System for automatic generation of logical formulas

Radosław Klimek[1,*], *Katarzyna* Grobler-Dębska[2], and *Edyta* Kucharska[2]

[1] AGH University of Science and Technology, Department of Computer Science, Al. Mickiewicza 30, 30-059 Kraków
[2] AGH University of Science and Technology, Department of Automatics and Robotics, Al. Mickiewicza 30, 30-059 Kraków

Abstract. The satisfiability problem (SAT) is one of the classical and also most important problems of the theoretical computer science and has a direct bearing on numerous practical cases. It is one of the most prominent problems in artificial intelligence and has important applications in many fields, such as hardware and software verification, test-case generation, AI planning, scheduling, and data structures that allow efficient implementation of search space pruning. In recent years, there has been a huge development in SAT solvers, especially CDCL-based solvers (Conflict-Driven Clause-Learning) for propositional logic formulas. The goal of this paper is to design and implement a simple but effective system for random generation of long and complex logical formulas with a variety of difficulties encoded inside. The resulting logical formulas, *i.e.* problem instances, could be used for testing existing SAT solvers. The entire system would be widely available as a web application in the client-server architecture. The proposed system enables generation of syntactically correct logical formulas with a random structure, encoded in a manner understandable to SAT Solvers. Logical formulas can be presented in different formats. A number of parameters affect the form of generated instances, their complexity and physical dimensions. The randomness factor can be entered to every generated formula. The developed application is easy to modify and open for further extensions. The final part of the paper describes examples of solvers' tests of logical formulas generated by the implemented generator.

1 Introduction

Satisfiability problem is a classical problem of theoretical computer science which finds more and more practical applications [1]. Satisfiability problem, always resolvable, belongs to the NP (nondeterministic polynomial time) complexity class which, in reality, means that for a general case it is not computationally reachable. On the other hand, in recent years, there has been an enormous progress made when it comes to SAT solvers. It was the result of finding numerous effective heuristics, which enable finding satisfiable solutions in a formula. However, the main principle is the fact that a logical formula has to be represented in the Conjunction Normal Form CNF. It is a certain limitation but, on the other hand, such a formula may be treated as a collection of requirements (conjunction) in relation to a particular problem.

The progress in question (regarding the CNF format) historically results from using, in the first place, a strategy for finding substitutions known as CDCL, which is Conflict-Driven Clause Learning. Those solvers may find a solution for an average problem of the 50,000 variables on an average computing device in a few seconds. Bigger problems, of about 1 million variables, are computationally reachable but it depends on the internal structure of a given problem.

Encoding problems in the form of a logical formula is a separate topic, which was not in the scope of this work. Basically, it is an open problem and there is no effective algorithm which enables automatic encoding of any problem. On the other hand, there is a need to generate formulas of a different complexity level which can be used as a source of data for the tested SAT solvers. The random formula generation seems to be a proper line of action.

The aim of this work is to present a simple but effective system for random generation of long and complex logical formulas of a different level of complexity; it will also include the presentation of parameterisation possibilities of a thus implemented generator. The numerous tests carried out for the generated random logical formulas constituted the input data for existing logical engines. The application is easy to modify and open for further development. The present work discusses the results of a basic version of the application prepared as part of [2], since when the system itself has been modified and is now available as a web application[**].

2 Preliminaries

The satisfiability problem is equivalent to a decision problem of searching for substitution of logical constants under variables so that the whole formula is satisfied. In other words, the satisfiability problem for a particular formula is equivalent to the negation of the answer to the question whether the negation of a formula is a tautology. There are a few most popular classes of SAT problems. This type of classification can have a significant meaning for a designed problem of generating logical formulas. Therefore [1]:

[*] Corresponding author: rklimek@agh.edu.pl

- k-SAT – every clause from the formula consists of at most *k* literals, often exactly *k* literals.
  Especially:
- 1-SAT – is a single conjunction,
- 2-SAT – a case of SAT for which its solution can be found in a polynomial time.
- 3-SAT - the most often discussed example of SAT. It belongs to NP complexity class so there is no way to solve it in a polynomial time. Its difficulty may range from trivial to very difficult cases,
- Horn-SAT – every clause has at most one non-negative literal,
- NAE3SAT (not-all-equal 3-satisfiability) – in this case, every clause has exactly three literals, ensuring that they are not equal to each other.

