Polynomial SAT-Solver

Usage Instructions and further Information

by Matthias Mueller (a.k.a. Louis Coder)
louis@louis-coder.com
Instructions Version 1.0 - November 24, 2013

1 Introduction

1.1 What is this Document about?

This is the usage manual that gives information about how to use the 'Polynomial SAT-Solver' by Matthias Mueller. The 'Polynomial SAT-Solver' application is a Windows program that implements the polynomial SAT solving algorithm (which was also invented by Matthias Mueller). This polynomial algorithm itself is explained in an extra document - this here tells only how to use the solver demo program. You can download the solver, instructions and algorithm explanation from:

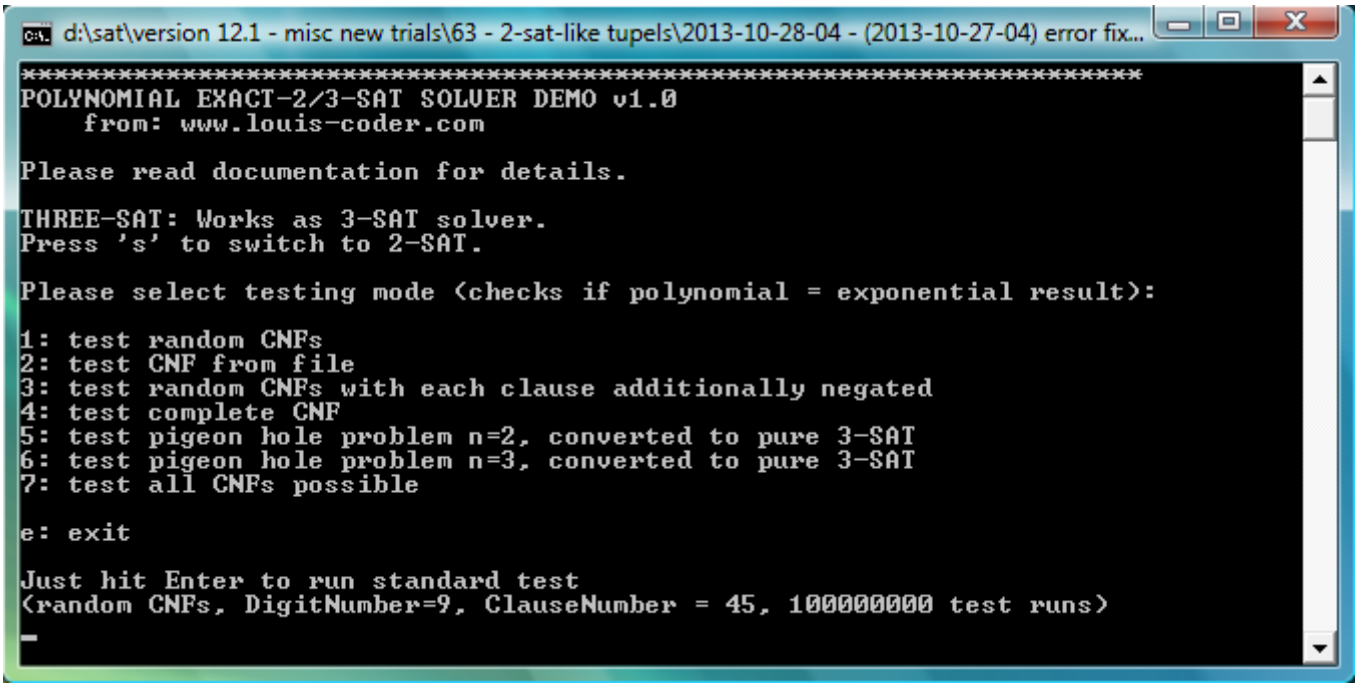
http://www.louis-coder.com/Polynomial_3-SAT_Solver/Polynomial_3-SAT_Solver.zip.

1.2 Requirements of the Reader

The Polynomial SAT-Solver Usage Manual, as well as the solver and its 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 The Solver Application

When you run the solver program (in the following also called 'solver application' or 'solver app'), a console window will open that looks similar to the following one (in the screenshot, you see the solver's main menu):



2.1 What does the Solver Application do?

The solver solves exact 2- or 3-SAT-formulas with both a time-exponential, fail-safe method (trying out all possible solutions, known as "brute-force") and the algorithm that is supposed to solve the Satisfiability Problem in polynomial time and space. 'To solve' means output if the SAT-formula is satisfiable or not. The polynomial algorithm does not find a concrete solution.

