 3 or 4, 'while (ChangesExisting)' loop iterations. In further tests, I realized that the Active[] values of the clauses in the very last block of the clause path were always set correctly also without 'while (ChangesExisting)' loop. This worked in some 10,000 tests and also solving the pigeon hole problem PHP_4 was managed without 'while (ChangesExisting)' loop. So if the operational scenario of the polynomial solver does not require to set absolutely all Active[] values correctly, but only those in the last block, then the solving process can possibly be sped up by removing the 'while (ChangesExisting)' loop. But this variant was tested much less intensively and its correctness is not as confident as that of the main version.

6. Complexity analysis

It follows the determination of the worst-case complexity of the polynomial solver, for the 3-SAT case. The size of the set *PC* is of great importance for determining the complexity because the polynomial solver's main work consists substantially of looping through the set of possible clauses. There are $PCNum = |PC| = O(n^3)$ many possible clauses, because we can place the three indices of all possible clauses using three nested loops, each having an iteration range not larger than 1 to n. Furthermore there are $2^3 = 8$ possibilities for each clause to choose the three ε values out of $\{0,1\}$. But because this is a constant complexity, it will not be observed in the O notation. Regarding all possible combinations of x many possible clauses one time has a complexity of $O((n^3)^x)$. This is the case because we had to implement x many nested loops, each having an iteration range of 1 to $|PC|$. The polynomial solver consists of 4 nested loops, each iterates from 0 to (excluding) *PCNum*. Within the loops, additional noteworthy work is only 1) checking if a clause appears in the CNF, 2) checking if two clauses are in conflict, 3) reading/writing array or single variable values. The results of 1) and 2) can be pre-computed independently and with a complexity smaller than that of the 4 nested loops.
1) has a complexity of $O(PCNum \times CNFClauseCount \times n)$, because for each possible clause we need to loop through the CNF clauses and compare maximal n many 0/1/- chars (assumed the possible clauses exist in clause line notation). There cannot be more *distinct* clauses in the CNF than there are possible clauses, so that checking if a clause appears in the CNF does not have a greater complexity than $O(PCNum \times PCNum \times n)$. 2) has a complexity of $O(PCNum \times PCNum \times n)$, because for each pair of possible clauses we need to loop through their n many 0/1/- chars and check if there is a conflict (0 comes upon 1 or vice versa). 3) Reading/writing variable values from/to RAM has a constant complexity O(1). This means running the main loops i, j, k, l is the most comprehensive work, which has a complexity of $O((PCNum)^4) = O((n^3)^4) = O(n^{12})$.
The last solver part to consider is the 'while (ChangesExisting)' loop. This loop is iterated not more than *PCNum* times, because at the latest when all Active[] values have been set to false, then the while loop is left. There are *PCNum* many Active[] values (i.e. array elements), so that the main loops i, j, k, l are never re-started more than *PCNum* times. Checking if there's an Active[c] = true with c from the last clause path block has a constant complexity and thus needn't to be added in. So the total complexity is composed of running the 'while (ChangesExisting)' loop and the 4 main loops i, j, k, l within, so that we get summarized a time and space complexity not greater than $O((PCNum)^5) = O((n^3)^5) = \underline{O(n^{15})}$.

7. Solver implementation

7.1. Download location and scope

There are C++ sample implementations available of both the polynomial solver and the exponential solver. They are each offered in versions for Windows and Linux. The demo implementations can be downloaded packed as a Zip file from the author's homepage www.louis-coder.com. Within the Zip, please regard the directory tree "Algorithm E ...". The directories "Algorithm B ..." and "Algorithm D ..." contain material related to older algorithms I published earlier. The reader can later attend to those versions, if desired. But this is not necessary to understand the Algorithm E presented in this paper. I recommend to focus on Algorithm E, because it is the easiest and best-explained one.