There are also many other similar problems, the example of which are formulas in which quantification of Boolean variables – *Quantified Boolean Formulas* QBF, is accepted. On the other hand, #SAT is an issue related to SAT and based on calculating the number of substitutions satisfying a given formula.

A huge development has been made since the CDCL strategy was proposed. It is based on DPLL algorithm (Davis–Putnam–Logemann–Loveland) and its significant improvement is known as the so-called clause learning, *i.e.* gathering new knowledge about a logical problem after detecting a conflict on the basis of the analysis of existing and available clauses. A number of further heuristics improving the approach have been introduced, *e.g.* non-chronological shifting back in the search tree, using lazy structures, restart strategies, heuristics of variable selection for substitutions and many others, which led to operational usability of SAT problem and many practical applications.

The work presented in this paper is related to the problem of generating logical formulas. However, the reference should be made to the existing solutions. Created by Henry Yuen and Joseph Bebel, ThoughSAT Project [3] is a browser generator of random logical formulas in DIMACS CNF format. It is based on the hard encoding of mathematical problems such as factorisation into primes or the subset sum problem. After choosing the coding type it is necessary to define proper parameters, whereas the next step consists in generating a file with an extension which may be used as the input data for a selected SAT solver. On the other hand, CNFgen [4] is a console application which enables generating 20 different logical formulas, *e.g.* the pigeonhole principle (Dirichlet's box principle), ordering principle, counting principle, pebbling formula, graph isomorphism formula, graph automorphism formula, Ramsey number formula, *etc*. The generators presented above are efficient, however, they have quite a high input threshold and require specific knowledge or a long time of analysis in order to be used effectively. Moreover, it is not possible to generate several files which have the same parameters (for example a number of clauses and variables) with different results. In the generators mentioned above, creating formulas is based on encoding a real problem. n the majority of cases they are not random ones, however, random generation is the aim of this work.

3 Notation of logical formulas

There are many ways of coding logical formulas. The files which consist of formulas are intended for solvers. However, it is desired to make the saving format readable for system users. The encoded problems may be huge and complex, which is reflected in notation. The issue of problem encoding itself has a fundamental meaning. In general, it is a very complicated problem, however, from the point of view of this work, it will be treated as less important one because the main emphasis is put on the saving format of formulas, expressing particular problems.

One of the most popular formats for a classical SAT problem is DIMACS format. It is connected with CNF form (Conjunction Normal Form) of the formulas. DIMACS files consist of ASCII chars. They have a few types of rows. The rows describe a preamble of the file and the rows consisting of clauses are separate.

The preamble contains data describing a logical problem, which is saved in the subsequent rows. Every row starts from a single sign which describes a type of row. There may be:

- comments – enable saving additional information about a file – the information is clear for a human but ignored by reasoning engines and other programs. Comment lines appear at the beginning of the preamble starting from the „c" letter;
- problem line – has to be placed before any clause lines, at the end of the preamble.

In a file there can be only one problem line together with parameters corresponding to the file. It has the following structure:

```
p FORMAT VARIABLES CLAUSES
```

where: p – defines a line of a problem, FORMAT – describes a kind of problem, in case of CNF it will be a sequence of letters "cnf", VARIABLES – consists of the number of variables in a clause, CLAUSES – consists of the number of clauses in a formula.