The solver is able to solve one CNF from a file or to do up to 100 million test runs consecutively, whereby in each test run the solver creates a random formula, solves it once by the exponential and once by the polynomial algorithm and checks the two results for equality - they must always be equal. If they are not, it might be that the polynomial algorithm failed, because it is very improbable that the exponential solving worked faulty. But I strongly assume that the polynomial algorithm will not return wrong results (at least it did not do this for ~1 million test runs).

2.2 Input the Solver shall process

As already mentioned, the solver processes SAT-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.

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 The '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:

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

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- respectively console 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.

2.4 Origin of the SAT-Formulas

The CNFs to be solved can come from:

a) A file in DIMACS format.

The DIMACS format description is viewable at:

http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps.

Practically, the 'CNF from file' feature is not meant to be used often. It is better to use the random formula generator (see b)), unless you really have a file containing a CNF you need to test. But note that the solver ignores meta data in DIMACS files, the solver just reads in clauses until the end of the file because this is easier to implement. I experienced people had problems as the meta data did not match the actual content of the file. Some other third-party solvers might process the meta data, my solver does not, so there could possibly be different results.

b) Created by the solver using a random generator.

Randomly created formulas can have different characteristics:

b.1) Completely randomly created.

- Implemented like this:
  - Randomly select two (2-SAT) or three (3-SAT) different literal indices.
  - These become the indices of the literals.
  - Randomly negate or not negate each of those indices.

b.2) Completely randomly created, each clause also negated.

- Done like b.1, plus each clause negated.
  - Three examples (clause + negated clause):

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

  - If the desired clause count is not even, there might be one non-negated clause in the created CNF.
  - Tests showed that negating each clause leads to formulas 'harder to solve', i.e. produced more wrong solver output in an early state of my solver development.

b.3) CNF containing all possible clauses (always unsatisfiable).

- The CNF consists of all thinkable clauses with two (2-SAT) or three (3-SAT) literals.
  - Example (2-SAT, DigitNumber = 3):

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

b.4) Test run with all thinkable formulas of a defined size.

- The solver creates CNF consisting of all possible on/off combinations of the possible clauses.
  - For 2-SAT, DigitNumber = 3, the possible clauses are:

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

So the first CNF will be:

```
00-
```

The second CNF will be:

```
01-
```

The third CNF will be:

```
00-
01-
```

The fourth CNF will be:

```
10-
```

and so on. Please view the code for implementation details.