7.2. Performed testing and its results

The demo implementation of the polynomial algorithm (see previous topic) correctly solved over 1,000,000 random CNFs. The solvability of each random CNF was first determined using a fail-safe, exponential, brute force solver (which just tries out all 2^n many models, not to be confused with the exponential solver introduced in 3.1). Subsequently, the solvability of each random CNF was determined by the polynomial solver. The results did in each of the over 1,000,000 tests match. There was not one divergence. The CNF dimensions varied mainly from $n = 4$ to $n = 15$, with n times 2 to n times 10 many clauses. The largest CNF checked was a pigeon hole problem CNF with $n = 26$.[2]
In case a CNF is solvable, it will be the case that Active[c] = true array elements will remain. The demo implementation verified (using another fail-safe, exponential auxiliary algorithm) that all Active[c] values the polynomial solver left true are really meant to be, i.e. that each c from an Active[c] = true is a clause not in conflict with the inverse of any solution ('model') which satisfies the CNF. Like this, billions of Active[c] values have been verified (there are up to a few thousand of them for each CNF instance). There was not one error. The check is done by the procedure Debug_VerifyActive(), the interested reader might want to examine the source code for further details.
Also pigeon hole problem CNFs of size 2, 3 and 4 have been correctly determined by the polynomial solver as UNSAT.[3] Especially the correct solving of the pigeon hole problem CNF of size 4 is interesting, because if the polynomial solver would be 'some kind of' resolution solver, it could not detect this pigeon hole problem as UNSAT. The theory says that any resolution solver would need to store and further process at least one 4-SAT resolvent clause for this pigeon hole problem.[7] So this limitation of (bounded-width) resolution solvers seems not to apply to the presented polynomial algorithm. To be absolutely sure that the polynomial solver outperforms the bounded-width resolution solver, I inputted the pigeon hole problem CNF of size 4 into a self-programmed, bounded-width resolution solver which does not use resolvents with more than 3 literals. That resolution solver failed (it returned 'satisfiable'), as expected. The polynomial solver whereas worked, it returned 'UNSAT'.

8. Conclusion and discussion

An algorithm was presented of which I assume that it determines the solvability of any 2- or 3-SAT CNF in polynomial time and space. The finding that my solver implementation solved all CNFs I inputted, and the fact that I found no fatal problems or contradictions when developing the proof of correctness, make me suppose the algorithm is correct. On the other hand, results of testing are no guarantee for absolute validity. And the proof of correctness could contain flaws, or could miss one or more important problems. Nevertheless, I do

[2] PHP_4, converted by me to pure 3-SAT. Search Zip file for 'Pigeon_n=4_3.txt'.
[3] A definition of the pigeon hole problem is given in e.g. [6, p. 52].

strongly believe the paper can contribute important ideas in the field of building SAT solvers. The established polynomial algorithm is very easy to understand, because it works according to a very small set of rules only. Therefore it will probably not need much time until it is completely understood and verified by the scientific community. In particular, I would appreciate if someone who has the capabilities could test the polynomial solver on a high-performance computer, e.g. by inputting very large pigeon hole and random CNFs. In case the polynomial solver should be finally proven to be correct, a breakthrough would be achieved in the related fields of mathematics and theoretical computer science.

9. Acknowledgments

I thank Dr. Mihai Prunescu, Simion Stoilow Institute of Mathematics of the Romanian Academy, for helpful tips and a reference to an earlier version of the polynomial algorithm in one of his articles (see [4] and [5]).

References

[1] Michael R. Garey, 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] Bronstein, Semendjajew, Musiol, Mühlig, Taschenbuch der Mathematik, *Verlag Harri Deutsch*, Thun und Frankfurt am Main, (2000), ISBN 3-8171-2015-X.
[4] Mihai Prunescu, About a Surprizing Computer Program of Matthias Müller, https://imar.academia.edu/MihaiPrunescu. Accessed 2020-July-21.
[5] Mihai Prunescu, About a Surprising Computer Program of Matthias Müller, Convexity and Discrete Geometry Including Graph Theory, *Springer International Publishing*, Mulhouse, France, (2014), ISBN 978-3-319-28186-5_9, http://dx.doi.org/10.1007/978-3-319-28186-5_9. Accessed 2020-July-21.
[6] Schöning, Torán, The Satisfiability Problem, *Lehmanns Media Berlin*, (2013), ISBN 978-3-86541-527-1.
[7] Anup Rao, More Pigeons, and a Proof Complexity Lower Bound, https://homes.cs.washington.edu/~anuprao/pubs/CSE599sExtremal/lecture5.pdf. Accessed 2020-July-21.