On the other hand, the rows of clauses consist of natural numbers, preceded (or not) by a minus sign, where the numbers signify clause literals and a minus sign is a variable negation. All variables are numbered from 1 to n and those numbers are used to signify a particular variable. The zero number marks the end of a clause or, in other words, zero is a separator of subsequent clauses in a formula. In a given clause there have to appear all variables. For example, for the formula:

(x1 | -x3 | x4) ^ (-x4) ^ (-x2 | x3)

file in DIMACS would have the form as follows:

```
c sample DIMACS file
c
p cnf 4 3
1 -3 4 0
-4 0
-2 3 0
```

The other popular format is QCIR which enables to represent QBF formulas. The file in QCIR form consists of the following elements:

- comments – every line starting from "#" sign is considered to be a comment and can be ignored,
- ID of a format – sequence of signs after which n number may appear – it describes the maximal number of variables in a formula and later the sign of a new line,
- free variables – signify non-quantified variables, they are saved in the form:
  `free(var1 var2... vark)`
- quantifiers (quant) – enable lining of variables by quantifiers in one of the forms:
- `exists(var1; var2; : : : ; vark)` – signifies a precise quantifier
- `forall(var1; var2; : : : ; vark)` – signifies a general quantifier

Quantifiers, which are placed before the input declaration, are used to link the whole formula by a quantifier. In order to use a quantifier in the part of a formula, it needs to be used in a proper gate declaration.

- Output declaration – it has to be a gate variable and it describes the gate in a circuit.
- The variables, not connected in any way with the output, will be neglected. Output declaration takes the form of `output (gvar).`

– Gate declarations – enable defining the way of describing a value of a gate variable.

The QCIR format will not be discussed further as it is not the main topic of this work. To sum up, the language syntax in BNF form, not including comment lines is the following:

```
   qcir-file ::= format-id qblock-stmt*
output-stmt gate-stmt*
   format-id ::= #QCIR-13
   qblock-stmt ::= quant(var* )
   output-stmt ::= output(lit)
   gate-stmt ::= var = gate type(lit * )
```

Because QCIR files with formulas can be converted to DIMACS format, in the end we get such files, before [5]:

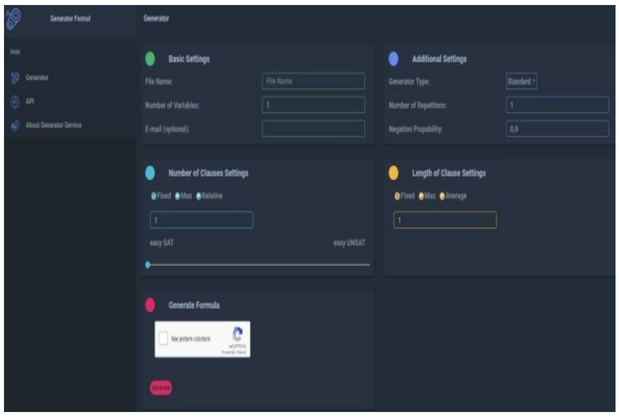
```
#QCIR-G14
forall(v1)
exists(v2, v3)
output(g3)
g1 = and(v1, v2)
g2 = and(-v1, -v2, v3)
g3 = or(g1, g2)
```

and after conversion:

```
c VarName   1 : v1
c VarName   2 : v2
c VarName   3 : v3
p cnf 6 11
a 1  0
e 2 3 4 5 6  0
6  0
4 -1 -2  0
-4 1  0
-4 2  0
5 1 2 -3  0
-5 -1  0
-5 -2  0
-5 3  0
-6 4 5  0
6 -4  0
6 -5  0
```

4 Generator

The aim of this application was to create a system for generating logical formulas in the DIMACS form. An important factor, distinguishing this application, is its easy handling, formula randomness and parameterisation of a formula generation process. It needs to be emphasised that this is a completely new approach, different from the existing solutions, which are rather based on generating formulas on the basis of coding some well-known problems. Obviously, the random factor, which is randomised, results in the fact that even for the identical input parameters there are created different output results. The present system is based on the work [2]; however, it has been adjusted to the web requirements. Figure 1 shows the screenshot of the system.