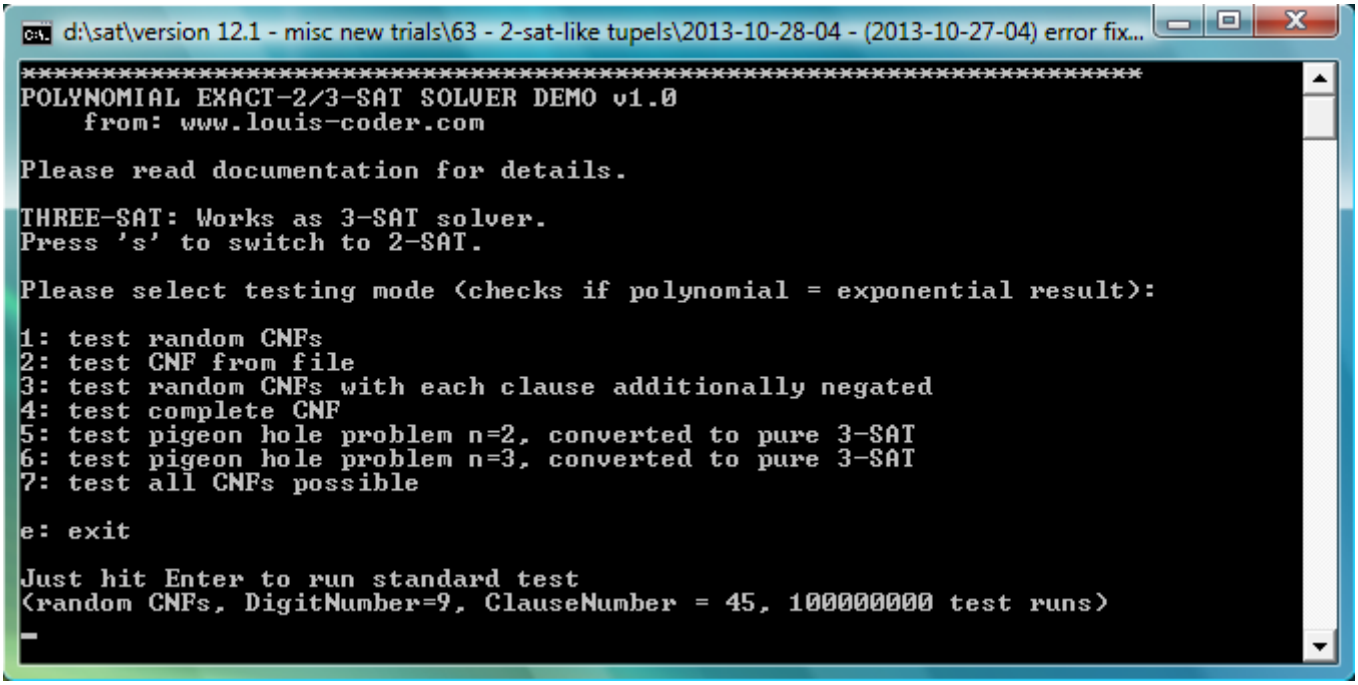
3 Using the Solver

3.1 Resizing the Console Window

For the future use, it might be good to know that it is possible to change the console's size by left-clicking on the window frame and move the window border while holding the mouse button. You can durably re-size the console window by clicking on the icon in the left top corner. In the menu that pops up, click on 'properties'. A dialog opens that you can use to customize the console window. The changes last until you change the solver's path (i.e. if you move or rename the solver executable). You can also increase the amount of buffered text, see 'properties'. But note that printing to the console and therewith the time required for solving slows down if you use a large console text buffer and/or you enlarge the console window. Generally, you can speed up printing/solving many formulas by minimizing the window, so that the OS doesn't need to re-draw the console text permanently.

Also note that you can press the 'Pause' key on your keyboard (on most keyboard layouts, at the right end of the top-most key row) to interrupt the solver app. Press any key once to continue. If you accidentally made the solver ask 'abort? (y/n)', enter "n" and press Return once.

3.2 Features accessible in the Main Menu



In the solver's main menu (see screenshot), you can enter:

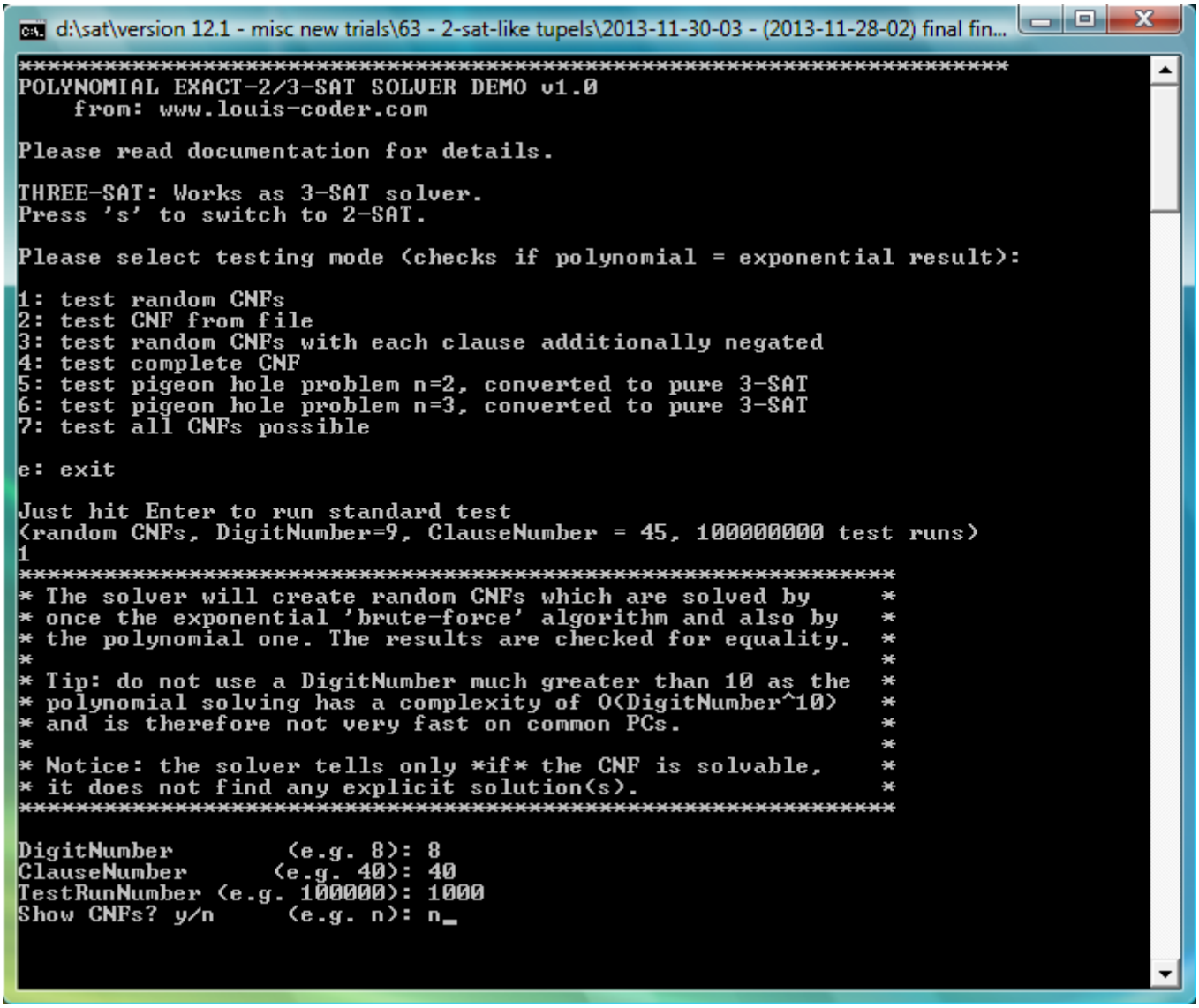
- "1" to "7": starts a test run.
  - "1", "3", "4" and "7": the solver 'invents' one or more random CNF(s), see topic 2.4.
  - "2": make the solver read a CNF from a file, also mentioned in topic 2.4.
  - "5" or "6": solve a 'Pigeon Hole Problem' CNF. The pigeon hole problem is a class of CNFs that need exponential time to be solved with resolution-based solvers. The polynomial solver should process also those formulas in polynomial time and space (what it does). Note that a Pigeon Hole Problem needn't to be exact 3-SAT, but here it has been converted to, by using some extra literals.
  - "s" switches between 2- and 3-SAT mode. You can make the solver solve 2-SAT formulas, which need less space to be displayed as ClauseTable and CNFs are easier to analyze for humans, or you use the 3-SAT mode, which is more important, as for 3-SAT there were only exponential algorithms known.

- "e" exits (alternatively you can click on the red window close button in the top right corner).

- Return runs the standard test. The standard test has been implemented to make the user able to get a quick impression of how the solver works. When running the solver app the first time you can just press Return.