File Name: File Name
Number of Variables: 1
E-mail (optional):
Generator Type: Standard
Number of Repetitons: 1
Negation Propability: 0,0
Fixed Max Relative
1
easy SAT easy UNSAT
Fixed Max Average
1

Fig. 1. Main system window – the screenshot.

Parameterisation of formula generation process includes the following elements grouped in four sections:

1. Basic settings – include file name, its following number in a file sequence and a number of variables in the whole formula. The maximal number is *50,000*. If this number is not specified *i.e.* if it is given 0 number, the number of variables will be drawn from the range *(0;50000>*.
2. Additional settings – include three parameters:
   a. Generator type – allows choosing one of many generators and probability distributions available in the library [6]. It is a state-of-the-art software which was used to generate formulas. It is necessary to mention that the formulas generated in this way may be unsatisfiable which, also needs to be a subject of analysis as UNSAT formulas. Those cases are more computationally challenging than finding, usually the first, satisfiable solution. UNSAT is also a separate area of competition in the organised contests of solvers.
   b. Number of repetitions – describes the number of generated files which have the same parameters. The number may have a value in the range *<0;50>*. Those files may be different from one other but also similar - in the sense that they have similar principles.

c. Negation probability – describes a probability with which the variables in a clause will be negated. It is a floating point number from the range *<0;1>*.

3. Number of clauses settings – includes information about a number of clauses in a formula. One of three options are available:
   Fixed – constant/defined number of clauses, a parameter which can take values from the range *<1;2000000>*.
   Max – it is a maximal permissible number of clauses, the parameter can take values from the range *<0;2000000>*, if *0* number is given, the number of clauses will be randomly taken as an integer from the range *<0;2000000>*.
   Relative – the number of clauses will be calculated in a given ratio to a number of variables within a clause within the range *<2;8>*. It can be a floating point number expressed to two decimal places. The range was chosen on the basis of the article [7].
4. Length of clause settings – includes information related to the lengths of particular clauses. It may be selected as one of three values:
5. Fixed – the length of every clause fixed and described by a user, it is possible to select values from the range *<1;200>*.
   Max – a given number is a maximal acceptable clause length and the real value will be randomly chosen from the range *<0;200>*. If a set parameter has a value *0*, the maximal value will be randomly selected and, later on, this parameter will be interpreted as if it was set by a user.
   Average – an average clause length, the lengths of every separate clause will be randomly chosen as integers from the range <0;100>.

Exemplary parameterisation of a generator was presented in Figures 2 and 3.

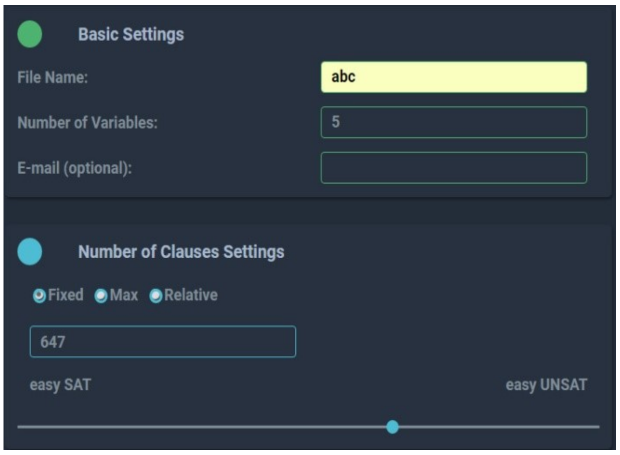

Basic Settings
File Name: abc
Number of Variables: 5
E-mail (optional):
Number of Clauses Settings
Fixed Max Relative
647
easy SAT
easy UNSAT

Fig. 2. Exemplary parameterisation of a generator – a screenshot.

The system generates files whose size is a result of set parameters and is proportional to the number of clauses and their length. Figure 4 illustrates that very well. There were presented files with constant clause lengths.