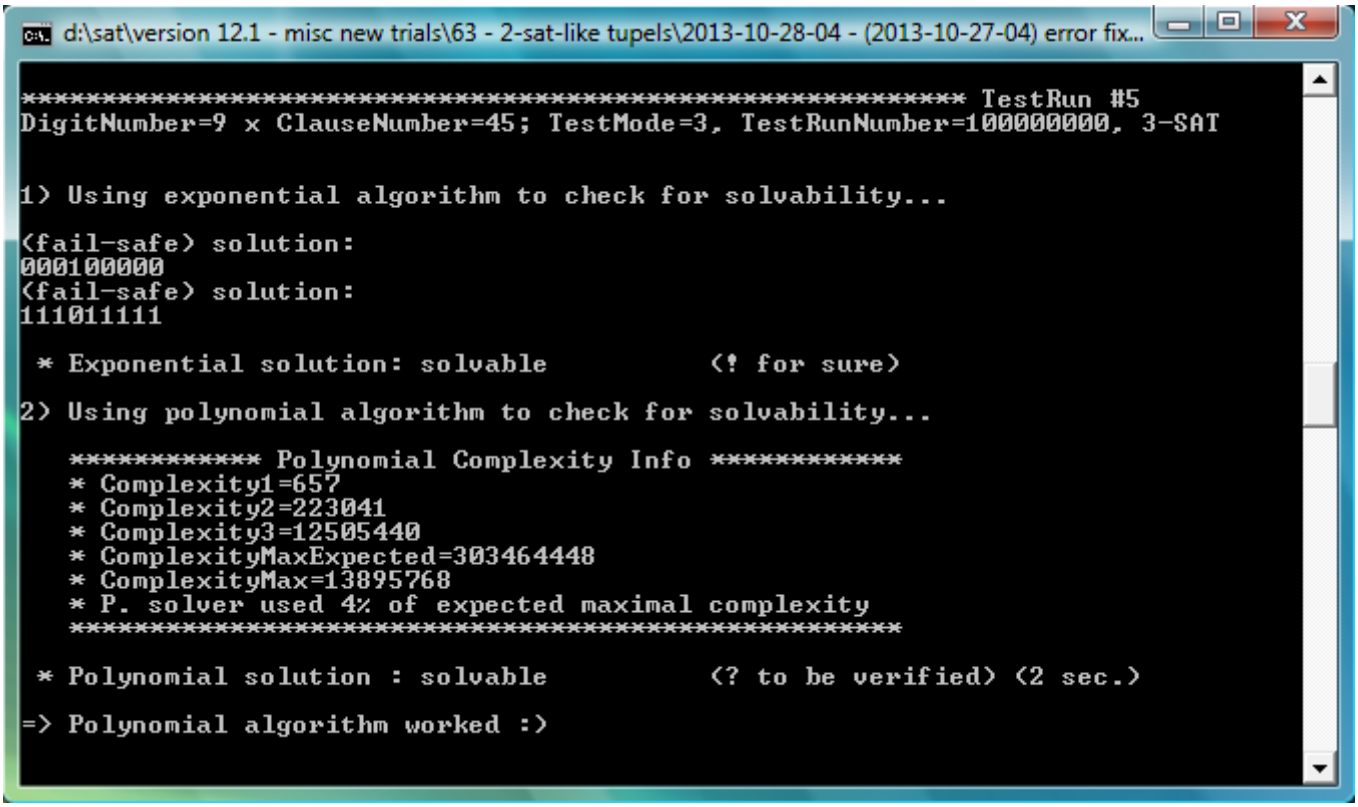
4 Instructions for a Sample Usage

1. Start the solver application.
2. In the main menu, enter "1" (always followed by Return) to begin a test run with randomly created formulas.
3. The solver asks for the DigitNumber. This is the highest possible literal index. Enter "8". This means that all literals have an index from minimal 1 to maximal 8.
4. The solver asks for the ClauseNumber. This is the exact number of clauses (AND-terms) that each of the randomly created CNFs has. Enter "40".
5. Now you are asked to enter the number of test runs. In each test run, a random CNF is created and tested with first the exponential, fail-safe algorithm and then with the polynomial one. The solver would stop and display an error message if the two results (exponential/polynomial) are not equal. But it is improbable that this happens (at least it did not happen for the latest ~1 million tests I did).
6. Finally the solver wants to know if you want to see each tested formula. If you agree ("y"), each CNF is printed to the console in ClauseLine notation (see topic 2.3). You might want to view the formulas during the first tests to get an idea of how they look like. Later void their displaying as the displaying slows down the solving.



To abort a running testing, press "y" and then Return. Note that aborting could take some time to react (aborting does not work during actual solving but only before and after).

5 Output while testing



- The beginning of each test run is marked by the asterisks line containing the current TestRun number (5 in the example).
- Then comes information what CNF was tested: DigitNumber (maximal possible literal index), number of clauses, what kind of formula ('TestMode', see topic 2.4), the total count of TestRuns to do and the number of used literals per clause (2-SAT/3-SAT). You can enter all this information when beginning a test run cycle (see topic 4).
- If you specified at the beginning of the test that you want to see the created CNFs, then you'd see it right now (not visible in the screenshot).
- First the solver checks if the CNF has a solution using the exponential algorithm, which is practically fail-safe but too slow for larger CNFs (with DigitNumber > ~25). The result is printed to the console ("solvable" for 'there is any solution', or "UNSAT" for 'there is no solution').
- The same is done with the polynomial algorithm.
- The result of both algorithms is compared (you should always see "=> Polynomial algorithm worked :)"), if there's a difference the solver would instantly stop and display an error message.
- Additionally, in the asterisks box, there's information about the complexity, which is practically the number of iterations in the inner-most loops the polynomial algorithm uses. The listed percentage is (ComplexityMax * 100 / ComplexityMaxExpected).

6 Further Information

6.1 ClauseTable Test

For the following formulas:

2-SAT, DigitNumber = {4, 5 or 6}

...the polynomial solver does an additional hard-coded ClauseTable test. Please read the document about the actual polynomial algorithm for information about the 'ClauseTable'. So there might actually be up to three different solver algorithms working:

1. the exponential solver (function "CNF_Solve_Exponential()" in "Polynomial_3SAT_Solver.cpp"),
2. the polynomial solver (function "CNF_Solve_Polynomial()" in "ActualSolver.h"),
3. possibly additionally the ClauseTable test (function "ClauseTableTest()" in "ActualSolver.h").

6.2 Ingenious CNF Sizes

- When you run the solver in 2-SAT mode, set the ClauseNumber to twice the DigitNumber.
- You can use DigitNumbers up to 25, the upper DigitNumber limit of the current solver implementation.
- Generally 2-SAT formulas are solved more quickly than 3-SAT ones.
- In 3-SAT mode, set the ClauseNumber to around 4.25 times the DigitNumber.
- Note that solving 3-SAT CNFs with DigitNumber > 10 might be slow, especially for CNFs that turn out as solvable (the solver might seem to 'hang' for some time).
- The ratio DigitNumber/ClauseNumber determines the tendency if more random CNFs will be satisfiable or unsatisfiable. For the DigitNumber and ClauseNumbers mentioned above, the ratio is around 1, as I tested out (equally many solvable/unsolvable ones). If you use a much greater ClauseNumber than DigitNumber, you get more unsolvable ones.

Generally, you can make the solver (invent and) process exact 2-SAT or exact 3-SAT formulas. It is important that the solver can solve exact 3-SAT formulas, because therefore no polynomial algorithms had been existing. Furthermore the solver has also an exact 2-SAT mode as 2-SAT formulas create a smaller ClauseTable and are easier to understand for humans. To put it in a nutshell, exact 3-SAT must work, exact 2-SAT is mainly used to visualize the algorithm and to make it better understandable.

7 Complexity of the Solver and practical Meaning

A detailed derivation of the exact complexity is shown in the algorithm explanation.

You can easily see that the implementation of the polynomial solver has a polynomial time-complexity because it consists of nested loops only. There is no pow() function and also no recursive procedure calling.

But unfortunately the solving speed of the polynomial solver is nevertheless not that high as the polynomial that describes the (worst-case) runtime has a high degree: O(DigitNumber^10), which is slow on normal PCs. But high-performance computers might solve even large formula instances, while even those HP-PCs could not solve large instances using the exponential algorithm (I assume an exponential solver needs O(2^DigitNumber) time).

This fact could be described descriptively like this:

- The polynomial solver needs 'medium' (solving-) time for small CNFs and 'much' time for large CNFs.
- An exponential solver needs 'few' time for small CNFs and 'incredibly much' time for large CNFs.

With 'small CNFs' I mean e.g. 10 different literals and 50 clauses and 'large CNFs' could have e.g. 100 literals and 500 clauses. Look at the number of work (e.g. internal loop-runs) the polynomial O(n^10) solver would have to do if n is the number of literals:

```
10^10  = 10000000000
100^10 = 10000000000000000000
```

The exponential algorithm would be faster for 10 literals, but it is over twelve billion times slower for 100 literals:

```
2^10  = 1024
2^100 = 1267650600228229401496703205376   (~12676506002 times 100^10)
```

This required-CPU-time difference gets larger and larger the more literals the CNFs to solve have.

So the polynomial algorithm should really be worthy for institutions that have a lot of calculating power, but as my sample solver is meant to be executed on normal PCs, it limits the DigitNumber (maximal literal index) to 25 and the ClauseNumber to 18500. Above these limits you would wait too long for an output.

8 What has been tested until now

As mentioned before, I had run over 1 million tests. In each of those tests the polynomial algorithm returned the same as the exponential, fail-safe one.

Most formulas I tested were of the type
- 2-SAT, DigitNumber = 3 to 25, ClauseNumber ~= DigitNumber * 2,
- 3-SAT, DigitNumber = 3 to 15, ClauseNumber ~= DigitNumber * 5.