For example, for the formulas of a ratio of clause to a number of variables equal 4.2, the probability of negation of variable *0.5* and fixed clause of 3-literal length, the average file size for a *100* generation sample is 6 KB for *100* variables, 67 KB for *1000* variables, 787 KB for *10000* variables and 4478 KB for *50000* variables [2] respectively.

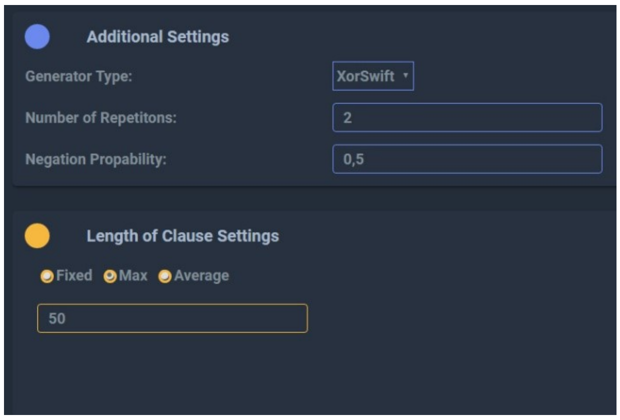

Additional Settings
Generator Type: XorSwift
Number of Repetitons: 2
Negation Propability: 0,5
Length of Clause Settings
Fixed Max Average
50

Fig. 3. Exemplary parameterisation of a generator – another screenshot.

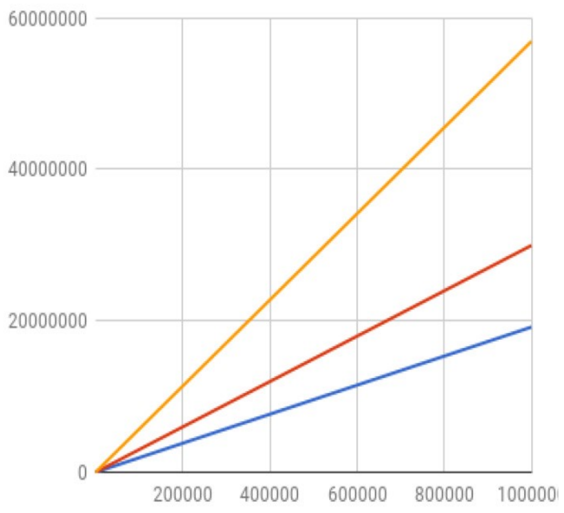


Fig. 4. The size of file in bytes (vertical axis) depends on a number of clauses in a formula (blue: 3 - literal clauses, red: 5- literal clauses, yellow: 10 – literal clauses) [2].

The work of a generator does not raise any concerns because it is stable regardless of the size of a generated file. It results from the fact that in an operational memory only one, last clause is saved, and the previous clauses are saved in an external file. Such a way of proceeding does not overload an operational memory.

The formulas generated by the system were tested using many solvers. The results of those tests were in accordance with the ones presented in the work [8]. The formulas, depending on the difficulty level, were successfully solved.

The well-known work [6] described a link between formula difficulty in relation to the satisfiability for k-

SAT formula of a fixed clause length, according to the ratio of a number of clauses; to the number of variables in a formula, as it was presented in Figure 5. Our studies confirm these results, *i.e.* cover their range.

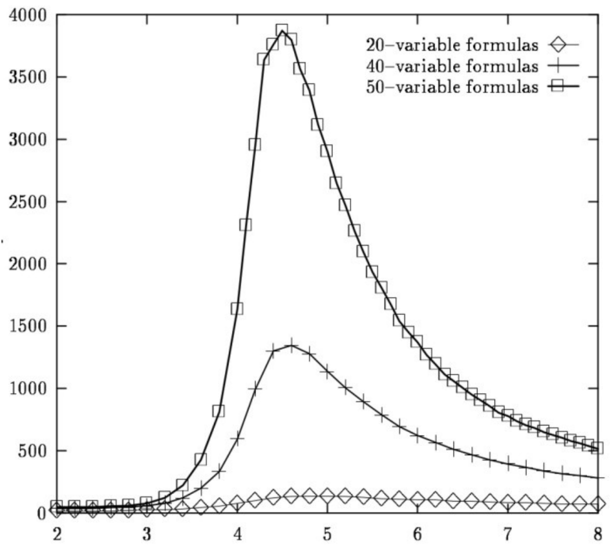


Fig. 5. Relation between formula difficulty and the ratio of a number of variables for 3-SAT formula [7].