I also used the solver's 'Test all CNFs possible' feature and tested
- all thinkable 2-SAT formulas with DigitNumber=4.

Finally,
- the solver successfully processed the 'Pigeon Hole Problems' (converted to exact 3-SAT) with n = 2, n = 3, which you can both re-test using the solver app.

Besides the mentioned ones, ten thousands formulas of various sizes have been tested.

9 Solver Source Code

I implemented the polynomial SAT solver as a computer program. You can download the C++ source code and the compiled Windows (native) binary. The binary should run on Windows XP, Vista, 7 and 8. It has been successfully tested on Windows Vista and 7. At best, you open and edit (and compile) the source code with Microsoft Visual Studio 2005 or later.

If you do not have a Windows PC or/and Visual Studio, you can view the source code using any text editor. As the solver is not a .NET program, it can probably be compiled for Linux with some little code changes.

Please note that within the source code, there are a lot of annotations that give additional information about the implementation. I encourage you to have a look at the code, especially if you are a computer scientist.

10 If the Solver should fail

If the demo solver program should report an error, please attend the following points:

- Make sure you ran the original, unchanged solver and you used it according to the instructions.
- If the polynomial algorithm should really turn out to be faulty, it might not be completely useless, possibly some changes suffice to make the solver work again (I experienced this often during development).
- I created several different implementations, maybe a more complicated one that hasn't yet been published might work.
- However, although I tested the algorithm intensively and tried to construct an algorithm that can be proven to be correct, it could nevertheless be that the algorithm is faulty.
- If you think you found an error, please mail me: louis@louis-coder.com. Thank you.

11 Advantages compared to common Algorithms

In the following I want to list some suppositions why my algorithms shall work better than exponential, existing ones. The task shall be to solve exact 3-SAT formulas.

My algorithm is better than ... because ...:

Logical resolution: for exact 2-SAT CNFs, the resolution terminates after polynomially many steps. For exact-3-SAT this is not guaranteed. For instance, when using the resolution on large-enough pigeon hole problems any resolution-based solver will use exponentially many steps. Professor Haken proved this in 1985 [1]. The reason for the exponential complexity is, in my own words, the following:
When resolving two exact 3-SAT clauses with each 3 literals, a resolved clause may be created that has 4 literals. This 4-literal clause might, when being involved in further resolving steps, grow to a 5-literal clause and so on. This means that the number of (resolved) clauses, and thus also the complexity, can grow heavily. In contrast, two 2-SAT clauses can only be resolved to another 2-SAT clause, the literal count will not grow as the resolution removes one literal per input clause and puts the resting two or less literals in the resolved clause. Note that my algorithm does not produce "inbetween-data" like the resolution might create resolved clauses with 4, 5, 6 and more literals. My algorithm will process each triple of possible clauses only once, by design. The question is not if my solver processes each triple only once but if this one-time-treatment is sufficient. I suppose that it is, as millions of tests did not lead to any error and furthermore I suppose the induction proof confirms the correctness. For the induction proof, please read the algorithm explanation (in the zip file downloadable from the URL mentioned in topic 1.1).

Backtracking algorithms (e.g. DPLL): My algorithm does not use a binary search tree and also no backtracking, which might make an algorithm exponential. My algorithm only uses triples created out of the possible clauses, of which there are merely polynomially many ones.

Furthermore notice that my algorithm does not use:
- a pow() function to calculate loop iteration counts,
- it does never process a significantly large amount of data multiple times,
- it does also not use exponential space (memory).

All those evidences and the tests and the proofs in the algorithm explanation document make me believe that my algorithm does really solve any exact 3-SAT formula in polynomial time. But it could nevertheless be that the algorithm is incorrect. I ask the scientific community to verify my algorithm, for what I would be very thankful.

12 Summary

This document should have given you more detailed information about what the polynomial solver application does and how to use it. For the explanation of the actual polynomial SAT-solving algorithm, please download the document 'Polynomial_3-SAT_Solver.pdf' from:

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

13 References

13.1 General Literature

- 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.
- http://en.wikipedia.org/wiki/3-SAT (accessed 2013-11-23).

13.2 Concrete References

- [1] http://www.ti.inf.ethz.ch/ew/courses/extremal04/raemy.pdf (accessed 2013-12-08).