It was shown that, when testing the formula, the higher number of call-backs, *DP calls*, appearing in a search tree, the more significant impact it has on a formula difficulty and, as a result, on a time of testing. For a 3-SAT problem it gives a value of around *4.5*; for 2-SAT it is around *1*; for k-SAT, where *k>3*, there is no unambiguous result. The work mentioned before [7] observed the following properties of 3-SAT formulas:

1. for a ratio of *2* the formulas are easy and satisfiable. This type of formula is underconstrained which enables finding a rapid satisfiable solution;
2. for a ratio of around *4.5* the formulas are very difficult because they require many call-backs, as the number of subformulas of this ratio which are satisfiable and unsatisfiable is almost the same. When solving such formulas it is necessary to check a huge number of substitutions before we being classified as SAT or UNSAT;
3. for a ratio higher than *5.5* the formulas are easy to test but they usually belong to UNSAT. This type of formula is overconstrained which enables finding rapid solutions in a search tree until UNSAT is concluded.

The work [2] reports obtaining a difficult formula of the following parameters: number of variables: *32000*; the ratio of number of clauses to the number of variables: *4.2* which is equivalent *134400* clauses; clause length: *3*; the probability of negation of a literal: *0.5*. This formula was not solved in spite of 24-hour testing (Riss, MiniSAT, Lingeling, Glucose).

5 Conclusions

The paper reports on the creation of the system for automatic generation of logical formulas in CNF form in DIMACS format. This system was designed as a service which, thanks to the web application, can be widely accessible. It is especially important in a situation when it is necessary to get formulas of a different difficulty level as test data for the existing reasoning engines or new solvers. It is also possible to carry on the own research studies related to the problem of generating difficult cases for k-SAT problems with different k. Finally, an interesting direction of research is the use of various probability distributions when generating the formulas.

This research is supported by AGH University of Science and Technology, Contract Nos.: 11.11.120.859, 11.11.120.417 and 15.11.120.625. The planned website for this service is http://forgen.agh.edu.pl.

References

1. A. Biere, M. Heule, H. van Maaren, and T. Walsh, Handbook of Satisfiability: (Volume 185 Frontiers in Artificial Intelligence and Applications.) Amsterdam, IOS Press (2009)
2. K. Burczyk, System losowego generowania formuł logicznych/System for randomly generated logical formulas, Engineering diploma thesis, supervisor: Radosław Klimek, AGH University of Science and Technology (2018)
3. H. Yuen i J. Bebel. Website ToughtSAT. URL: https://toughsat.appspot.com/ (accessed 2017-12-10)
4. M. Lauria. Website CNFgen. URL: https://massimolauria.github.io/cnfgen/ (accessed 2017-12-05)
5. W. Kleiber: QCIR-to-QDIMACS converter. URL: https://www.wklieber.com/ghostq/qcir-converter.html (accessed 2018.07.13)
6. S. Troschütz. Website Project Troschuetz. URL: https://www.codeproject.com/articles/15102/netrandom-number-generators-and-distributions (accessed 2018-01-01)
7. B. Selman, D. Mitchelly i H. Levesquez. Generating Hard Satisfiability Problems. *Artificial Intelligence*, **81**, (1996)
8. R. Klimek, Exploration of human activities using message streaming brokers and automated logical reasoning for ambient-assisted services, *IEEE Access*, **6**, (2018)